

---

**d3rlpy**

**Aug 11, 2020**



<b>1</b>	<b>Jupyter Notebooks</b>	<b>3</b>
<b>2</b>	<b>API Reference</b>	<b>5</b>
2.1	Algorithms . . . . .	5
2.2	Q Functions . . . . .	79
2.3	MDPDataSet . . . . .	80
2.4	Datasets . . . . .	88
2.5	Preprocessing . . . . .	90
2.6	Data Augmentation . . . . .	95
2.7	Metrics . . . . .	99
2.8	Save and Load . . . . .	106
2.9	Logging . . . . .	107
2.10	scikit-learn compatibility . . . . .	108
2.11	Online Training . . . . .	110
2.12	Model-based Data Augmentation . . . . .	114
<b>3</b>	<b>Installation</b>	<b>121</b>
3.1	Recommended Platforms . . . . .	121
3.2	from pip . . . . .	121
3.3	from source . . . . .	121
<b>4</b>	<b>License</b>	<b>123</b>
<b>5</b>	<b>Indices and tables</b>	<b>125</b>
	<b>Python Module Index</b>	<b>127</b>
	<b>Index</b>	<b>129</b>



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

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

These are the design principles of **d3rlpy**:

- **d3rlpy** is desined for practical projects.
- **d3rlpy** provides state-of-the-art techniques across RL researches to let everyone easily achieve an incredible performance.
- **d3rlpy** provides scikit-learn-compatible APIs.



# CHAPTER 1

---

## Jupyter Notebooks

---

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





### 2.1 Algorithms

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

#### 2.1.1 Continuous control algorithms

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

#### d3rlpy.algos.BC

```
class d3rlpy.algos.BC(learning_rate=0.001, batch_size=100, eps=1e-08, use_batch_norm=False,  
                    n_epochs=1000, use_gpu=False, scaler=None, augmentation=[],  
                    n_augmentations=1, encoder_params={}, dynamics=None, impl=None,  
                    **kwargs)
```

Behavior Cloning algorithm.

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

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

**Parameters**

- **learning\_rate** (*float*) – learning rate.
- **batch\_size** (*int*) – mini-batch size.
- **eps** (*float*) –  $\epsilon$  for Adam optimizer.
- **use\_batch\_norm** (*bool*) – flag to insert batch normalization layers.
- **n\_epochs** (*int*) – the number of epochs to train.
- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.
- **scaler** (*d3rlpy.preprocessingScaler* or *str*) – preprocessor. The available options are ['pixel', 'min\_max', 'standard']
- **augmentation** (*d3rlpy.augmentation.AugmentationPipeline* or *list(str)*) – augmentation pipeline.
- **n\_augmentations** (*int*) – the number of data augmentations to update.
- **encoder\_params** (*dict*) – optional arguments for encoder setup. If the observation is pixel, you can pass *filters* with list of tuples consisting with (*filter\_size*, *kernel\_size*, *stride*) and *feature\_size* with an integer scaler for the last linear layer size. If the observation is vector, you can pass *hidden\_units* with list of hidden unit sizes.
- **dynamics** (*d3rlpy.dynamics.base.DynamicsBase*) – dynamics model for data augmentation.
- **impl** (*d3rlpy.algos.torch.bc\_impl.BCImpl*) – implementation of the algorithm.

**n\_epochs**

the number of epochs to train.

**Type** *int*

**batch\_size**

mini-batch size.

**Type** *int*

**learning\_rate**

learning rate.

**Type** *float*

**eps**

$\epsilon$  for Adam optimizer.

**Type** *float*

**use\_batch\_norm**

flag to insert batch normalization layers.

**Type** *bool*

**use\_gpu**

GPU device.

**Type** *d3rlpy.gpu.Device*

**scaler**

preprocessor.

**Type** `d3rlpy.preprocessing.Scaler`

**augmentation**

augmentation pipeline.

**Type** `d3rlpy.augmentation.AugmentationPipeline`

**n\_augmentations**

the number of data augmentations to update.

**Type** `int`

**encoder\_params**

optional arguments for encoder setup.

**Type** `dict`

**dynamics**

dynamics model.

**Type** `d3rlpy.dynamics.base.DynamicsBase`

**impl**

implemenation of the algorithm.

**Type** `d3rlpy.algos.torch.bc_impl.BCImpl`

## Methods

**create\_impl** (*observation\_shape*, *action\_size*)

Instantiate implementation objects with the dataset shapes.

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

### Parameters

- **observation\_shape** (*tuple*) – observation shape.
- **action\_size** (*int*) – dimension of action-space.

**fit** (*episodes*, *experiment\_name=None*, *with\_timestamp=True*, *logdir='d3rlpy\_logs'*, *verbose=True*, *show\_progress=True*, *tensorboard=True*, *eval\_episodes=None*, *save\_interval=1*, *scorers=None*)  
Trains with the given dataset.

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

### Parameters

- **episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to train.
- **experiment\_name** (*str*) – experiment name for logging. If not passed, the directory name will be `{class name}_{timestamp}`.
- **with\_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show\_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **eval\_episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to test.

- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*list (callable)*) – list of scorer functions used with *eval\_episodes*.

**classmethod from\_json** (*fname, use\_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

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

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

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

#### Parameters

- **fname** (*str*) – file path to *params.json*.
- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.

**Returns** algorithm.

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

**get\_params** (*deep=True*)

Returns the all attributes.

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

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

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

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

**Returns** attribute values in dictionary.

**Return type** *dict*

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

Load neural network parameters.

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

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

**predict** (*x*)

Returns greedy actions.

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

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

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

**Returns** greedy actions

**Return type** *numpy.ndarray*

**predict\_value** (*x, action*)

value prediction is not supported by BC algorithms.

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

sampling action is not supported by BC algorithm.

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

Saves neural network parameters.

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

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

**save\_policy** (*fname*)

Save the greedy-policy computational graph as TorchScript.

```
algo.save_policy('policy.pt')
```

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

See also

- [https://pytorch.org/tutorials/beginner/Intro\\_to\\_TorchScript\\_tutorial.html](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](https://pytorch.org/tutorials/advanced/cpp_export.html) (for C++).

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

**set\_params** (*\*\*params*)

Sets the given arguments to the attributes if they exist.

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

```
algo.set_params(n_epochs=10, batch_size=100)
```

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

**Returns** itself.

**Return type** *d3rlpy.algos.base.AlgoBase*

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

Update parameters with mini-batch of data.

**Parameters**

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

**Returns** loss values.

**Return type** `list`

**d3rlpy.algos.DDPG**

```
class d3rlpy.algos.DDPG(actor_learning_rate=0.0003, critic_learning_rate=0.0003,
                        batch_size=100, gamma=0.99, tau=0.005, n_critics=1, bootstrap=False,
                        share_encoder=False, reguralizing_rate=1e-10, eps=1e-08, use_batch_norm=False,
                        q_func_type='mean', n_epochs=1000, use_gpu=False, scaler=None,
                        augmentation=[], n_augmentations=1, encoder_params={}, dynamics=None, impl=None, **kwargs)
```

Deep Deterministic Policy Gradients algorithm.

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

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

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

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

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

**References**

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

**Parameters**

- **actor\_learning\_rate** (*float*) – learning rate for policy function.
- **critic\_learning\_rate** (*float*) – learning rate for Q function.
- **batch\_size** (*int*) – mini-batch size.
- **gamma** (*float*) – discount factor.
- **tau** (*float*) – target network synchronization coefficient.
- **n\_critics** (*int*) – the number of Q functions for ensemble.
- **bootstrap** (*bool*) – flag to bootstrap Q functions.
- **share\_encoder** (*bool*) – flag to share encoder network.
- **reguralizing\_rate** (*float*) – reguralizing term for policy function.
- **eps** (*float*) –  $\epsilon$  for Adam optimizer.

- **use\_batch\_norm** (*bool*) – flag to insert batch normalization layers.
- **q\_func\_type** (*str*) – type of Q function. Available options are ['mean', 'qr', 'iqn', 'fqf'].
- **n\_epochs** (*int*) – the number of epochs to train.
- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.
- **scaler** (*d3rlpy.preprocessingScaler* or *str*) – preprocessor. The available options are ['pixel', 'min\_max', 'standard']
- **augmentation** (*d3rlpy.augmentation.AugmentationPipeline* or *list(str)*) – augmentation pipeline.
- **n\_augmentations** (*int*) – the number of data augmentations to update.
- **encoder\_params** (*dict*) – optional arguments for encoder setup. If the observation is pixel, you can pass filters with list of tuples consisting with (filter\_size, kernel\_size, stride) and feature\_size with an integer scaler for the last linear layer size. If the observation is vector, you can pass hidden\_units with list of hidden unit sizes.
- **dynamics** (*d3rlpy.dynamics.base.DynamicsBase*) – dynamics model for data augmentation.
- **impl** (*d3rlpy.algos.torch.ddpg\_impl.DDPGImpl*) – algorithm implementation.

**actor\_learning\_rate**

learning rate for policy function.

**Type** float**critic\_learning\_rate**

learning rate for Q function.

**Type** float**batch\_size**

mini-batch size.

**Type** int**gamma**

discount factor.

**Type** float**tau**

target network synchronization coefficient.

**Type** float**n\_critics**

the number of Q functions for ensemble.

**Type** int**bootstrap**

flag to bootstrap Q functions.

**Type** bool

**share\_encoder**  
flag to share encoder network.  
**Type** `bool`

**reguralizing\_rate**  
reguralizing term for policy function.  
**Type** `float`

**eps**  
 $\epsilon$  for Adam optimizer.  
**Type** `float`

**use\_batch\_norm**  
flag to insert batch normalization layers.  
**Type** `bool`

**q\_func\_type**  
type of Q function.  
**Type** `str`

**n\_epochs**  
the number of epochs to train.  
**Type** `int`

**use\_gpu**  
GPU device.  
**Type** `d3rlpy.gpu.Device`

**scaler**  
preprocessor.  
**Type** `d3rlpy.preprocessing.Scaler`

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

**n\_augmentations**  
the number of data augmentations to update.  
**Type** `int`

**encoder\_params**  
optional arguments for encoder setup.  
**Type** `dict`

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

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



## Methods

**create\_impl** (*observation\_shape*, *action\_size*)

Instantiate implementation objects with the dataset shapes.

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

### Parameters

- **observation\_shape** (*tuple*) – observation shape.
- **action\_size** (*int*) – dimension of action-space.

**fit** (*episodes*, *experiment\_name=None*, *with\_timestamp=True*, *logdir='d3rlpy\_logs'*, *verbose=True*, *show\_progress=True*, *tensorboard=True*, *eval\_episodes=None*, *save\_interval=1*, *scorers=None*)  
Trains with the given dataset.

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

### Parameters

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

**classmethod from\_json** (*fname*, *use\_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

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

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

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

### Parameters

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

- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.

**Returns** algorithm.

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

**get\_params** (*deep=True*)

Returns the all attributes.

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

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

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

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

**Returns** attribute values in dictionary.

**Return type** *dict*

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

Load neural network parameters.

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

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

**predict** (*x*)

Returns greedy actions.

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

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

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

**Returns** greedy actions

**Return type** *numpy.ndarray*

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

Returns predicted action-values.

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

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

# for discrete control
# 100 actions in integer values
```

(continues on next page)

(continued from previous page)

```

actions = np.random.randint(2, size=100)

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

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

```

**Parameters**

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

**Returns** predicted action-values**Return type** *numpy.ndarray***sample\_action** (*x*)

Returns sampled actions.

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

Saves neural network parameters.

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

**Parameters** **fname** (*str*) – destination file path.**save\_policy** (*fname*)

Save the greedy-policy computational graph as TorchScript.

```
algo.save_policy('policy.pt')
```

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

See also

- [https://pytorch.org/tutorials/beginner/Intro\\_to\\_TorchScript\\_tutorial.html](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](https://pytorch.org/tutorials/advanced/cpp_export.html) (for C++).

**Parameters** **fname** (*str*) – destination file path.**set\_params** (*\*\*params*)

Sets the given arguments to the attributes if they exist.

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

```
algo.set_params(n_epochs=10, batch_size=100)
```

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

**Returns** itself.

**Return type** d3rlpy.algos.base.AlgoBase

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

Update parameters with mini-batch of data.

**Parameters**

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

**Returns** loss values.

**Return type** list

## d3rlpy.algos.TD3

```
class d3rlpy.algos.TD3(actor_learning_rate=0.0003, critic_learning_rate=0.0003, batch_size=100,
                       gamma=0.99, tau=0.005, regularizing_rate=0.0, n_critics=2, boot-
                       strap=False, share_encoder=False, target_smoothing_sigma=0.2,
                       target_smoothing_clip=0.5, update_actor_interval=2, eps=1e-08,
                       use_batch_norm=False, q_func_type='mean', n_epochs=1000,
                       use_gpu=False, scaler=None, augmentation=[], n_augmentations=1,
                       encoder_params={}, dynamics=None, impl=None, **kwargs)
```

Twin Delayed Deep Deterministic Policy Gradients algorithm.

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

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

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

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

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

## References

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

**Parameters**

- **actor\_learning\_rate** (*float*) – learning rate for a policy function.

- **critic\_learning\_rate** (*float*) – learning rate for Q functions.
- **batch\_size** (*int*) – mini-batch size.
- **gamma** (*float*) – discount factor.
- **tau** (*float*) – target network synchronization coefficient.
- **reguralizing\_rate** (*float*) – reguralizing term for policy function.
- **n\_critics** (*int*) – the number of Q functions for ensemble.
- **bootstrap** (*bool*) – flag to bootstrap Q functions.
- **share\_encoder** (*bool*) – flag to share encoder network.
- **target\_smoothing\_sigma** (*float*) – standard deviation for target noise.
- **target\_smoothing\_clip** (*float*) – clipping range for target noise.
- **update\_actor\_interval** (*int*) – interval to update policy function described as *delayed policy update* in the paper.
- **eps** (*float*) –  $\epsilon$  for Adam optimizer.
- **use\_batch\_norm** (*bool*) – flag to insert batch normalization layers.
- **q\_func\_type** (*str*) – type of Q function. Available options are ['mean', 'qr', 'iqn', 'fqf'].
- **n\_epochs** (*int*) – the number of epochs to train.
- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.
- **scaler** (*d3rlpy.preprocessingScaler* or *str*) – preprocessor. The available options are ['pixel', 'min\_max', 'standard']
- **augmentation** (*d3rlpy.augmentation.AugmentationPipeline* or *list(str)*) – augmentation pipeline.
- **n\_augmentations** (*int*) – the number of data augmentations to update.
- **encoder\_params** (*dict*) – optional arguments for encoder setup. If the observation is pixel, you can pass filters with list of tuples consisting with (filter\_size, kernel\_size, stride) and feature\_size with an integer scaler for the last linear layer size. If the observation is vector, you can pass hidden\_units with list of hidden unit sizes.
- **dynamics** (*d3rlpy.dynamics.base.DynamicsBase*) – dynamics model for data augmentation.
- **impl** (*d3rlpy.algos.torch.td3\_impl.TD3Impl*) – algorithm implementation.

**actor\_learning\_rate**

learning rate for a policy function.

Type *float***critic\_learning\_rate**

learning rate for Q functions.

Type *float***batch\_size**

mini-batch size.

Type *int*

**gamma**  
discount factor.  
**Type** float

**tau**  
target network synchronization coefficient.  
**Type** float

**reguralizing\_rate**  
reguralizing term for policy function.  
**Type** float

**n\_critics**  
the number of Q functions for ensemble.  
**Type** int

**bootstrap**  
flag to bootstrap Q functions.  
**Type** bool

**share\_encoder**  
flag to share encoder network.  
**Type** bool

**target\_smoothing\_sigma**  
standard deviation for target noise.  
**Type** float

**target\_smoothing\_clip**  
clipping range for target noise.  
**Type** float

**update\_actor\_interval**  
interval to update policy function described as *delayed policy update* in the paper.  
**Type** int

**eps**  
 $\epsilon$  for Adam optimizer.  
**Type** float

**use\_batch\_norm**  
flag to insert batch normalization layers.  
**Type** bool

**q\_func\_type**  
type of Q function..  
**Type** str

**n\_epochs**  
the number of epochs to train.  
**Type** int

**use\_gpu**  
GPU device.

**Type** d3rlpy.gpu.Device

**scaler**  
preprocessor.

**Type** d3rlpy.preprocessing.Scaler

**augmentation**  
augmentation pipeline.

**Type** d3rlpy.augmentation.AugmentationPipeline

**n\_augmentations**  
the number of data augmentations to update.

**Type** int

**encoder\_params**  
optional arguments for encoder setup.

**Type** dict

**dynamics**  
dynamics model.

**Type** d3rlpy.dynamics.base.DynamicsBase

**impl**  
algorithm implementation.

**Type** d3rlpy.algos.torch.td3\_impl.TD3Impl

## Methods

**create\_impl** (*observation\_shape*, *action\_size*)  
Instantiate implementation objects with the dataset shapes.  
This method will be used internally when *fit* method is called.

### Parameters

- **observation\_shape** (*tuple*) – observation shape.
- **action\_size** (*int*) – dimension of action-space.

**fit** (*episodes*, *experiment\_name=None*, *with\_timestamp=True*, *logdir='d3rlpy\_logs'*, *verbose=True*, *show\_progress=True*, *tensorboard=True*, *eval\_episodes=None*, *save\_interval=1*, *scorers=None*)  
Trains with the given dataset.

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

### Parameters

- **episodes** (*list* (d3rlpy.dataset.Episode)) – list of episodes to train.
- **experiment\_name** (*str*) – experiment name for logging. If not passed, the directory name will be {class name}\_{timestamp}.
- **with\_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show\_progress** (*bool*) – flag to show progress bar for iterations.

- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **eval\_episodes** (*list* (*d3rlpy.dataset.Episode*)) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*list* (*callable*)) – list of scorer functions used with *eval\_episodes*.

**classmethod** **from\_json** (*fname*, *use\_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

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

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

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

#### Parameters

- **fname** (*str*) – file path to *params.json*.
- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.

**Returns** algorithm.

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

**get\_params** (*deep=True*)

Returns the all attributes.

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

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

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

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

**Returns** attribute values in dictionary.

**Return type** *dict*

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

Load neural network parameters.

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

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



**predict** (*x*)

Returns greedy actions.

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

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

**Parameters** *x* (*numpy.ndarray*) – observations**Returns** greedy actions**Return type** *numpy.ndarray***predict\_value** (*x*, *action*, *with\_std=False*)

Returns predicted action-values.

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

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

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

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

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

**Parameters**

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

**Returns** predicted action-values**Return type** *numpy.ndarray***sample\_action** (*x*)

Returns sampled actions.

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

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

Saves neural network parameters.

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

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

**save\_policy** (*fname*)

Save the greedy-policy computational graph as TorchScript.

```
algo.save_policy('policy.pt')
```

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

See also

- [https://pytorch.org/tutorials/beginner/Intro\\_to\\_TorchScript\\_tutorial.html](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](https://pytorch.org/tutorials/advanced/cpp_export.html) (for C++).

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

**set\_params** (*\*\*params*)

Sets the given arguments to the attributes if they exist.

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

```
algo.set_params(n_epochs=10, batch_size=100)
```

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

**Returns** itself.

**Return type** `d3rlpy.algos.base.AlgoBase`

**update** (*epoch*, *total\_step*, *batch*)

Update parameters with mini-batch of data.

**Parameters**

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

**Returns** loss values.

**Return type** `list`

## d3rlpy.algos.SAC

```
class d3rlpy.algos.SAC(actor_learning_rate=0.0003, critic_learning_rate=0.0003,
                       temp_learning_rate=0.0003, batch_size=100, gamma=0.99,
                       tau=0.005, n_critics=2, bootstrap=False, share_encoder=False,
                       update_actor_interval=2, initial_temperature=1.0, eps=1e-08,
                       use_batch_norm=False, q_func_type='mean', n_epochs=1000,
                       use_gpu=False, scaler=None, augmentation=[], n_augmentations=1,
                       encoder_params={}, dynamics=None, impl=None, **kwargs)
```

Soft Actor-Critic algorithm.

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

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

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

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

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

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

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

## References

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

### Parameters

- **actor\_learning\_rate** (*float*) – learning rate for policy function.
- **critic\_learning\_rate** (*float*) – learning rate for Q functions.
- **temp\_learning\_rate** (*float*) – learning rate for temperature parameter.
- **batch\_size** (*int*) – mini-batch size.
- **gamma** (*float*) – discount factor.
- **tau** (*float*) – target network synchronization coefficient.
- **n\_critics** (*int*) – the number of Q functions for ensemble.
- **bootstrap** (*bool*) – flag to bootstrap Q functions.
- **share\_encoder** (*bool*) – flag to share encoder network.
- **update\_actor\_interval** (*int*) – interval to update policy function.
- **initial\_temperature** (*float*) – initial temperature value.
- **eps** (*float*) –  $\epsilon$  for Adam optimizer.
- **use\_batch\_norm** (*bool*) – flag to insert batch normalization layers.

- **q\_func\_type** (*str*) – type of Q function. Available options are [*'mean'*, *'qr'*, *'iqn'*, *'fqf'*].
- **n\_epochs** (*int*) – the number of epochs to train.
- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.
- **scaler** (*d3rlpy.preprocessingScaler* or *str*) – preprocessor. The available options are [*'pixel'*, *'min\_max'*, *'standard'*]
- **augmentation** (*d3rlpy.augmentation.AugmentationPipeline* or *list(str)*) – augmentation pipeline.
- **n\_augmentations** (*int*) – the number of data augmentations to update.
- **encoder\_params** (*dict*) – optional arguments for encoder setup. If the observation is pixel, you can pass *filters* with list of tuples consisting with (*filter\_size*, *kernel\_size*, *stride*) and *feature\_size* with an integer scaler for the last linear layer size. If the observation is vector, you can pass *hidden\_units* with list of hidden unit sizes.
- **dynamics** (*d3rlpy.dynamics.base.DynamicsBase*) – dynamics model for data augmentation.
- **impl** (*d3rlpy.algos.torch.sac\_impl.SACImpl*) – algorithm implementation.

**actor\_learning\_rate**

learning rate for policy function.

Type *float*

**critic\_learning\_rate**

learning rate for Q functions.

Type *float*

**temp\_learning\_rate**

learning rate for temperature parameter.

Type *float*

**batch\_size**

mini-batch size.

Type *int*

**gamma**

discount factor.

Type *float*

**tau**

target network synchronization coefficient.

Type *float*

**n\_critics**

the number of Q functions for ensemble.

Type *int*

**bootstrap**

flag to bootstrap Q functions.

Type *bool*

**share\_encoder**  
flag to share encoder network.  
**Type** `bool`

**update\_actor\_interval**  
interval to update policy function.  
**Type** `int`

**initial\_temperature**  
initial temperature value.  
**Type** `float`

**eps**  
 $\epsilon$  for Adam optimizer.  
**Type** `float`

**use\_batch\_norm**  
flag to insert batch normalization layers.  
**Type** `bool`

**q\_func\_type**  
type of Q function.  
**Type** `str`

**n\_epochs**  
the number of epochs to train.  
**Type** `int`

**use\_gpu**  
GPU device.  
**Type** `d3rlpy.gpu.Device`

**scaler**  
preprocessor.  
**Type** `d3rlpy.preprocessing.Scaler`

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

**n\_augmentations**  
the number of data augmentations to update.  
**Type** `int`

**encoder\_params**  
optional arguments for encoder setup.  
**Type** `dict`

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

**impl**  
algorithm implementation.

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

## Methods

**create\_impl** (*observation\_shape*, *action\_size*)

Instantiate implementation objects with the dataset shapes.

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

### Parameters

- **observation\_shape** (*tuple*) – observation shape.
- **action\_size** (*int*) – dimension of action-space.

**fit** (*episodes*, *experiment\_name=None*, *with\_timestamp=True*, *logdir='d3rlpy\_logs'*, *verbose=True*, *show\_progress=True*, *tensorboard=True*, *eval\_episodes=None*, *save\_interval=1*, *scorers=None*)  
Trains with the given dataset.

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

### Parameters

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

**classmethod from\_json** (*fname*, *use\_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

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

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

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

### Parameters

- **fname** (*str*) – file path to *params.json*.
- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.

**Returns** algorithm.

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

**get\_params** (*deep=True*)

Returns the all attributes.

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

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

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

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

**Returns** attribute values in dictionary.

**Return type** *dict*

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

Load neural network parameters.

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

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

**predict** (*x*)

Returns greedy actions.

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

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

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

**Returns** greedy actions

**Return type** *numpy.ndarray*

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

Returns predicted action-values.

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

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

(continues on next page)

(continued from previous page)

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

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

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

**Parameters**

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

**Returns** predicted action-values**Return type** *numpy.ndarray***sample\_action** (*x*)

Returns sampled actions.

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

Saves neural network parameters.

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

**Parameters** **fname** (*str*) – destination file path.**save\_policy** (*fname*)

Save the greedy-policy computational graph as TorchScript.

```
algo.save_policy('policy.pt')
```

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

See also

- [https://pytorch.org/tutorials/beginner/Intro\\_to\\_TorchScript\\_tutorial.html](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](https://pytorch.org/tutorials/advanced/cpp_export.html) (for C++).

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



**set\_params** (\*\*params)

Sets the given arguments to the attributes if they exist.

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

```
algo.set_params(n_epochs=10, batch_size=100)
```

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

**Returns** itself.

**Return type** d3rlpy.algos.base.AlgoBase

**update** (epoch, total\_step, batch)

Update parameters with mini-batch of data.

**Parameters**

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

**Returns** loss values.

**Return type** list

## d3rlpy.algos.BCQ

```
class d3rlpy.algos.BCQ(actor_learning_rate=0.001, critic_learning_rate=0.001, im-
    itator_learning_rate=0.001, batch_size=100, gamma=0.99,
    tau=0.005, n_critics=2, bootstrap=False, share_encoder=False,
    update_actor_interval=1, lam=0.75, n_action_samples=100, ac-
    tion_flexibility=0.05, rl_start_epoch=0, latent_size=32, beta=0.5,
    eps=1e-08, use_batch_norm=False, q_func_type='mean', n_epochs=1000,
    use_gpu=False, scaler=None, augmentation=[], n_augmentations=1,
    encoder_params={}, dynamics=None, impl=None, **kwargs)
```

Batch-Constrained Q-learning algorithm.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

---

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

---

## References

- [Fujimoto et al., Off-Policy Deep Reinforcement Learning without Exploration.](#)

### Parameters

- **actor\_learning\_rate** (*float*) – learning rate for policy function.
- **critic\_learning\_rate** (*float*) – learning rate for Q functions.
- **imitator\_learning\_rate** (*float*) – learning rate for Conditional VAE.
- **batch\_size** (*int*) – mini-batch size.
- **gamma** (*float*) – discount factor.
- **tau** (*float*) – target network synchronization coefficient.
- **n\_critics** (*int*) – the number of Q functions for ensemble.
- **bootstrap** (*bool*) – flag to bootstrap Q functions.
- **share\_encoder** (*bool*) – flag to share encoder network.
- **update\_actor\_interval** (*int*) – interval to update policy function.
- **lam** (*float*) – weight factor for critic ensemble.
- **n\_action\_samples** (*int*) – the number of action samples to estimate action-values.
- **action\_flexibility** (*float*) – output scale of perturbation function represented as  $\Phi$ .
- **rl\_start\_epoch** (*int*) – epoch to start to update policy function and Q functions. If this is large, RL training would be more stabilized.
- **latent\_size** (*int*) – size of latent vector for Conditional VAE.

- **beta** (*float*) – KL regularization term for Conditional VAE.
- **eps** (*float*) –  $\epsilon$  for Adam optimizer.
- **use\_batch\_norm** (*bool*) – flag to insert batch normalization layers.
- **q\_func\_type** (*str*) – type of Q function. Available options are ['mean', 'qr', 'iqn', 'fqf'].
- **n\_epochs** (*int*) – the number of epochs to train.
- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.
- **scaler** (*d3rlpy.preprocessingScaler* or *str*) – preprocessor. The available options are ['pixel', 'min\_max', 'standard']
- **augmentation** (*d3rlpy.augmentation.AugmentationPipeline* or *list(str)*) – augmentation pipeline.
- **n\_augmentations** (*int*) – the number of data augmentations to update.
- **encoder\_params** (*dict*) – optional arguments for encoder setup. If the observation is pixel, you can pass filters with list of tuples consisting with (filter\_size, kernel\_size, stride) and feature\_size with an integer scaler for the last linear layer size. If the observation is vector, you can pass hidden\_units with list of hidden unit sizes.
- **dynamics** (*d3rlpy.dynamics.base.DynamicsBase*) – dynamics model for data augmentation.
- **impl** (*d3rlpy.algos.torch.bcq\_impl.BCQImpl*) – algorithm implementation.

**actor\_learning\_rate**

learning rate for policy function.

**Type** *float***critic\_learning\_rate**

learning rate for Q functions.

**Type** *float***imitator\_learning\_rate**

learning rate for Conditional VAE.

**Type** *float***batch\_size**

mini-batch size.

**Type** *int***gamma**

discount factor.

**Type** *float***tau**

target network synchronization coefficient.

**Type** *float***n\_critics**

the number of Q functions for ensemble.

**Type** *int*

**bootstrap**  
flag to bootstrap Q functions.  
Type `bool`

**share\_encoder**  
flag to share encoder network.  
Type `bool`

**update\_actor\_interval**  
interval to update policy function.  
Type `int`

**lam**  
weight factor for critic ensemble.  
Type `float`

**n\_action\_samples**  
the number of action samples to estimate action-values.  
Type `int`

**action\_flexibility**  
output scale of perturbation function.  
Type `float`

**rl\_start\_epoch**  
epoch to start to update policy function and Q functions.  
Type `int`

**latent\_size**  
size of latent vector for Conditional VAE.  
Type `int`

**beta**  
KL regularization term for Conditional VAE.  
Type `float`

**eps**  
 $\epsilon$  for Adam optimizer.  
Type `float`

**use\_batch\_norm**  
flag to insert batch normalization layers.  
Type `bool`

**q\_func\_type**  
type of Q function.  
Type `str`

**n\_epochs**  
the number of epochs to train.  
Type `int`

**use\_gpu**  
GPU device.

**Type** d3rlpy.gpu.Device

**scaler**  
preprocessor.

**Type** d3rlpy.preprocessing.Scaler

**augmentation**  
augmentation pipeline.

**Type** d3rlpy.augmentation.AugmentationPipeline

**n\_augmentations**  
the number of data augmentations to update.

**Type** int

**encoder\_params**  
optional arguments for encoder setup.

**Type** dict

**dynamics**  
dynamics model.

**Type** d3rlpy.dynamics.base.DynamicsBase

**impl**  
algorithm implementation.

**Type** d3rlpy.algos.torch.bcq\_impl.BCQImpl

## Methods

**create\_impl** (*observation\_shape*, *action\_size*)  
Instantiate implementation objects with the dataset shapes.  
This method will be used internally when *fit* method is called.

### Parameters

- **observation\_shape** (*tuple*) – observation shape.
- **action\_size** (*int*) – dimension of action-space.

**fit** (*episodes*, *experiment\_name=None*, *with\_timestamp=True*, *logdir='d3rlpy\_logs'*, *verbose=True*, *show\_progress=True*, *tensorboard=True*, *eval\_episodes=None*, *save\_interval=1*, *scorers=None*)  
Trains with the given dataset.

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

### Parameters

- **episodes** (*list* (d3rlpy.dataset.Episode)) – list of episodes to train.
- **experiment\_name** (*str*) – experiment name for logging. If not passed, the directory name will be {class name}\_{timestamp}.
- **with\_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show\_progress** (*bool*) – flag to show progress bar for iterations.

- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **eval\_episodes** (*list* (*d3rlpy.dataset.Episode*)) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*list* (*callable*)) – list of scorer functions used with *eval\_episodes*.

**classmethod** **from\_json** (*fname*, *use\_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

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

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

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

#### Parameters

- **fname** (*str*) – file path to *params.json*.
- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.

**Returns** algorithm.

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

**get\_params** (*deep=True*)

Returns the all attributes.

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

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

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

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

**Returns** attribute values in dictionary.

**Return type** *dict*

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

Load neural network parameters.

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

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

**predict** (*x*)

Returns greedy actions.

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

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

**Parameters** *x* (*numpy.ndarray*) – observations**Returns** greedy actions**Return type** *numpy.ndarray***predict\_value** (*x*, *action*, *with\_std=False*)

Returns predicted action-values.

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

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

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

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

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

**Parameters**

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

**Returns** predicted action-values**Return type** *numpy.ndarray***sample\_action** (*x*)

BCQ does not support sampling action.

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

Saves neural network parameters.

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

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

**save\_policy** (*fname*)

Save the greedy-policy computational graph as TorchScript.

```
algo.save_policy('policy.pt')
```

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

See also

- [https://pytorch.org/tutorials/beginner/Intro\\_to\\_TorchScript\\_tutorial.html](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](https://pytorch.org/tutorials/advanced/cpp_export.html) (for C++).

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

**set\_params** (*\*\*params*)

Sets the given arguments to the attributes if they exist.

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

```
algo.set_params(n_epochs=10, batch_size=100)
```

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

**Returns** itself.

**Return type** `d3rlpy.algos.base.AlgoBase`

**update** (*epoch*, *total\_step*, *batch*)

Update parameters with mini-batch of data.

**Parameters**

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

**Returns** loss values.

**Return type** `list`

**d3rlpy.algos.BEAR**

```
class d3rlpy.algos.BEAR(actor_learning_rate=0.0003, critic_learning_rate=0.0003, im-
    itator_learning_rate=0.001, temp_learning_rate=0.0003, al-
    pha_learning_rate=0.001, batch_size=100, gamma=0.99,
    tau=0.005, n_critics=2, bootstrap=False, share_encoder=False, up-
    date_actor_interval=1, initial_temperature=1.0, initial_alpha=1.0,
    alpha_threshold=0.05, lam=0.75, n_action_samples=4,
    mmd_sigma=20.0, rl_start_epoch=0, eps=1e-08, use_batch_norm=False,
    q_func_type='mean', n_epochs=1000, use_gpu=False, scaler=None, aug-
    mentation=[], n_augmentations=1, encoder_params={}, dynamics=None,
    impl=None, **kwargs)
```

Bootstrapping Error Accumulation Reduction algorithm.

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



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

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

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

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

where MMD is computed as follows.

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

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

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

## References

- Kumar et al., [Stabilizing Off-Policy Q-Learning via Bootstrapping Error Reduction](#).

### Parameters

- **actor\_learning\_rate** (*float*) – learning rate for policy function.
- **critic\_learning\_rate** (*float*) – learning rate for Q functions.
- **imitator\_learning\_rate** (*float*) – learning rate for behavior policy function.
- **temp\_learning\_rate** (*float*) – learning rate for temperature parameter.
- **alpha\_learning\_rate** (*float*) – learning rate for  $\alpha$ .
- **batch\_size** (*int*) – mini-batch size.
- **gamma** (*float*) – discount factor.
- **tau** (*float*) – target network synchronization coefficient.
- **n\_critics** (*int*) – the number of Q functions for ensemble.
- **bootstrap** (*bool*) – flag to bootstrap Q functions.
- **share\_encoder** (*bool*) – flag to share encoder network.
- **update\_actor\_interval** (*int*) – interval to update policy function.
- **initial\_temperature** (*float*) – initial temperature value.
- **initial\_alpha** (*float*) – initial  $\alpha$  value.
- **alpha\_threshold** (*float*) – threshold value described as  $\epsilon$ .
- **lam** (*float*) – weight for critic ensemble.
- **n\_action\_samples** (*int*) – the number of action samples to estimate action-values.
- **mmd\_sigma** (*float*) –  $\sigma$  for gaussian kernel in MMD calculation.
- **rl\_start\_epoch** (*int*) – epoch to start to update policy function and Q functions. If this is large, RL training would be more stabilized.

- **eps** (*float*) –  $\epsilon$  for Adam optimizer.
- **use\_batch\_norm** (*bool*) – flag to insert batch normalization layers.
- **q\_func\_type** (*str*) – type of Q function. Available options are ['mean', 'qr', 'iqn', 'fqf'].
- **n\_epochs** (*int*) – the number of epochs to train.
- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.
- **scaler** (*d3rlpy.preprocessingScaler* or *str*) – preprocessor. The available options are ['pixel', 'min\_max', 'standard'].
- **augmentation** (*d3rlpy.augmentation.AugmentationPipeline* or *list(str)*) – augmentation pipeline.
- **n\_augmentations** (*int*) – the number of data augmentations to update.
- **encoder\_params** (*dict*) – optional arguments for encoder setup. If the observation is pixel, you can pass filters with list of tuples consisting with (filter\_size, kernel\_size, stride) and feature\_size with an integer scaler for the last linear layer size. If the observation is vector, you can pass hidden\_units with list of hidden unit sizes.
- **dynamics** (*d3rlpy.dynamics.base.DynamicsBase*) – dynamics model for data augmentation.
- **impl** (*d3rlpy.algos.torch.bear\_impl.BEARImpl*) – algorithm implementation.

**actor\_learning\_rate**

learning rate for policy function.

Type *float*

**critic\_learning\_rate**

learning rate for Q functions.

Type *float*

**imitator\_learning\_rate**

learning rate for behavior policy function.

Type *float*

**temp\_learning\_rate**

learning rate for temperature parameter.

Type *float*

**alpha\_learning\_rate**

learning rate for  $\alpha$ .

Type *float*

**batch\_size**

mini-batch size.

Type *int*

**gamma**

discount factor.

Type *float*

<b>tau</b>	target network synchronization coefficient.
<b>Type</b>	float
<b>n_critics</b>	the number of Q functions for ensemble.
<b>Type</b>	int
<b>bootstrap</b>	flag to bootstrap Q functions.
<b>Type</b>	bool
<b>share_encoder</b>	flag to share encoder network.
<b>Type</b>	bool
<b>update_actor_interval</b>	interval to update policy function.
<b>Type</b>	int
<b>initial_temperature</b>	initial temperature value.
<b>Type</b>	float
<b>initial_alpha</b>	initial $\alpha$ value.
<b>Type</b>	float
<b>alpha_threshold</b>	threshold value described as $\epsilon$ .
<b>Type</b>	float
<b>lam</b>	weight for critic ensemble.
<b>Type</b>	float
<b>n_action_samples</b>	the number of action samples to estimate action-values.
<b>Type</b>	int
<b>mmd_sigma</b>	$\sigma$ for gaussian kernel in MMD calculation.
<b>Type</b>	float
<b>rl_start_epoch</b>	epoch to start to update policy function and Q functions. If this is large, RL training would be more stabilized.
<b>Type</b>	int
<b>eps</b>	$\epsilon$ for Adam optimizer.
<b>Type</b>	float

**use\_batch\_norm**  
flag to insert batch normalization layers.  
Type `bool`

**q\_func\_type**  
type of Q function..  
Type `str`

**n\_epochs**  
the number of epochs to train.  
Type `int`

**use\_gpu**  
GPU device.  
Type `d3rlpy.gpu.Device`

**scaler**  
preprocessor.  
Type `d3rlpy.preprocessing.Scaler`

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

**n\_augmentations**  
the number of data augmentations to update.  
Type `int`

**encoder\_params**  
optional arguments for encoder setup.  
Type `dict`

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

**impl**  
algorithm implementation.  
Type `d3rlpy.algos.torch.bear_impl.BEARImpl`

## Methods

**create\_impl** (*observation\_shape*, *action\_size*)  
Instantiate implementation objects with the dataset shapes.  
This method will be used internally when *fit* method is called.

### Parameters

- **observation\_shape** (*tuple*) – observation shape.
- **action\_size** (*int*) – dimension of action-space.

**fit** (*episodes*, *experiment\_name=None*, *with\_timestamp=True*, *logdir='d3rlpy\_logs'*, *verbose=True*, *show\_progress=True*, *tensorboard=True*, *eval\_episodes=None*, *save\_interval=1*, *scorers=None*)  
Trains with the given dataset.

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

#### Parameters

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

**classmethod from\_json** (*fname*, *use\_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

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

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

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

#### Parameters

- **fname** (*str*) – file path to *params.json*.
- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.

**Returns** algorithm.

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

**get\_params** (*deep=True*)

Returns the all attributes.

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

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

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

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

**Returns** attribute values in dictionary.

**Return type** `dict`

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

Load neural network parameters.

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

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

**predict** (*x*)

Returns greedy actions.

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

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

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

**Returns** greedy actions

**Return type** `numpy.ndarray`

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

Returns predicted action-values.

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

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

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

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

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

**Parameters**

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

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

**Returns** predicted action-values

**Return type** *numpy.ndarray*

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

Returns sampled actions.

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

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

**Returns** sampled actions.

**Return type** *numpy.ndarray*

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

Saves neural network parameters.

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

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

**save\_policy** (*fname*)

Save the greedy-policy computational graph as TorchScript.

```
algo.save_policy('policy.pt')
```

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

See also

- [https://pytorch.org/tutorials/beginner/Intro\\_to\\_TorchScript\\_tutorial.html](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](https://pytorch.org/tutorials/advanced/cpp_export.html) (for C++).

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

**set\_params** (*\*\*params*)

Sets the given arguments to the attributes if they exist.

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

```
algo.set_params(n_epochs=10, batch_size=100)
```

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

**Returns** itself.

**Return type** *d3rlpy.algos.base.AlgoBase*

**update** (*epoch, total\_step, batch*)

Update parameters with mini-batch of data.

**Parameters**

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

**Returns** loss values.

**Return type** `list`

**d3rlpy.algos.CQL**

```
class d3rlpy.algos.CQL (actor_learning_rate=3e-05,          critic_learning_rate=0.0003,
                        temp_learning_rate=3e-05, alpha_learning_rate=0.0003, batch_size=100,
                        gamma=0.99,      tau=0.005,      n_critics=2,      bootstrap=False,
                        share_encoder=False, update_actor_interval=1, initial_temperature=1.0,
                        initial_alpha=5.0, alpha_threshold=10.0, n_action_samples=10, eps=1e-08,
                        use_batch_norm=False, q_func_type='mean', n_epochs=1000,
                        use_gpu=False, scaler=None, augmentation=[], n_augmentations=1,
                        encoder_params={}, dynamics=None, impl=None, **kwargs)
```

Conservative Q-Learning algorithm.

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

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

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

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

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

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

where  $N$  is the number of sampled actions.

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

**References**

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

**Parameters**

- **actor\_learning\_rate** (*float*) – learning rate for policy function.
- **critic\_learning\_rate** (*float*) – learning rate for Q functions.
- **temp\_learning\_rate** (*float*) – learning rate for temperature parameter of SAC.
- **alpha\_learning\_rate** (*float*) – learning rate for  $\alpha$ .



- **batch\_size** (*int*) – mini-batch size.
- **gamma** (*float*) – discount factor.
- **tau** (*float*) – target network synchronization coefficient.
- **n\_critics** (*int*) – the number of Q functions for ensemble.
- **bootstrap** (*bool*) – flag to bootstrap Q functions.
- **share\_encoder** (*bool*) – flag to share encoder network.
- **update\_actor\_interval** (*int*) – interval to update policy function.
- **initial\_temperature** (*float*) – initial temperature value.
- **initial\_alpha** (*float*) – initial  $\alpha$  value.
- **alpha\_threshold** (*float*) – threshold value described as  $\tau$ .
- **n\_action\_samples** (*int*) – the number of sampled actions to compute  $\log \sum_a \exp Q(s, a)$ .
- **eps** (*float*) –  $\epsilon$  for Adam optimizer.
- **use\_batch\_norm** (*bool*) – flag to insert batch normalization layers.
- **q\_func\_type** (*str*) – type of Q function. Available options are ['mean', 'qr', 'iqn', 'fqf'].
- **n\_epochs** (*int*) – the number of epochs to train.
- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.
- **scaler** (*d3rlpy.preprocessingScaler* or *str*) – preprocessor. The available options are ['pixel', 'min\_max', 'standard']
- **augmentation** (*d3rlpy.augmentation.AugmentationPipeline* or *list(str)*) – augmentation pipeline.
- **n\_augmentations** (*int*) – the number of data augmentations to update.
- **encoder\_params** (*dict*) – optional arguments for encoder setup. If the observation is pixel, you can pass filters with list of tuples consisting with (filter\_size, kernel\_size, stride) and feature\_size with an integer scaler for the last linear layer size. If the observation is vector, you can pass hidden\_units with list of hidden unit sizes.
- **dynamics** (*d3rlpy.dynamics.base.DynamicsBase*) – dynamics model for data augmentation.
- **impl** (*d3rlpy.algos.torch.cql\_impl.CQLImpl*) – algorithm implementation.

**actor\_learning\_rate**

learning rate for policy function.

Type *float***critic\_learning\_rate**

learning rate for Q functions.

Type *float***temp\_learning\_rate**

learning rate for temperature parameter of SAC.

Type *float*

**alpha\_learning\_rate**

learning rate for  $\alpha$ .

Type float

**batch\_size**

mini-batch size.

Type int

**gamma**

discount factor.

Type float

**tau**

target network synchronization coefficient.

Type float

**n\_critics**

the number of Q functions for ensemble.

Type int

**bootstrap**

flag to bootstrap Q functions.

Type bool

**share\_encoder**

flag to share encoder network.

Type bool

**update\_actor\_interval**

interval to update policy function.

Type int

**initial\_temperature**

initial temperature value.

Type float

**initial\_alpha**

initial  $\alpha$  value.

Type float

**alpha\_threshold**

threshold value described as  $\tau$ .

Type float

**n\_action\_samples**

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

Type int

**eps**

$\epsilon$  for Adam optimizer.

Type float

**use\_batch\_norm**

flag to insert batch normalization layers.

**Type** `bool`

**q\_func\_type**  
type of Q function.

**Type** `str`

**n\_epochs**  
the number of epochs to train.

**Type** `int`

**use\_gpu**  
GPU device.

**Type** `d3rlpy.gpu.Device`

**scaler**  
preprocessor.

**Type** `d3rlpy.preprocessing.Scaler`

**augmentation**  
augmentation pipeline.

**Type** `d3rlpy.augmentation.AugmentationPipeline`

**n\_augmentations**  
the number of data augmentations to update.

**Type** `int`

**encoder\_params**  
optional arguments for encoder setup.

**Type** `dict`

**dynamics**  
dynamics model.

**Type** `d3rlpy.dynamics.base.DynamicsBase`

**impl**  
algorithm implementation.

**Type** `d3rlpy.algos.torch.cql_impl.CQLImpl`

## Methods

**create\_impl** (*observation\_shape*, *action\_size*)  
Instantiate implementation objects with the dataset shapes.  
This method will be used internally when *fit* method is called.

### Parameters

- **observation\_shape** (*tuple*) – observation shape.
- **action\_size** (*int*) – dimension of action-space.

**fit** (*episodes*, *experiment\_name=None*, *with\_timestamp=True*, *logdir='d3rlpy\_logs'*, *verbose=True*, *show\_progress=True*, *tensorboard=True*, *eval\_episodes=None*, *save\_interval=1*, *scorers=None*)  
Trains with the given dataset.

```
algo.fit(epochs)
```

### Parameters

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

**classmethod** `from_json` (*fname*, *use\_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

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

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

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

### Parameters

- **fname** (*str*) – file path to *params.json*.
- **use\_gpu** (*bool* or `d3rlpy.gpu.Device`) – flag to use GPU or device.

**Returns** algorithm.

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

**get\_params** (*deep=True*)

Returns the all attributes.

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

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

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

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

**Returns** attribute values in dictionary.

**Return type** *dict*

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

Load neural network parameters.

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

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

**predict** (*x*)

Returns greedy actions.

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

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

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

**Returns** greedy actions

**Return type** *numpy.ndarray*

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

Returns predicted action-values.

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

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

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

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

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

**Parameters**

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

**Returns** predicted action-values

**Return type** `numpy.ndarray`

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

Returns sampled actions.

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

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

**Returns** sampled actions.

**Return type** `numpy.ndarray`

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

Saves neural network parameters.

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

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

**save\_policy** (*fname*)

Save the greedy-policy computational graph as TorchScript.

```
algo.save_policy('policy.pt')
```

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

See also

- [https://pytorch.org/tutorials/beginner/Intro\\_to\\_TorchScript\\_tutorial.html](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](https://pytorch.org/tutorials/advanced/cpp_export.html) (for C++).

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

**set\_params** (*\*\*params*)

Sets the given arguments to the attributes if they exist.

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

```
algo.set_params(n_epochs=10, batch_size=100)
```

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

**Returns** itself.

**Return type** `d3rlpy.algos.base.AlgoBase`

**update** (*epoch*, *total\_step*, *batch*)

Update parameters with mini-batch of data.

**Parameters**

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

**Returns** loss values.

**Return type** list

## 2.1.2 Discrete control algorithms

<code>d3rlpy.algos.DiscreteBC</code>	Behavior Cloning algorithm for discrete control.
<code>d3rlpy.algos.DQN</code>	Deep Q-Network algorithm.
<code>d3rlpy.algos.DoubleDQN</code>	Double Deep Q-Network algorithm.
<code>d3rlpy.algos.DiscreteBCQ</code>	Discrete version of Batch-Constrained Q-learning algorithm.
<code>d3rlpy.algos.DiscreteCQL</code>	Discrete version of Conservative Q-Learning algorithm.

### d3rlpy.algos.DiscreteBC

```
class d3rlpy.algos.DiscreteBC(learning_rate=0.001, batch_size=100, eps=1e-08, beta=0.5,
                               use_batch_norm=True, n_epochs=1000, use_gpu=False,
                               scaler=None, augmentation=[], n_augmentations=1, en-
                               coder_params={}, dynamics=None, impl=None, **kwargs)
```

Behavior Cloning algorithm for discrete control.

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

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

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

#### Parameters

- **n\_epochs** (*int*) – the number of epochs to train.
- **batch\_size** (*int*) – mini-batch size.
- **learning\_rate** (*float*) – learning rate.
- **eps** (*float*) –  $\epsilon$  for Adam optimizer.
- **beta** (*float*) – regularization factor.
- **use\_batch\_norm** (*bool*) – flag to insert batch normalization layers.
- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.
- **scaler** (*d3rlpy.preprocessingScaler* or *str*) – preprocessor. The available options are [*'pixel'*, *'min\_max'*, *'standard'*]
- **augmentation** (*d3rlpy.augmentation.AugmentationPipeline* or *list(str)*) – augmentation pipeline.
- **n\_augmentations** (*int*) – the number of data augmentations to update.
- **encoder\_params** (*dict*) – optional arguments for encoder setup. If the observation is pixel, you can pass filters with list of tuples consisting with (*filter\_size*, *kernel\_size*, *stride*) and *feature\_size* with an integer scaler for the last linear layer size. If the observation is vector, you can pass *hidden\_units* with list of hidden unit sizes.

- **dynamics** (*d3rlpy.dynamics.base.DynamicsBase*) – dynamics model for data augmentation.
- **impl** (*d3rlpy.algos.torch.bc\_impl.DiscreteBCImpl*) – implementation of the algorithm.

**n\_epochs**

the number of epochs to train.

**Type** `int`

**batch\_size**

mini-batch size.

**Type** `int`

**learning\_rate**

learning rate.

**Type** `float`

**eps**

$\epsilon$  for Adam optimizer.

**Type** `float`

**beta**

regularization factor.

**Type** `float`

**use\_batch\_norm**

flag to insert batch normalization layers.

**Type** `bool`

**use\_gpu**

GPU device.

**Type** `d3rlpy.gpu.Device`

**scaler**

preprocessor.

**Type** `d3rlpy.preprocessing.Scaler`

**augmentation**

augmentation pipeline.

**Type** `d3rlpy.augmentation.AugmentationPipeline`

**n\_augmentations**

the number of data augmentations to update.

**Type** `int`

**encoder\_params**

optional arguments for encoder setup.

**Type** `dict`

**dynamics**

dynamics model.

**Type** `d3rlpy.dynamics.base.DynamicsBase`



**impl**

implemenation of the algorithm.

**Type** d3rlpy.algos.torch.bc\_impl.DiscreteBCImpl

## Methods

**create\_impl** (*observation\_shape*, *action\_size*)

Instantiate implementation objects with the dataset shapes.

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

### Parameters

- **observation\_shape** (*tuple*) – observation shape.
- **action\_size** (*int*) – dimension of action-space.

**fit** (*episodes*, *experiment\_name=None*, *with\_timestamp=True*, *logdir='d3rlpy\_logs'*, *verbose=True*, *show\_progress=True*, *tensorboard=True*, *eval\_episodes=None*, *save\_interval=1*, *scorers=None*)  
Trains with the given dataset.

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

### Parameters

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

**classmethod from\_json** (*fname*, *use\_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

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

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

(continues on next page)

(continued from previous page)

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

**Parameters**

- **fname** (*str*) – file path to *params.json*.
- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.

**Returns** algorithm.**Return type** *d3rlpy.base.LearnableBase***get\_params** (*deep=True*)

Returns the all attributes.

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

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

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

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

Load neural network parameters.

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

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

Returns greedy actions.

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

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

**Parameters** **x** (*numpy.ndarray*) – observations**Returns** greedy actions**Return type** *numpy.ndarray***predict\_value** (*x, action*)

value prediction is not supported by BC algorithms.

**sample\_action** (*x*)  
sampling action is not supported by BC algorithm.

**save\_model** (*fname*)  
Saves neural network parameters.

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

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

**save\_policy** (*fname*)  
Save the greedy-policy computational graph as TorchScript.

```
algo.save_policy('policy.pt')
```

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

See also

- [https://pytorch.org/tutorials/beginner/Intro\\_to\\_TorchScript\\_tutorial.html](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](https://pytorch.org/tutorials/advanced/cpp_export.html) (for C++).

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

**set\_params** (*\*\*params*)  
Sets the given arguments to the attributes if they exist.

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

```
algo.set_params(n_epochs=10, batch_size=100)
```

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

**Returns** itself.

**Return type** `d3rlpy.algos.base.AlgoBase`

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

**Parameters**

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

**Returns** loss values.

**Return type** `list`

## d3rlpy.algos.DQN

```
class d3rlpy.algos.DQN (learning_rate=6.25e-05, batch_size=32, gamma=0.99, n_critics=1,
                        bootstrap=False, share_encoder=False, eps=0.00015, target_update_interval=8000.0, use_batch_norm=True, q_func_type='mean',
                        n_epochs=1000, use_gpu=False, scaler=None, augmentation=[], n_augmentations=1, encoder_params={}, dynamics=None, impl=None,
                        **kwargs)
```

Deep Q-Network algorithm.

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

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

## References

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

### Parameters

- **learning\_rate** (*float*) – learning rate.
- **batch\_size** (*int*) – mini-batch size.
- **gamma** (*float*) – discount factor.
- **n\_critics** (*int*) – the number of Q functions for ensemble.
- **bootstrap** (*bool*) – flag to bootstrap Q functions.
- **share\_encoder** (*bool*) – flag to share encoder network.
- **eps** (*float*) –  $\epsilon$  for Adam optimizer.
- **target\_update\_interval** (*int*) – interval to update the target network.
- **use\_batch\_norm** (*bool*) – flag to insert batch normalization layers
- **q\_func\_type** (*str*) – type of Q function. Available options are ['mean', 'qr', 'iqn', 'fqf'].
- **n\_epochs** (*int*) – the number of epochs to train.
- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.
- **scaler** (*d3rlpy.preprocessingScaler* or *str*) – preprocessor. The available options are ['pixel', 'min\_max', 'standard']
- **augmentation** (*d3rlpy.augmentation.AugmentationPipeline* or *list(str)*) – augmentation pipeline.
- **n\_augmentations** (*int*) – the number of data augmentations to update.
- **encoder\_params** (*dict*) – optional arguments for encoder setup. If the observation is pixel, you can pass filters with list of tuples consisting with (*filter\_size*, *kernel\_size*, *stride*) and *feature\_size* with an integer scaler for the last linear layer size. If the observation is vector, you can pass *hidden\_units* with list of hidden unit sizes.
- **dynamics** (*d3rlpy.dynamics.base.DynamicsBase*) – dynamics model for data augmentation.

- **impl** (*d3rlpy.algos.torch.dqn\_impl.DQNImpl*) – algorithm implementation.

**learning\_rate**

learning rate.

**Type** `float`

**batch\_size**

mini-batch size.

**Type** `int`

**gamma**

discount factor.

**Type** `float`

**n\_critics**

the number of Q functions for ensemble.

**Type** `int`

**bootstrap**

flag to bootstrap Q functions.

**Type** `bool`

**share\_encoder**

flag to share encoder network.

**Type** `bool`

**eps**

$\epsilon$  for Adam optimizer.

**Type** `float`

**target\_update\_interval**

interval to update the target network.

**Type** `int`

**use\_batch\_norm**

flag to insert batch normalization layers

**Type** `bool`

**q\_func\_type**

type of Q function.

**Type** `str`

**n\_epochs**

the number of epochs to train.

**Type** `int`

**use\_gpu**

GPU device.

**Type** `d3rlpy.gpu.Device`

**scaler**

preprocessor.

**Type** `d3rlpy.preprocessing.Scaler`

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

**n\_augmentations**  
the number of data augmentations to update.  
**Type** `int`

**encoder\_params**  
optional arguments for encoder setup.  
**Type** `dict`

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

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

## Methods

**create\_impl** (*observation\_shape*, *action\_size*)  
Instantiate implementation objects with the dataset shapes.  
This method will be used internally when *fit* method is called.

### Parameters

- **observation\_shape** (*tuple*) – observation shape.
- **action\_size** (*int*) – dimension of action-space.

**fit** (*episodes*, *experiment\_name=None*, *with\_timestamp=True*, *logdir='d3rlpy\_logs'*, *verbose=True*, *show\_progress=True*, *tensorboard=True*, *eval\_episodes=None*, *save\_interval=1*, *scorers=None*)  
Trains with the given dataset.

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

### Parameters

- **episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to train.
- **experiment\_name** (*str*) – experiment name for logging. If not passed, the directory name will be `{class name}_{timestamp}`.
- **with\_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show\_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **eval\_episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.

- **scorers** (*list (callable)*) – list of scorer functions used with *eval\_episodes*.

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

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

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

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

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

#### Parameters

- **fname** (*str*) – file path to *params.json*.
- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.

**Returns** algorithm.

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

**get\_params** (*deep=True*)

Returns the all attributes.

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

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

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

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

**Returns** attribute values in dictionary.

**Return type** *dict*

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

Load neural network parameters.

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

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

**predict** (*x*)

Returns greedy actions.

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

(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** (*numpy.ndarray*) – observations

**Returns** greedy actions

**Return type** *numpy.ndarray*

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

Returns predicted action-values.

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

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

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

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

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

**Parameters**

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

**Returns** predicted action-values

**Return type** *numpy.ndarray*

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

Returns sampled actions.

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

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

**Returns** sampled actions.

**Return type** *numpy.ndarray*

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

Saves neural network parameters.

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



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

**save\_policy** (*fname*)

Save the greedy-policy computational graph as TorchScript.

```
algo.save_policy('policy.pt')
```

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

See also

- [https://pytorch.org/tutorials/beginner/Intro\\_to\\_TorchScript\\_tutorial.html](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](https://pytorch.org/tutorials/advanced/cpp_export.html) (for C++).

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

**set\_params** (*\*\*params*)

Sets the given arguments to the attributes if they exist.

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

```
algo.set_params(n_epochs=10, batch_size=100)
```

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

**Returns** itself.

**Return type** d3rlpy.algos.base.AlgoBase

**update** (*epoch*, *total\_step*, *batch*)

Update parameters with mini-batch of data.

**Parameters**

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

**Returns** loss values.

**Return type** list

## d3rlpy.algos.DoubleDQN

```
class d3rlpy.algos.DoubleDQN (learning_rate=6.25e-05, batch_size=32, gamma=0.99,
                             n_critics=1, bootstrap=False, share_encoder=False, eps=0.00015,
                             target_update_interval=8000.0, use_batch_norm=True,
                             q_func_type='mean', n_epochs=1000, use_gpu=False,
                             scaler=None, augmentation=[], n_augmentations=1, en-
                             coder_params={}, dynamics=None, impl=None, **kwargs)
```

Double Deep Q-Network algorithm.

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

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

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

## References

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

### Parameters

- **learning\_rate** (*float*) – learning rate.
- **batch\_size** (*int*) – mini-batch size.
- **gamma** (*float*) – discount factor.
- **n\_critics** (*int*) – the number of Q functions.
- **bootstrap** (*bool*) – flag to bootstrap Q functions.
- **share\_encoder** (*bool*) – flag to share encoder network.
- **eps** (*float*) –  $\epsilon$  for Adam optimizer.
- **target\_update\_interval** (*int*) – interval to synchronize the target network.
- **use\_batch\_norm** (*bool*) – flag to insert batch normalization layers
- **q\_func\_type** (*str*) – type of Q function. Available options are ['mean', 'qr', 'iqn', 'fqf'].
- **n\_epochs** (*int*) – the number of epochs to train.
- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.
- **scaler** (*d3rlpy.preprocessingScaler* or *str*) – preprocessor. The available options are ['pixel', 'min\_max', 'standard']
- **augmentation** (*d3rlpy.augmentation.AugmentationPipeline* or *list(str)*) – augmentation pipeline.
- **n\_augmentations** (*int*) – the number of data augmentations to update.
- **encoder\_params** (*dict*) – optional arguments for encoder setup. If the observation is pixel, you can pass *filters* with list of tuples consisting with (*filter\_size*, *kernel\_size*, *stride*) and *feature\_size* with an integer scaler for the last linear layer size. If the observation is vector, you can pass *hidden\_units* with list of hidden unit sizes.
- **dynamics** (*d3rlpy.dynamics.base.DynamicsBase*) – dynamics model for data augmentation.
- **impl** (*d3rlpy.algos.torch.dqn\_impl.DoubledDQNImpl*) – algorithm implementation.

**learning\_rate**

learning rate.

Type *float*

**batch\_size**  
mini-batch size.  
**Type** `int`

**gamma**  
discount factor.  
**Type** `float`

**n\_critics**  
the number of Q functions.  
**Type** `int`

**bootstrap**  
flag to bootstrap Q functions.  
**Type** `bool`

**share\_encoder**  
flag to share encoder network.  
**Type** `bool`

**eps**  
 $\epsilon$  for Adam optimizer.  
**Type** `float`

**target\_update\_interval**  
interval to synchronize the target network.  
**Type** `int`

**use\_batch\_norm**  
flag to insert batch normalization layers  
**Type** `bool`

**q\_func\_type**  
type of Q function.  
**Type** `str`

**n\_epochs**  
the number of epochs to train.  
**Type** `int`

**use\_gpu**  
GPU device.  
**Type** `d3rlpy.gpu.Device`

**scaler**  
preprocessor.  
**Type** `d3rlpy.preprocessing.Scaler`

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

**n\_augmentations**  
the number of data augmentations to update.

**Type** `int`

**encoder\_params**

optional arguments for encoder setup.

**Type** `dict`

**dynamics**

dynamics model.

**Type** `d3rlpy.dynaics.base.DynamicsBase`

**impl**

algorithm implementation.

**Type** `d3rlpy.algos.torch.dqn_impl.DoubleDQNImpl`

## Methods

**create\_impl** (*observation\_shape*, *action\_size*)

Instantiate implementation objects with the dataset shapes.

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

### Parameters

- **observation\_shape** (*tuple*) – observation shape.
- **action\_size** (*int*) – dimension of action-space.

**fit** (*episodes*, *experiment\_name=None*, *with\_timestamp=True*, *logdir='d3rlpy\_logs'*, *verbose=True*, *show\_progress=True*, *tensorboard=True*, *eval\_episodes=None*, *save\_interval=1*, *scorers=None*)  
Trains with the given dataset.

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

### Parameters

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

**classmethod from\_json** (*fname*, *use\_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```

from d3rlpy.algos import Algo

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

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

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

```

**Parameters**

- **fname** (*str*) – file path to *params.json*.
- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.

**Returns** algorithm.**Return type** *d3rlpy.base.LearnableBase***get\_params** (*deep=True*)

Returns the all attributes.

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

```

params = algo.get_params(deep=True)

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

```

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

Load neural network parameters.

```

algo.load_model('model.pt')

```

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

Returns greedy actions.

```

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

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

```

**Parameters** **x** (*numpy.ndarray*) – observations**Returns** greedy actions

**Return type** `numpy.ndarray`

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

Returns predicted action-values.

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

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

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

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

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

#### Parameters

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

**Returns** predicted action-values

**Return type** `numpy.ndarray`

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

Returns sampled actions.

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

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

**Returns** sampled actions.

**Return type** `numpy.ndarray`

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

Saves neural network parameters.

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

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

**save\_policy** (*fname*)

Save the greedy-policy computational graph as TorchScript.

```
algo.save_policy('policy.pt')
```

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

See also

- [https://pytorch.org/tutorials/beginner/Intro\\_to\\_TorchScript\\_tutorial.html](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](https://pytorch.org/tutorials/advanced/cpp_export.html) (for C++).

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

**set\_params** (*\*\*params*)

Sets the given arguments to the attributes if they exist.

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

```
algo.set_params(n_epochs=10, batch_size=100)
```

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

**Returns** itself.

**Return type** `d3rlpy.algos.base.AlgoBase`

**update** (*epoch*, *total\_step*, *batch*)

Update parameters with mini-batch of data.

**Parameters**

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

**Returns** loss values.

**Return type** *list*

## d3rlpy.algos.DiscreteBCQ

```
class d3rlpy.algos.DiscreteBCQ (learning_rate=6.25e-05, batch_size=32, gamma=0.99,
                                n_critics=1, bootstrap=False, share_encoder=False,
                                action_flexibility=0.3, beta=0.5, eps=0.00015, tar-
                                get_update_interval=8000.0, use_batch_norm=True,
                                q_func_type='mean', n_epochs=1000, use_gpu=False,
                                scaler=None, augmentation=[], n_augmentations=1, en-
                                coder_params={}, dynamics=None, impl=None, **kwargs)
```

Discrete version of Batch-Constrained Q-learning algorithm.

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

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

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

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

which eliminates actions with probabilities  $\tau$  times smaller than the maximum one.

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

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

## References

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

### Parameters

- **learning\_rate** (*float*) – learning rate.
- **batch\_size** (*int*) – mini-batch size.
- **gamma** (*float*) – discount factor.
- **n\_critics** (*int*) – the number of Q functions for ensemble.
- **bootstrap** (*bool*) – flag to bootstrap Q functions.
- **share\_encoder** (*bool*) – flag to share encoder network.
- **action\_flexibility** (*float*) – probability threshold represented as :math: 'au'.
- **beta** (*float*) – regularization term for imitation function.
- **eps** (*float*) –  $\epsilon$  for Adam optimizer.
- **target\_update\_interval** (*int*) – interval to update the target network.
- **use\_batch\_norm** (*bool*) – flag to insert batch normalization layers.
- **q\_func\_type** (*str*) – type of Q function. Available options are ['mean', 'qr', 'iqn', 'fqf'].
- **n\_epochs** (*int*) – the number of epochs to train.
- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.
- **scaler** (*d3rlpy.preprocessingScaler* or *str*) – preprocessor. The available options are ['pixel', 'min\_max', 'standard']
- **augmentation** (*d3rlpy.augmentation.AugmentationPipeline* or *list(str)*) – augmentation pipeline.
- **n\_augmentations** (*int*) – the number of data augmentations to update.
- **encoder\_params** (*dict*) – optional arguments for encoder setup. If the observation is pixel, you can pass filters with list of tuples consisting with (filter\_size, kernel\_size, stride) and feature\_size with an integer scaler for the last linear layer size. If the observation is vector, you can pass hidden\_units with list of hidden unit sizes.
- **dynamics** (*d3rlpy.dynamics.base.DynamicsBase*) – dynamics model for data augmentation.
- **impl** (*d3rlpy.algos.torch.bcq\_impl.DiscreteBCQImpl*) – algorithm implementation.



**learning\_rate**

learning rate.

Type float

**batch\_size**

mini-batch size.

Type int

**gamma**

discount factor.

Type float

**n\_critics**

the number of Q functions for ensemble.

Type int

**bootstrap**

flag to bootstrap Q functions.

Type bool

**share\_encoder**

flag to share encoder network.

Type bool

**action\_flexibility**

probability threshold represented as :math: 'au'.

Type float

**beta**

reguralization term for imitation function.

Type float

**eps**

$\epsilon$  for Adam optimizer.

Type float

**target\_update\_interval**

interval to update the target network.

Type int

**use\_batch\_norm**

flag to insert batch normalization layers.

Type bool

**q\_func\_type**

type of Q function.

Type str

**n\_epochs**

the number of epochs to train.

Type int

**use\_gpu**

GPU device.

**Type** d3rlpy.gpu.Device

**scaler**

preprocessor.

**Type** d3rlpy.preprocessing.Scaler

**augmentation**

augmentation pipeline.

**Type** d3rlpy.augmentation.AugmentationPipeline

**n\_augmentations**

the number of data augmentations to update.

**Type** int

**encoder\_params**

optional arguments for encoder setup.

**Type** dict

**dynamics**

dynamics model.

**Type** d3rlpy.dynamics.base.DynamicsBase

**impl**

algorithm implementation.

**Type** d3rlpy.algos.torch.bcq\_impl.DiscreteBCQImpl

## Methods

**create\_impl** (*observation\_shape*, *action\_size*)

Instantiate implementation objects with the dataset shapes.

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

### Parameters

- **observation\_shape** (*tuple*) – observation shape.
- **action\_size** (*int*) – dimension of action-space.

**fit** (*episodes*, *experiment\_name=None*, *with\_timestamp=True*, *logdir='d3rlpy\_logs'*, *verbose=True*, *show\_progress=True*, *tensorboard=True*, *eval\_episodes=None*, *save\_interval=1*, *scorers=None*)  
Trains with the given dataset.

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

### Parameters

- **episodes** (*list* (d3rlpy.dataset.Episode)) – list of episodes to train.
- **experiment\_name** (*str*) – experiment name for logging. If not passed, the directory name will be {class name}\_{timestamp}.
- **with\_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show\_progress** (*bool*) – flag to show progress bar for iterations.

- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **eval\_episodes** (*list* (*d3rlpy.dataset.Episode*)) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*list* (*callable*)) – list of scorer functions used with *eval\_episodes*.

**classmethod** **from\_json** (*fname*, *use\_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

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

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

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

#### Parameters

- **fname** (*str*) – file path to *params.json*.
- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.

**Returns** algorithm.

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

**get\_params** (*deep=True*)

Returns the all attributes.

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

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

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

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

**Returns** attribute values in dictionary.

**Return type** *dict*

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

Load neural network parameters.

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

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

**predict** (*x*)

Returns greedy actions.

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

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

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

**Returns** greedy actions

**Return type** *numpy.ndarray*

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

Returns predicted action-values.

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

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

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

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

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

**Parameters**

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

**Returns** predicted action-values

**Return type** *numpy.ndarray*

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

Returns sampled actions.

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

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

**Returns** sampled actions.

**Return type** *numpy.ndarray*

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

Saves neural network parameters.

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

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

**save\_policy** (*fname*)

Save the greedy-policy computational graph as TorchScript.

```
algo.save_policy('policy.pt')
```

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

See also

- [https://pytorch.org/tutorials/beginner/Intro\\_to\\_TorchScript\\_tutorial.html](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](https://pytorch.org/tutorials/advanced/cpp_export.html) (for C++).

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

**set\_params** (*\*\*params*)

Sets the given arguments to the attributes if they exist.

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

```
algo.set_params(n_epochs=10, batch_size=100)
```

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

**Returns** itself.

**Return type** `d3rlpy.algos.base.AlgoBase`

**update** (*epoch*, *total\_step*, *batch*)

Update parameters with mini-batch of data.

**Parameters**

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

**Returns** loss values.

**Return type** `list`

## d3rlpy.algos.DiscreteCQL

```
class d3rlpy.algos.DiscreteCQL (learning_rate=6.25e-05,    batch_size=32,    gamma=0.99,
                                n_critics=1,    bootstrap=False,    share_encoder=False,
                                eps=0.00015,    target_update_interval=8000.0,
                                use_batch_norm=True, q_func_type='mean', n_epochs=1000,
                                use_gpu=False,    scaler=None,    augmentation=[],
                                n_augmentations=1, encoder_params={}, dynamics=None,
                                impl=None, **kwargs)
```

Discrete version of Conservative Q-Learning algorithm.

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

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

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

## References

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

### Parameters

- **learning\_rate** (*float*) – learning rate.
- **batch\_size** (*int*) – mini-batch size.
- **gamma** (*float*) – discount factor.
- **n\_critics** (*int*) – the number of Q functions for ensemble.
- **bootstrap** (*bool*) – flag to bootstrap Q functions.
- **eps** (*float*) –  $\epsilon$  for Adam optimizer.
- **target\_update\_interval** (*int*) – interval to synchronize the target network.
- **use\_batch\_norm** (*bool*) – flag to insert batch normalization layers
- **q\_func\_type** (*str*) – type of Q function. Available options are ['mean', 'qr', 'iqn', 'fqf'].
- **n\_epochs** (*int*) – the number of epochs to train.
- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.
- **scaler** (*d3rlpy.preprocessingScaler* or *str*) – preprocessor. The available options are ['pixel', 'min\_max', 'standard']
- **augmentation** (*d3rlpy.augmentation.AugmentationPipeline* or *list(str)*) – augmentation pipeline.
- **n\_augmentations** (*int*) – the number of data augmentations to update.
- **encoder\_params** (*dict*) – optional arguments for encoder setup. If the observation is pixel, you can pass filters with list of tuples consisting with (filter\_size, kernel\_size, stride) and feature\_size with an integer scaler for the last linear layer size. If the observation is vector, you can pass hidden\_units with list of hidden unit sizes.

- **dynamics** (*d3rlpy.dynamics.base.DynamicsBase*) – dynamics model for data augmentation.
- **impl** (*d3rlpy.algos.torch.cql\_impl.DiscreteCQLImpl*) – algorithm implementation.

**learning\_rate**

learning rate.

**Type** *float*

**batch\_size**

mini-batch size.

**Type** *int*

**gamma**

discount factor.

**Type** *float*

**n\_critics**

the number of Q functions for ensemble.

**Type** *int*

**bootstrap**

flag to bootstrap Q functions.

**Type** *bool*

**eps**

$\epsilon$  for Adam optimizer.

**Type** *float*

**target\_update\_interval**

interval to synchronize the target network.

**Type** *int*

**use\_batch\_norm**

flag to insert batch normalization layers

**Type** *bool*

**q\_func\_type**

type of Q function.

**Type** *str*

**n\_epochs**

the number of epochs to train.

**Type** *int*

**use\_gpu**

GPU device.

**Type** *d3rlpy.gpu.Device*

**scaler**

preprocessor.

**Type** *d3rlpy.preprocessingScaler*

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

**n\_augmentations**  
the number of data augmentations to update.  
**Type** `int`

**encoder\_params**  
optional arguments for encoder setup.  
**Type** `dict`

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

**impl**  
algorithm implementation.  
**Type** `d3rlpy.algos.torch.CQLImpl.DiscreteCQLImpl`

## Methods

**create\_impl** (*observation\_shape*, *action\_size*)  
Instantiate implementation objects with the dataset shapes.  
This method will be used internally when *fit* method is called.

### Parameters

- **observation\_shape** (*tuple*) – observation shape.
- **action\_size** (*int*) – dimension of action-space.

**fit** (*episodes*, *experiment\_name=None*, *with\_timestamp=True*, *logdir='d3rlpy\_logs'*, *verbose=True*, *show\_progress=True*, *tensorboard=True*, *eval\_episodes=None*, *save\_interval=1*, *scorers=None*)  
Trains with the given dataset.

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

### Parameters

- **episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to train.
- **experiment\_name** (*str*) – experiment name for logging. If not passed, the directory name will be `{class name}_{timestamp}`.
- **with\_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show\_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **eval\_episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.



- **scorers** (*list (callable)*) – list of scorer functions used with *eval\_episodes*.

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

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

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

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

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

#### Parameters

- **fname** (*str*) – file path to *params.json*.
- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.

**Returns** algorithm.

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

**get\_params** (*deep=True*)

Returns the all attributes.

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

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

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

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

**Returns** attribute values in dictionary.

**Return type** *dict*

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

Load neural network parameters.

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

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

**predict** (*x*)

Returns greedy actions.

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

(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** (*numpy.ndarray*) – observations

**Returns** greedy actions

**Return type** *numpy.ndarray*

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

Returns predicted action-values.

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

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

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

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

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

**Parameters**

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

**Returns** predicted action-values

**Return type** *numpy.ndarray*

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

Returns sampled actions.

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

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

**Returns** sampled actions.

**Return type** *numpy.ndarray*

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

Saves neural network parameters.

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

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

**save\_policy** (*fname*)

Save the greedy-policy computational graph as TorchScript.

```
algo.save_policy('policy.pt')
```

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

See also

- [https://pytorch.org/tutorials/beginner/Intro\\_to\\_TorchScript\\_tutorial.html](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](https://pytorch.org/tutorials/advanced/cpp_export.html) (for C++).

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

**set\_params** (*\*\*params*)

Sets the given arguments to the attributes if they exist.

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

```
algo.set_params(n_epochs=10, batch_size=100)
```

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

**Returns** itself.

**Return type** d3rlpy.algos.base.AlgoBase

**update** (*epoch, total\_step, batch*)

Update parameters with mini-batch of data.

**Parameters**

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

**Returns** loss values.

**Return type** list

## 2.2 Q Functions

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

```
from d3rlpy.algos import CQL
cql = CQL(q_func_type='qr') # use Quantile Regression Q function
```

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

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

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

Table 3: available Q functions

q_func_type	reference
mean (default)	N/A
qr	<a href="#">Quantile Regression</a>
iqn	<a href="#">Implicit Quantile Network</a>
fqn (experimental)	<a href="#">Fully-parametrized Quantile Function</a>

## 2.3 MDPDataset

d3rlpy provides useful dataset structure for data-driven deep reinforcement learning. In supervised learning, the training script iterates input data  $X$  and label data  $Y$ . However, in reinforcement learning, mini-batches consist with sets of  $(s_t, a_t, r_{t+1}, s_{t+1})$  and episode terminal flags. Converting a set of observations, actions, rewards and terminal flags into this tuples is boring and requires some codings.

Therefore, d3rlpy provides *MDPDataSet* class which enables you to handle reinforcement learning datasets without any efforts.

```
from d3rlpy.dataset import MDPDataSet

# 1000 steps of observations with shape of (100,)
observations = np.random.random((1000, 100))
# 1000 steps of actions with shape of (4,)
actions = np.random.random((1000, 4))
# 1000 steps of rewards
rewards = np.random.random(1000)
# 1000 steps of terminal flags
terminals = np.random.randint(2, size=1000)

dataset = MDPDataSet(observations, actions, rewards, terminals)

# automatically splitted into d3rlpy.dataset.Episode objects
dataset.episodes

# each episode is also splitted into d3rlpy.dataset.Transition objects
dataset.episodes[0].observation
dataset.episodes[0].action
dataset.episodes[0].next_reward
dataset.episodes[0].next_observation
dataset.episodes[0].terminal

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

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

<code>d3rlpy.dataset.MDPDataset</code>	Markov-Decision Process Dataset class.
<code>d3rlpy.dataset.Episode</code>	Episode class.
<code>d3rlpy.dataset.Transition</code>	Transition class.
<code>d3rlpy.dataset.TransitionMiniBatch</code>	mini-batch of Transition objects.

### 2.3.1 d3rlpy.dataset.MDPDataset

**class** `d3rlpy.dataset.MDPDataset` (*observations*, *actions*, *rewards*, *terminals*, *discrete\_action=False*)

Markov-Decision Process Dataset class.

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

```
from d3rlpy.dataset import MDPDataset

# 1000 steps of observations with shape of (100,)
observations = np.random.random((1000, 100))
# 1000 steps of actions with shape of (4,)
actions = np.random.random((1000, 4))
# 1000 steps of rewards
rewards = np.random.random(1000)
# 1000 steps of terminal flags
terminals = np.random.randint(2, size=1000)

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

The MDPDataset object automatically splits the given data into list of `d3rlpy.dataset.Episode` objects. Furthermore, the MDPDataset object behaves like a list in order to use with scikit-learn utilities.

```
# returns the number of episodes
len(dataset)

# access to the first episode
episode = dataset[0]

# iterate through all episodes
for episode in dataset:
    pass
```

#### Parameters

- **observations** (`numpy.ndarray` or `list(numpy.ndarray)`) – N-D array. If the observation is a vector, the shape should be  $(N, \text{dim\_observation})$ . If the observations is an image, the shape should be  $(N, C, H, W)$ .
- **actions** (`numpy.ndarray`) – N-D array. If the actions-space is continuous, the shape should be  $(N, \text{dim\_action})$ . If the action-space is discrete, the shape should be  $(N,)$ .
- **rewards** (`numpy.ndarray`) – array of scalar rewards.
- **terminals** (`numpy.ndarray`) – array of binary terminal flags.
- **discrete\_action** (`bool`) – flag to use the given actions as discrete action-space actions.

## Methods

`__getitem__(index)`

`__len__()`

`__iter__()`

`append(observations, actions, rewards, terminals)`

Appends new data.

### Parameters

- **observations** (*numpy.ndarray* or *list(numpy.ndarray)*) – N-D array.
- **actions** (*numpy.ndarray*) – actions.
- **rewards** (*numpy.ndarray*) – rewards.
- **terminals** (*numpy.ndarray*) – terminals.

`clip_reward(low=None, high=None)`

Clips rewards in the given range.

### Parameters

- **low** (*float*) – minimum value. If None, clipping is not performed on lower edge.
- **high** (*float*) – maximum value. If None, clipping is not performed on upper edge.

`compute_stats()`

Computes statistics of the dataset.

```
stats = dataset.compute_stats()

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

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

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

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

**Returns** statistics of the dataset.

**Return type** dict

**dump** (*fname*)

Saves dataset as HDF5.

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

**get\_action\_size** ()

Returns dimension of action-space.

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

**Returns** dimension of action-space.

**Return type** *int*

**get\_observation\_shape** ()

Returns observation shape.

**Returns** observation shape.

**Return type** *tuple*

**is\_action\_discrete** ()

Returns *discrete\_action* flag.

**Returns** *discrete\_action* flag.

**Return type** *bool*

**classmethod load** (*fname*)

Loads dataset from HDF5.

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

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

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

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

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

**size** ()

Returns the number of episodes in the dataset.

**Returns** the number of episodes.

**Return type** *int*

## Attributes

**actions**

Returns the actions.

**Returns** array of actions.

**Return type** *numpy.ndarray*

**episodes**

Returns the episodes.

**Returns** list of `d3rlpy.dataset.Episode` objects.

**Return type** `list(d3rlpy.dataset.Episode)`

**observations**

Returns the observations.

**Returns** array of observations.

**Return type** (`numpy.ndarray` or `list(numpy.ndarray)`)

**rewards**

Returns the rewards.

**Returns** array of rewards

**Return type** `numpy.ndarray`

**terminals**

Returns the terminal flags.

**Returns** array of terminal flags.

**Return type** `numpy.ndarray`

### 2.3.2 d3rlpy.dataset.Episode

**class** `d3rlpy.dataset.Episode` (*observation\_shape, action\_size, observations, actions, rewards*)

Episode class.

This class is designed to hold data collected in a single episode.

Episode object automatically splits data into list of `d3rlpy.dataset.Transition` objects. Also Episode object behaves like a list object for ease of access to transitions.

```
# return the number of transitions
len(episode)

# access to the first transition
transitions = episode[0]

# iterate through all transitions
for transition in episode:
    pass
```

**Parameters**

- **observation\_shape** (*tuple*) – observation shape.
- **action\_size** (*int*) – dimension of action-space.
- **observations** (*numpy.ndarray* or *list(numpy.ndarray)*) – observations.
- **actions** (*numpy.ndarray*) – actions.
- **rewards** (*numpy.ndarray*) – scalar rewards.
- **terminals** (*numpy.ndarray*) – binary terminal flags.



## Methods

`__getitem__ (index)`

`__len__ ()`

`__iter__ ()`

`compute_return ()`

Computes sum of rewards.

$$R = \sum_{i=1} r_i$$

**Returns** episode return.

**Return type** `float`

`get_action_size ()`

Returns dimension of action-space.

**Returns** dimension of action-space.

**Return type** `int`

`get_observation_shape ()`

Returns observation shape.

**Returns** observation shape.

**Return type** `tuple`

`size ()`

Returns the number of transitions.

**Returns** the number of transitions.

**Return type** `int`

## Attributes

**actions**

Returns the actions.

**Returns** array of actions.

**Return type** `numpy.ndarray`

**observations**

Returns the observations.

**Returns** array of observations.

**Return type** (`numpy.ndarray` or `list(numpy.ndarray)`)

**rewards**

Returns the rewards.

**Returns** array of rewards.

**Return type** `numpy.ndarray`

**transitions**

Returns the transitions.

**Returns** list of `d3rlpy.dataset.Transition` objects.

**Return type** `list(d3rlpy.dataset.Transition)`

### 2.3.3 d3rlpy.dataset.Transition

**class** `d3rlpy.dataset.Transition`(*observation\_shape, action\_size, observation, action, reward, next\_observation, next\_action, next\_reward, terminal*)

Transition class.

This class is designed to hold data between two time steps, which is usually used as inputs of loss calculation in reinforcement learning.

#### Parameters

- **observation\_shape** (*tuple*) – observation shape.
- **action\_size** (*int*) – dimension of action-space.
- **observation** (*numpy.ndarray*) – observation at *t*.
- **action** (*numpy.ndarray* or *int*) – action at *t*.
- **reward** (*float*) – reward at *t*.
- **next\_observation** (*numpy.ndarray*) – observation at *t+1*.
- **next\_action** (*numpy.ndarray* or *int*) – action at *t+1*.
- **next\_reward** (*float*) – reward at *t+1*.
- **terminal** (*int*) – terminal flag at *t+1*.

#### Methods

**get\_action\_size**()

Returns dimension of action-space.

**Returns** dimension of action-space.

**Return type** `int`

**get\_observation\_shape**()

Returns observation shape.

**Returns** observation shape.

**Return type** `tuple`

#### Attributes

**action**

Returns action at *t*.

**Returns** action at *t*.

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

**next\_action**

Returns action at *t+1*.

**Returns** action at *t+1*.

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

**next\_observation**  
Returns observation at  $t+1$ .

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

**Return type** `numpy.ndarray`

**next\_reward**  
Returns reward at  $t+1$ .

**Returns** reward at  $t+1$ .

**Return type** `float`

**observation**  
Returns observation at  $t$ .

**Returns** observation at  $t$ .

**Return type** `numpy.ndarray`

**reward**  
Returns reward at  $t$ .

**Returns** reward at  $t$ .

**Return type** `float`

**terminal**  
Returns terminal flag at  $t+1$ .

**Returns** terminal flag at  $t+1$ .

**Return type** `int`

### 2.3.4 d3rlpy.dataset.TransitionMiniBatch

**class** `d3rlpy.dataset.TransitionMiniBatch` (*transitions*)  
mini-batch of Transition objects.

This class is designed to hold `d3rlpy.dataset.Transition` objects for being passed to algorithms during fitting.

**Parameters** `transitions` (*list* (`d3rlpy.dataset.Transition`)) – mini-batch of transitions.

#### Methods

`__getitem__` (*index*)

`__len__` ()

`__iter__` ()

`size` ()  
Returns size of mini-batch.

**Returns** mini-batch size.

**Return type** `int`

### Attributes

**actions**

Returns mini-batch of actions at  $t$ .

**Returns** actions at  $t$ .

**Return type** `numpy.ndarray`

**next\_actions**

Returns mini-batch of actions at  $t+1$ .

**Returns** actions at  $t+1$ .

**Return type** `numpy.ndarray`

**next\_observations**

Returns mini-batch of observations at  $t+1$ .

**Returns** observations at  $t+1$ .

**Return type** `numpy.ndarray`

**next\_rewards**

Returns mini-batch of rewards at  $t+1$ .

**Returns** rewards at  $t+1$ .

**Return type** `numpy.ndarray`

**observations**

Returns mini-batch of observations at  $t$ .

**Returns** observations at  $t$ .

**Return type** `numpy.ndarray`

**rewards**

Returns mini-batch of rewards at  $t$ .

**Returns** rewards at  $t$ .

**Return type** `numpy.ndarray`

**terminals**

Returns mini-batch of terminal flags at  $t+1$ .

**Returns** terminal flags at  $t+1$ .

**Return type** `numpy.ndarray`

**transitions**

Returns transitions.

**Returns** list of transitions.

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

## 2.4 Datasets

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

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

### 2.4.1 d3rlpy.datasets.get\_cartpole

`d3rlpy.datasets.get_cartpole()`

Returns cartpole dataset and environment.

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

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

**Return type** tuple

### 2.4.2 d3rlpy.datasets.get\_pendulum

`d3rlpy.datasets.get_pendulum()`

Returns pendulum dataset and environment.

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

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

**Return type** tuple

### 2.4.3 d3rlpy.datasets.get\_pybullet

`d3rlpy.datasets.get_pybullet(env_name)`

Returns pybullet dataset and environment.

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

```
from d3rlpy.datasets import get_pybullet

dataset, env = get_pybullet('hopper-bullet-mixed-v0')
```

### References

- <https://github.com/takuseno/d4rl-pybullet>

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

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

**Return type** tuple

### 2.4.4 d3rlpy.datasets.get\_atari

`d3rlpy.datasets.get_atari(env_name)`

Returns atari dataset and environment.

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

```
from d3rlpy.datasets import get_atari

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

#### References

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

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

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

**Return type** tuple

## 2.5 Preprocessing

d3rlpy provides several preprocessors tightly incorporated with algorithms. Each preprocessor is implemented with PyTorch operation, which will be included in the model exported by `save_policy` method.

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

dataset = MDPDataset(...)

# choose from ['pixel', 'min_max', 'standard'] or None
cql = CQL(scaler='standard')

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

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

# you don't need to take care of preprocessing at production
policy = torch.jit.load('policy.pt')
action = policy(unprocessed_x)
```

You can also initialize scalers by yourself.

```
from d3rlpy.preprocessing import StandardScaler

scaler = StandardScaler(mean=..., std=...)

cql = CQL(scaler=scaler)
```

---

`d3rlpy.preprocessing.PixelScaler`

Pixel normalization preprocessing.

Continued on next page

Table 6 – continued from previous page

<code>d3rlpy.preprocessing.MinMaxScaler</code>	Min-Max normalization preprocessing.
<code>d3rlpy.preprocessing.StandardScaler</code>	Standardization preprocessing.

### 2.5.1 d3rlpy.preprocessing.PixelScaler

**class** d3rlpy.preprocessing.PixelScaler

Pixel normalization preprocessing.

$$x' = x/255$$

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

dataset = MDPDataset(observations, actions, rewards, terminals)

# initialize algorithm with PixelScaler
cql = CQL(scaler='pixel')

cql.fit(dataset.episodes)
```

#### Methods

**fit** (*episodes*)

**get\_params** ()

Returns scaling parameters.

PixelScaler returns empty dictionary.

**Returns** empty dictionary.

**Return type** dict

**get\_type** ()

Returns scaler type.

**Returns** pixel.

**Return type** str

**reverse\_transform** (*x*)

Returns reversely transformed observations.

**Parameters** *x* (*torch.Tensor*) – normalized observation tensor.

**Returns** unnormalized pixel observation tensor.

**Return type** torch.Tensor

**transform** (*x*)

Returns normalized pixel observations.

**Parameters** *x* (*torch.Tensor*) – pixel observation tensor.

**Returns** normalized pixel observation tensor.

**Return type** torch.Tensor

## 2.5.2 d3rlpy.preprocessing.MinMaxScaler

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

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

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

dataset = MDPDataset(observations, actions, rewards, terminals)

# initialize algorithm with MinMaxScaler
cql = CQL(scaler='min_max')

# scaler is initialized from the given episodes
cql.fit(dataset.episodes)
```

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

```
from d3rlpy.preprocessing import MinMaxScaler

# initialize with dataset
scaler = MinMaxScaler(dataset)

# initialize manually
minimum = observations.min(axis=0)
maximum = observations.max(axis=0)
scaler = MinMaxScaler(minimum=minimum, maximum=maximum)

cql = CQL(scaler=scaler)
```

### Parameters

- **dataset** (`d3rlpy.dataset.MDPDataset`) – dataset object.
- **min** (`numpy.ndarray`) – minimum values at each entry.
- **max** (`numpy.ndarray`) – maximum values at each entry.

### minimum

minimum values at each entry.

**Type** `numpy.ndarray`

### maximum

maximum values at each entry.

**Type** `numpy.ndarray`

### Methods

#### **fit** (episodes)

Fits minimum and maximum from list of episodes.

**Parameters** **episodes** (`list` (`d3rlpy.dataset.Episode`)) – list of episodes.



**get\_params()**

Returns scaling parameters.

**Returns** *maximum* and *minimum*.**Return type** `dict`**get\_type()**

Returns scaler type.

**Returns** *min\_max*.**Return type** `str`**reverse\_transform(x)**

Returns reversely transformed observations.

**Parameters** **x** (`torch.Tensor`) – normalized observation tensor.**Returns** unnormalized observation tensor.**Return type** `torch.Tensor`**transform(x)**

Returns normalized observation tensor.

**Parameters** **x** (`torch.Tensor`) – observation tensor.**Returns** normalized observation tensor.**Return type** `torch.Tensor`

### 2.5.3 d3rlpy.preprocessing.StandardScaler

**class** `d3rlpy.preprocessing.StandardScaler` (*dataset=None, mean=None, std=None*)  
Standardization preprocessing.

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

```

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

dataset = MDPDataset(observations, actions, rewards, terminals)

# initialize algorithm with StandardScaler
cql = CQL(scaler='standard')

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

```

You can initialize with `d3rlpy.dataset.MDPDataset` object or manually.

```

from d3rlpy.preprocessing import StandardScaler

# initialize with dataset
scaler = StandardScaler(dataset)

# initialize manually
mean = observations.mean(axis=0)
std = observations.std(axis=0)

```

(continues on next page)

(continued from previous page)

```

scaler = StandardScaler(mean=mean, std=std)

cql = CQL(scaler=scaler)

```

**Parameters**

- **dataset** (`d3rlpy.dataset.MDPDataset`) – dataset object.
- **mean** (`numpy.ndarray`) – mean values at each entry.
- **std** (`numpy.ndarray`) – standard deviation at each entry.

**mean**

mean values at each entry.

**Type** `numpy.ndarray`

**std**

standard deviation values at each entry.

**Type** `numpy.ndarray`

**Methods****fit** (*episodes*)

Fits mean and standard deviation from list of episodes.

**Parameters** **episodes** (`list(d3rlpy.dataset.Episode)`) – list of episodes.

**get\_params** ()

Returns scaling parameters.

**Returns** *mean* and *std*.

**Return type** `dict`

**get\_type** ()

Returns scaler type.

**Returns** *standard*.

**Return type** `str`

**reverse\_transform** (*x*)

Returns reversely transformed observation tensor.

**Parameters** **x** (`torch.Tensor`) – standardized observation tensor.

**Returns** unstandardized observation tensor.

**Return type** `torch.Tensor`

**transform** (*x*)

Returns standardized observation tensor.

**Parameters** **x** (`torch.Tensor`) – observation tensor.

**Returns** standardized observation tensor.

**Return type** `torch.Tensor`

## 2.6 Data Augmentation

d3rlpy provides data augmentation techniques tightly integrated with reinforcement learning algorithms.

Kostrikov et al., Image Augmentation Is All You Need: Regularizing Deep Reinforcement Learning from Pixels.

Efficient data augmentation potentially boosts algorithm performance significantly.

```
from d3rlpy.algos import DiscreteCQL

# choose data augmentation types
cql = DiscreteCQL(augmentation=['random_shift', 'intensity'],
                  n_augmentations=2)
```

You can also tune data augmentation parameters by yourself.

```
from d3rlpy.augmentation.image import RandomShift

random_shift = RandomShift(shift_size=10)

cql = DiscreteCQL(augmentation=[random_shift, 'intensity'],
                  n_augmentations=2)
```

### 2.6.1 Image Observation

<code>d3rlpy.augmentation.image.RandomShift</code>	Random shift augmentation.
<code>d3rlpy.augmentation.image.Cutout</code>	Cutout augmentation.
<code>d3rlpy.augmentation.image.HorizontalFlip</code>	Horizontal flip augmentation.
<code>d3rlpy.augmentation.image.VerticalFlip</code>	Vertical flip augmentation.
<code>d3rlpy.augmentation.image.RandomRotation</code>	Random rotation augmentation.
<code>d3rlpy.augmentation.image.Intensity</code>	Intensity augmentation.

#### d3rlpy.augmentation.image.RandomShift

**class** `d3rlpy.augmentation.image.RandomShift` (*shift\_size=4*)  
Random shift augmentation.

**Parameters** `shift_size` (*int*) – size to shift image.

**shift\_size**  
size to shift image.

**Type** `int`

#### Methods

**get\_params** ()  
Returns augmentation parameters.

**Returns** augmentation parameters.

**Return type** `dict`

**get\_type()**

Returns augmentation type.

**Returns** *random\_shift*.

**Return type** *str*

**transform**(*x*)

Returns shifted images.

**Parameters** *x* (*torch.Tensor*) – observation tensor.

**Returns** processed observation tensor.

**Return type** *torch.Tensor*

### **d3rlpy.augmentation.image.Cutout**

**class** *d3rlpy.augmentation.image.Cutout* (*probability=0.5*)

Cutout augmentation.

**Parameters** *probability* (*float*) – probability to cutout.

**probability**

probability to cutout.

**Type** *float*

#### **Methods**

**get\_params()**

Returns augmentation parameters.

**Returns** augmentation parameters.

**Return type** *dict*

**get\_type()**

Returns augmentation type.

**Returns** *cutout*.

**Return type** *str*

**transform**(*x*)

Returns observation performed Cutout.

**Parameters** *x* (*torch.Tensor*) – observation tensor.

**Returns** processed observation tensor.

**Return type** *torch.Tensor*

### **d3rlpy.augmentation.image.HorizontalFlip**

**class** *d3rlpy.augmentation.image.HorizontalFlip* (*probability=0.1*)

Horizontal flip augmentation.

**Parameters** *probability* (*float*) – probability to flip horizontally.

**probability**

probability to flip horizontally.

Type `float`

## Methods

**get\_params** ()

Returns augmentation parameters.

**Returns** augmentation parameters.

**Return type** `dict`

**get\_type** ()

Returns augmentation type.

**Returns** *horizontal\_flip*.

**Return type** `str`

**transform** (*x*)

Returns horizontally flipped image.

**Parameters** *x* (*torch.Tensor*) – observation tensor.

**Returns** processed observation tensor.

**Return type** `torch.Tensor`

## d3rlpy.augmentation.image.VerticalFlip

**class** `d3rlpy.augmentation.image.VerticalFlip` (*probability=0.1*)  
Vertical flip augmentation.

**Parameters** *probability* (*float*) – probability to flip vertically.

**probability**

probability to flip vertically.

**Type** `float`

## Methods

**get\_params** ()

Returns augmentation parameters.

**Returns** augmentation parameters.

**Return type** `dict`

**get\_type** ()

Returns augmentation type.

**Returns** *vertical\_flip*.

**Return type** `str`

**transform** (*x*)

Returns vertically flipped image.

**Parameters** *x* (*torch.Tensor*) – observation tensor.

**Returns** processed observation tensor.

**Return type** torch.Tensor

### d3rlpy.augmentation.image.RandomRotation

**class** d3rlpy.augmentation.image.**RandomRotation** (*degree=5.0*)  
Random rotation augmentation.

**Parameters** **degree** (*float*) – range of degrees to rotate image.

**degree**  
range of degrees to rotate image.

**Type** float

#### Methods

**get\_params** ()  
Returns augmentation parameters.  
**Returns** augmentation parameters.  
**Return type** dict

**get\_type** ()  
Returns augmentation type.  
**Returns** *random\_rotation*.  
**Return type** str

**transform** (*x*)  
Returns rotated image.  
**Parameters** **x** (*torch.Tensor*) – observation tensor.  
**Returns** processed observation tensor.  
**Return type** torch.Tensor

### d3rlpy.augmentation.image.Intensity

**class** d3rlpy.augmentation.image.**Intensity** (*scale=0.1*)  
Intensity augmentation.

$$x' = x + n$$

where  $n \sim N(0, scale)$ .

**Parameters** **scale** (*float*) – scale of multiplier.

**scale**  
scale of multiplier.

**Type** float

## Methods

**get\_params()**

Returns augmentation parameters.

**Returns** augmentation parameters.

**Return type** dict

**get\_type()**

Returns augmentation type.

**Returns** *intensity*.

**Return type** str

**transform(x)**

Returns multiplied image.

**Parameters** **x** (*torch.Tensor*) – observation tensor.

**Returns** processed observation tensor.

**Return type** torch.Tensor

## 2.7 Metrics

d3rlpy provides scoring functions without compromising scikit-learn compatibility. You can evaluate many metrics with test episodes during training.

```
from d3rlpy.datasets import get_cartpole
from d3rlpy.algos import DQN
from d3rlpy.metrics.scorer import td_error_scorer
from d3rlpy.metrics.scorer import average_value_estimation_scorer
from d3rlpy.metrics.scorer import evaluate_on_environment
from sklearn.model_selection import train_test_split

dataset, env = get_cartpole()

train_episodes, test_episodes = train_test_split(dataset)

dqn = DQN()

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

You can also use them with scikit-learn utilities.

```
from sklearn.model_selection import cross_validate

scores = cross_validate(dqn,
                        dataset,
                        scoring={
```

(continues on next page)

(continued from previous page)

```

        'td_error': td_error_scorer,
        'environment': evaluate_on_environment(env)
    })

```

## 2.7.1 Algorithms

<code>d3rlpy.metrics.scorer. td_error_scorer</code>	Returns average TD error (in negative scale).
<code>d3rlpy.metrics.scorer. discounted_sum_of_advantage_scorer</code>	Returns average of discounted sum of advantage (in negative scale).
<code>d3rlpy.metrics.scorer. average_value_estimation_scorer</code>	Returns average value estimation (in negative scale).
<code>d3rlpy.metrics.scorer. value_estimation_std_scorer</code>	Returns standard deviation of value estimation (in negative scale).
<code>d3rlpy.metrics.scorer. continuous_action_diff_scorer</code>	Returns squared difference of actions between algorithm and dataset.
<code>d3rlpy.metrics.scorer. discrete_action_match_scorer</code>	Returns percentage of identical actions between algorithm and dataset.
<code>d3rlpy.metrics.scorer. evaluate_on_environment</code>	Returns scorer function of evaluation on environment.
<code>d3rlpy.metrics.comparer. compare_continuous_action_diff</code>	Returns scorer function of action difference between algorithms.
<code>d3rlpy.metrics.comparer. compare_discrete_action_match</code>	Returns scorer function of action matches between algorithms.

### d3rlpy.metrics.scorer.td\_error\_scorer

`d3rlpy.metrics.scorer.td_error_scorer` (*algo*, *episodes*, *window\_size=1024*)

Returns average TD error (in negative scale).

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

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

#### Parameters

- **algo** (`d3rlpy.algos.base.AlgoBase`) – algorithm.
- **episodes** (`list` (`d3rlpy.dataset.Episode`)) – list of episodes.
- **window\_size** (`int`) – mini-batch size to compute.

**Returns** negative average TD error.

**Return type** `float`

### d3rlpy.metrics.scorer.discounted\_sum\_of\_advantage\_scorer

`d3rlpy.metrics.scorer.discounted_sum_of_advantage_scorer` (*algo*, *episodes*, *window\_size=1024*)

Returns average of discounted sum of advantage (in negative scale).



This metrics suggests how the greedy-policy selects different actions in action-value space. If the sum of advantage is small, the policy selects actions with larger estimated action-values.

$$\mathbb{E}_{s_t, a_t \sim D} \left[ \sum_{t'=t} \gamma^{t'-t} A(s_{t'}, a_{t'}) \right]$$

where  $A(s_t, a_t) = Q_\theta(s_t, a_t) - \max_a Q_\theta(s_t, a)$ .

## References

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

### Parameters

- **algo** (`d3rlpy.algos.base.AlgoBase`) – algorithm.
- **episodes** (`list(d3rlpy.dataset.Episode)`) – list of episodes.
- **window\_size** (`int`) – mini-batch size to compute.

**Returns** negative average of discounted sum of advantage.

**Return type** `float`

### d3rlpy.metrics.scorer.average\_value\_estimation\_scorer

`d3rlpy.metrics.scorer.average_value_estimation_scorer(algo, episodes, window_size=1024)`

Returns average value estimation (in negative scale).

This metrics suggests the scale for estimation of Q functions. If average value estimation is too large, the Q functions overestimate action-values, which possibly makes training failed.

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

### Parameters

- **algo** (`d3rlpy.algos.base.AlgoBase`) – algorithm.
- **episodes** (`list(d3rlpy.dataset.Episode)`) – list of episodes.
- **window\_size** (`int`) – mini-batch size to compute.

**Returns** negative average value estimation.

**Return type** `float`

### d3rlpy.metrics.scorer.value\_estimation\_std\_scorer

`d3rlpy.metrics.scorer.value_estimation_std_scorer(algo, episodes, window_size=1024)`

Returns standard deviation of value estimation (in negative scale).

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

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

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

**Parameters**

- **algo** (`d3rlpy.algos.base.AlgoBase`) – algorithm.
- **episodes** (`list(d3rlpy.dataset.Episode)`) – list of episodes.
- **window\_size** (`int`) – mini-batch size to compute.

**d3rlpy.metrics.scorer.continuous\_action\_diff\_scorer**

`d3rlpy.metrics.scorer.continuous_action_diff_scorer` (*algo*, *episodes*, *window\_size=1024*)

Returns squared difference of actions between algorithm and dataset.

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

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

**Parameters**

- **algo** (`d3rlpy.algos.base.AlgoBase`) – algorithm.
- **episodes** (`list(d3rlpy.dataset.Episode)`) – list of episodes.
- **window\_size** (`int`) – mini-batch size to compute.

**Returns** negative squared action difference.

**Return type** `float`

**d3rlpy.metrics.scorer.discrete\_action\_match\_scorer**

`d3rlpy.metrics.scorer.discrete_action_match_scorer` (*algo*, *episodes*, *window\_size=1024*)

Returns percentage of identical actions between algorithm and dataset.

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

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

**Parameters**

- **algo** (`d3rlpy.algos.base.AlgoBase`) – algorithm.
- **episodes** (`list(d3rlpy.dataset.Episode)`) – list of episodes.
- **window\_size** (`int`) – mini-batch size to compute.

**Returns** percentage of identical actions.

**Return type** `float`

**d3rlpy.metrics.scorer.evaluate\_on\_environment**

`d3rlpy.metrics.scorer.evaluate_on_environment` (*env*, *n\_trials=10*, *epsilon=0.0*, *render=False*)

Returns scorer function of evaluation on environment.

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

```
import gym

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

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

scorer = evaluate_on_environment(env)

cql = CQL()

mean_episode_return = scorer(cql)
```

#### Parameters

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

**Returns** scorer function.

**Return type** callable

### d3rlpy.metrics.comparer.compare\_continuous\_action\_diff

d3rlpy.metrics.comparer.**compare\_continuous\_action\_diff** (*base\_algo*, *window\_size=1024*) *win-*

Returns scorer function of action difference between algorithms.

This metrics suggests how different the two algorithms are in continuous action-space. If the algorithm to compare with is near-optimal, the small action difference would be better.

$$\mathbb{E}_{s_t \sim D}[(\pi_{\phi_1}(s_t) - \pi_{\phi_2}(s_t))^2]$$

```
from d3rlpy.algos import CQL
from d3rlpy.metrics.comparer import compare_continuous_action_diff

cql1 = CQL()
cql2 = CQL()

scorer = compare_continuous_action_diff(cql1)

squared_action_diff = scorer(cql2, ...)
```

#### Parameters

- **base\_algo** (*d3rlpy.algos.base.AlgoBase*) – algorithm to compare with.
- **window\_size** (*int*) – mini-batch size to compute.

**Returns** scorer function.

**Return type** callable

### d3rlpy.metrics.comparer.compare\_discrete\_action\_match

d3rlpy.metrics.comparer.**compare\_discrete\_action\_match**(*base\_algo*, *window\_size=1024*)

Returns scorer function of action matches between algorithms.

This metrics suggests how different the two algorithms are in discrete action-space. If the algorithm to compare with is near-optimal, the small action difference would be better.

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

```
from d3rlpy.algos import DQN
from d3rlpy.metrics.comparer import compare_continuous_action_diff

dqn1 = DQN()
dqn2 = DQN()

scorer = compare_continuous_action_diff(dqn1)

percentage_of_identical_actions = scorer(dqn2, ...)
```

#### Parameters

- **base\_algo** (*d3rlpy.algos.base.AlgoBase*) – algorithm to compare with.
- **window\_size** (*int*) – mini-batch size to compute.

**Returns** scorer function.

**Return type** callable

## 2.7.2 Dynamics

<code>d3rlpy.metrics.scorer.dynamics_observation_prediction_error_scorer</code>	Returns MSE of observation prediction (in negative scale).
<code>d3rlpy.metrics.scorer.dynamics_reward_prediction_error_scorer</code>	Returns MSE of reward prediction (in negative scale).
<code>d3rlpy.metrics.scorer.dynamics_prediction_variance_scorer</code>	Returns prediction variance of ensemble dynamics (in negative scale).

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

d3rlpy.metrics.scorer.**dynamics\_observation\_prediction\_error\_scorer**(*dynamics*, *episodes*, *window\_size=1024*)

Returns MSE of observation prediction (in negative scale).

This metrics suggests how dynamics model is generalized to test sets. If the MSE is large, the dynamics model are overfitting.

$$\mathbb{E}_{s_t, a_t, s_{t+1} \sim D} [(s_{t+1} - s')^2]$$

where  $s' \sim T(s_t, a_t)$ .

#### Parameters

- **dynamics** (`d3rlpy.dynamics.base.DynamicsBase`) – dynamics model.
- **episodes** (`list(d3rlpy.dataset.Episode)`) – list of episodes.
- **window\_size** (`int`) – mini-batch size to compute.

**Returns** negative mean squared error.

**Return type** `float`

### `d3rlpy.metrics.scorer.dynamics_reward_prediction_error_scorer`

`d3rlpy.metrics.scorer.dynamics_reward_prediction_error_scorer(dynamics, episodes, window_size=1024)`

Returns MSE of reward prediction (in negative scale).

This metrics suggests how dynamics model is generalized to test sets. If the MSE is large, the dynamics model are overfitting.

$$\mathbb{E}_{s_t, a_t, r_{t+1} \sim D}[(r_{t+1} - r')^2]$$

where  $r' \sim T(s_t, a_t)$ .

#### Parameters

- **dynamics** (`d3rlpy.dynamics.base.DynamicsBase`) – dynamics model.
- **episodes** (`list(d3rlpy.dataset.Episode)`) – list of episodes.
- **window\_size** (`int`) – mini-batch size to compute.

**Returns** negative mean squared error.

**Return type** `float`

### `d3rlpy.metrics.scorer.dynamics_prediction_variance_scorer`

`d3rlpy.metrics.scorer.dynamics_prediction_variance_scorer(dynamics, episodes, window_size=1024)`

Returns prediction variance of ensemble dynamics (in negative scale).

This metrics suggests how dynamics model is confident of test sets. If the variance is large, the dynamics model has large uncertainty.

#### Parameters

- **dynamics** (`d3rlpy.dynamics.base.DynamicsBase`) – dynamics model.
- **episodes** (`list(d3rlpy.dataset.Episode)`) – list of episodes.
- **window\_size** (`int`) – mini-batch size to compute.

**Returns** negative variance.

**Return type** `float`

## 2.8 Save and Load

### 2.8.1 save\_model and load\_model

```
from d3rlpy.datasets import get_cartpole
from d3rlpy.algos import DQN

dataset, env = get_cartpole()

dqn = DQN(n_epochs=1)
dqn.fit(dataset.episodes)

# save entire model parameters.
dqn.save_model('model.pt')

# load entire model parameters.
dqn.load_model('model.pt')
```

*save\_model* method saves all parameters including optimizer states, which is useful when checking all the outputs or re-training from snapshots.

### 2.8.2 from\_json

It is very boring to set the same hyperparameters to initialize algorithms when loading model parameters. In d3rlpy, *params.json* is saved at the beginning of *fit* method, which includes all hyperparameters within the algorithm object. You can recreate algorithm objects from *params.json* via *from\_json* method.

```
from d3rlpy.algos import DQN

dqn = DQN.from_json('d3rlpy_logs/<path-to-json>/params.json')

# ready to load
dqn.load_model('model.pt')
```

### 2.8.3 save\_policy

```
from d3rlpy.datasets import get_cartpole
from d3rlpy.algos import DQN

dataset, env = get_cartpole()

dqn = DQN(n_epochs=1)
dqn.fit(dataset.episodes)

# save only greedy-policy
dqn.save_policy('policy.pt')
```

*save\_policy* method saves the only greedy-policy computation graph as TorchScript. TorchScript is a optimizable graph expression provided by PyTorch. When *save\_policy* method is called, the greedy-policy graph is constructed and traced via *torch.jit.trace* function. The saved policy can be loaded without any dependencies except PyTorch.

```
import torch

# load greedy-policy only with PyTorch
policy = torch.jit.load('policy.pt')

# returns greedy actions
actions = policy(torch.rand(32, 6))
```

This is especially useful when deploying the trained models to productions. The computation can be faster and you don't need to install d3rlpy. Moreover, TorchScript model can be easily loaded even with C++, which will empower your robotics and embedding system projects.

```
#include <torch/script.h>

int main(int argc, char* argv[]) {
    torch::jit::script::Module module;
    try {
        module = torch::jit::load("policy.pt")
    } catch (const c10::Error& e) {
        return -1;
    }
    return 0;
}
```

You can get more information about TorchScript [here](#).

## 2.9 Logging

d3rlpy algorithms automatically save model parameters and metrics under `d3rlpy_logs` directory.

```
from d3rlpy.datasets import get_cartpole
from d3rlpy.algos import DQN

dataset, env = get_cartpole()

dqn = DQN()

# metrics and parameters are saved in `d3rlpy_logs/DQN_YYYYMMDDHHmmss`
dqn.fit(dataset.episodes)
```

You can designate the directory.

```
# the directory will be `custom_logs/custom_YYYYMMDDHHmmss`
dqn.fit(dataset.episodes, logdir='custom_logs', experiment_name='custom')
```

The same information is also automatically saved for tensorboard under `runs` directory. You can interactively visualize training metrics easily.

```
$ pip install tensorboard
$ tensorboard --logdir runs
```

This tensorboard logs can be disabled by passing `tensorboard=False`.

```
dqn.fit(dataset.episodes, tensorboard=False)
```

## 2.10 scikit-learn compatibility

d3rlpy provides complete scikit-learn compatible APIs.

### 2.10.1 train\_test\_split

`d3rlpy.dataset.MDPDataset` is compatible with splitting functions in scikit-learn.

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

dataset, env = get_cartpole()

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

dqn = DQN(n_epochs=1)
dqn.fit(train_episodes,
        eval_episodes=test_episodes,
        scorers={'td_error': td_error_scorer})
```

### 2.10.2 cross\_validate

cross validation is also easily performed.

```
from d3rlpy.algos import DQN
from d3rlpy.datasets import get_cartpole
from d3rlpy.metrics import td_error_scorer
from sklearn.model_selection import cross_validate

dataset, env = get_cartpole()

dqn = DQN(n_epochs=1)

scores = cross_validate(dqn, dataset, scoring={'td_error': td_error_scorer})
```

### 2.10.3 GridSearchCV

You can also perform grid search to find good hyperparameters.

```
from d3rlpy.algos import DQN
from d3rlpy.datasets import get_cartpole
from d3rlpy.metrics import td_error_scorer
from sklearn.model_selection import GridSearchCV

dataset, env = get_cartpole()

dqn = DQN(n_epochs=1)

gscv = GridSearchCV(estimator=dqn,
                    param_grid={'learning_rate': [1e-4, 3e-4, 1e-3]},
```

(continues on next page)



(continued from previous page)

```

        scoring={'td_error': td_error_scorer},
        refit=False)

gscv.fit(dataset.episodes)

```

## 2.10.4 parallel execution with multiple GPUs

Some scikit-learn utilities provide `n_jobs` option, which enable fitting process to run in parallel to boost productivity. Ideally, if you have multiple GPUs, the multiple processes use different GPUs for computational efficiency.

d3rlpy provides special device assignment mechanism to realize this.

```

from d3rlpy.algos import DQN
from d3rlpy.datasets import get_cartpole
from d3rlpy.metrics import td_error_scorer
from d3rlpy.context import parallel
from sklearn.model_selection import cross_validate

dataset, env = get_cartpole()

# enable GPU
dqn = DQN(n_epochs=1, use_gpu=True)

# automatically assign different GPUs for the 4 processes.
with parallel():
    scores = cross_validate(dqn,
                           dataset,
                           scoring={'td_error': td_error_scorer},
                           n_jobs=4)

```

If `use_gpu=True` is passed, d3rlpy internally manages GPU device id via `d3rlpy.gpu.Device` object. This object is designed for scikit-learn's multi-process implementation that makes deep copies of the estimator object before dispatching. The `Device` object will increment its device id when deeply copied under the parallel context.

```

import copy
from d3rlpy.context import parallel
from d3rlpy.gpu import Device

device = Device(0)
# device.get_id() == 0

new_device = copy.deepcopy(device)
# new_device.get_id() == 0

with parallel():
    new_device = copy.deepcopy(device)
    # new_device.get_id() == 1
    # device.get_id() == 1

    new_device = copy.deepcopy(device)
    # if you have only 2 GPUs, it goes back to 0.
    # new_device.get_id() == 0
    # device.get_id() == 0

from d3rlpy.algos import DQN

```

(continues on next page)

(continued from previous page)

```
dqn = DQN(use_gpu=Device(0)) # assign id=0
dqn = DQN(use_gpu=Device(1)) # assign id=1
```

## 2.11 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 gym

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

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

# setup algorithm
dqn = DQN(n_epochs=30,
          batch_size=32,
          learning_rate=2.5e-4,
          target_update_interval=100,
          use_gpu=True)

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

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

# start training
train(env,
      dqn,
      buffer,
      explorer=explorer, # you don't need this with probabilistic policy algorithms
      eval_env=eval_env,
      n_steps_per_epoch=1000,
      n_updates_per_epoch=100)
```

### 2.11.1 Replay Buffer

---

*d3rlpy.online.buffers.ReplayBuffer*Standard Replay Buffer.

---

**d3rlpy.online.buffers.ReplayBuffer**

**class** d3rlpy.online.buffers.**ReplayBuffer** (*maxlen*, *env*)  
 Standard Replay Buffer.

**Parameters**

- **maxlen** (*int*) – the maximum number of data length.
- **env** (*gym.Env*) – gym-like environment to extract shape information.

**maxlen**

the maximum number of data length

**Type** *int*

**observations**

list of observations.

**Type** *list(numpy.ndarray)*

**actions**

list of actions.

**Type** *list(numpy.ndarray)* or *list(int)*

**rewards**

list of rewards.

**Type** *list(float)*

**terminals**

list of terminal flags.

**Type** *list(float)*

**cursor**

current cursor pointing to list location to insert.

**Type** *int*

**observation\_shape**

observation shape.

**Type** *tuple*

**action\_size**

action size.

**Type** *int*

**Methods**

**\_\_len\_\_** ()

**append** (*observation*, *action*, *reward*, *terminal*)

Append observation, action, reward and terminal flag to buffer.

**Parameters**

- **observation** (*numpy.ndarray*) – observation.
- **action** (*numpy.ndarray* or *int*) – action.
- **reward** (*float*) – reward.

- **terminal** (*bool or float*) – terminal flag.

**sample** (*batch\_size*)

Returns sampled mini-batch of transitions.

**Parameters** **batch\_size** (*int*) – mini-batch size.

**Returns** mini-batch.

**Return type** *d3rlpy.dataset.TransitionMiniBatch*

**size** ()

Returns the number of appended elements in buffer.

**Returns** the number of elements in buffer.

**Return type** *int*

## 2.11.2 Explorers

<i>d3rlpy.online.explorers.</i> <i>LinearDecayEpsilonGreedy</i>	$\epsilon$ -greedy explorer with linear decay schedule.
<i>d3rlpy.online.explorers.NormalNoise</i>	Normal noise explorer.

### **d3rlpy.online.explorers.LinearDecayEpsilonGreedy**

**class** `d3rlpy.online.explorers.LinearDecayEpsilonGreedy` (*start\_epsilon=1.0,*  
*end\_epsilon=0.1,* *duration=1000000*)

$\epsilon$ -greedy explorer with linear decay schedule.

**Parameters**

- **start\_epsilon** (*float*) – the beginning  $\epsilon$ .
- **end\_epsilon** (*float*) – the end  $\epsilon$ .
- **duration** (*int*) – the scheduling duration.

**start\_epsilon**

the beginning  $\epsilon$ .

**Type** *float*

**end\_epsilon**

the end  $\epsilon$ .

**Type** *float*

**duration**

the scheduling duration.

**Type** *int*

### **Methods**

**compute\_epsilon** (*step*)

Returns decayed  $\epsilon$ .

**Returns**  $\epsilon$ .

**Return type** `float`

**sample** (*algo*, *x*, *step*)  
Returns  $\epsilon$ -greedy action.

**Parameters**

- **algo** (*d3rlpy.algos.base.AlgoBase*) – algorithm.
- **x** (*numpy.ndarray*) – observation.
- **step** (*int*) – current environment step.

**Returns**  $\epsilon$ -greedy action.

**Return type** `int`

### d3rlpy.online.explorers.NormalNoise

**class** `d3rlpy.online.explorers.NormalNoise` (*mean=0.0*, *std=0.1*)  
Normal noise explorer.

**Parameters**

- **mean** (*float*) – mean.
- **std** (*float*) – standard deviation.

**mean**  
mean.

**Type** `float`

**std**  
standard deviation.

**Type** `float`

### Methods

**sample** (*algo*, *x*, *\*args*)  
Returns action with noise injection.

**Parameters**

- **algo** (*d3rlpy.algos.base.AlgoBase*) – algorithm.
- **x** (*numpy.ndarray*) – observation.

**Returns** action with noise injection.

**Return type** `numpy.ndarray`

### 2.11.3 Iterators

---

`d3rlpy.online.iterators.train`

Start training loop of online deep reinforcement learning.

---

## d3rlpy.online.iterators.train

```
d3rlpy.online.iterators.train(env, algo, buffer, explorer=None, n_steps_per_epoch=4000,
                              n_updates_per_epoch=100, eval_env=None, eval_epsilon=0.05,
                              experiment_name=None, with_timestamp=True,
                              logdir='d3rlpy_logs', verbose=True, show_progress=True,
                              tensorboard=True, save_interval=1)
```

Start training loop of online deep reinforcement learning.

### Parameters

- **env** (*gym.Env*) – gym-like environment.
- **algo** (*d3rlpy.algos.base.AlgoBase*) – algorithm.
- **buffer** (*d3rlpy.online.buffers.Buffer*) – replay buffer.
- **explorer** (*d3rlpy.online.explorers.Explorer*) – action explorer.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch.
- **n\_updates\_per\_epoch** (*int*) – the number of updates per epoch.
- **eval\_env** (*gym.Env*) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (*float*) –  $\epsilon$ -greedy factor during evaluation.
- **experiment\_name** (*str*) – experiment name for logging. If not passed, the directory name will be *{class name}\_online\_{timestamp}*.
- **with\_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show\_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **save\_interval** (*int*) – interval to save parameters.

## 2.12 Model-based Data Augmentation

d3rlpy provides model-based reinforcement learning algorithms. In d3rlpy, model-based algorithms are viewed as data augmentation techniques, which can boost performance potentially beyond the model-free algorithms.

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

dataset, _ = get_pendulum()

train_episodes, test_episodes = train_test_split(dataset)

mopo = MOPO(n_epochs=100, learning_rate=1e-4, use_gpu=True)
```

(continues on next page)

(continued from previous page)

```
# same as algorithms
mopo.fit(train_episodes,
        eval_episodes=test_episodes,
        scorers={
            'observation_error': dynamics_observation_prediction_error_scorer,
            'reward_error': dynamics_reward_prediction_error_scorer,
            'variance': dynamics_prediction_variance_scorer,
        })
```

Pick the best model based on evaluation metrics.

```
from d3rlpy.dynamics import MOPO
from d3rlpy.algos import CQL

# load trained dynamics model
mopo = MOPO.from_json('<path-to-params.json>/params.json')
mopo.load_model('<path-to-model>/model_xx.pt')
mopo.n_transitions = 400 # tunable parameter
mopo.horizon = 5 # tunable parameter
mopo.lam = 1.0 # tunable parameter

# give mopo as dynamics argument.
cql = CQL(dynamics=mopo)
```

If you pass a dynamics model to algorithms, new transitions are generated at the beginning of every epoch.

---

*d3rlpy.dynamics.mopo.MOPO*


---

 Model-based Offline Policy Optimization.
 

---

### 2.12.1 d3rlpy.dynamics.mopo.MOPO

```
class d3rlpy.dynamics.mopo.MOPO (n_epochs=30, batch_size=100, learning_rate=0.001, eps=1e-08,
    weight_decay=0.0001, n_ensembles=5, n_transitions=400, horizon=5, lam=1.0, use_batch_norm=False,
    discrete_action=False, scaler=None, augmentation=[], use_gpu=False, impl=None, **kwargs)
```

Model-based Offline Policy Optimization.

MOPO is a model-based RL approach for offline policy optimization. MOPO leverages the probabilistic ensemble dynamics model to generate new dynamics data with uncertainty penalties.

The ensemble dynamics model consists of  $N$  probabilistic models  $\{T_{\theta_i}\}_{i=1}^N$ . At each epoch, new transitions are generated via randomly picked dynamics model  $T_{\theta}$ .

$$s_{t+1}, r_{t+1} \sim T_{\theta}(s_t, a_t)$$

where  $s_t \sim D$  for the first step, otherwise  $s_t$  is the previous generated observation, and  $a_t \sim \pi(\cdot|s_t)$ . The generated  $r_{t+1}$  would be far from the ground truth if the actions sampled from the policy function is out-of-distribution. Thus, the uncertainty penalty regularizes this bias.

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

where  $\Sigma(s_t, a_t)$  is the estimated variance.

Finally, the generated transitions  $(s_t, a_t, r_{t+1}^{\sim}, s_{t+1})$  are appended to dataset  $D$ .

This generation process starts with randomly sampled  $n\_transitions$  transitions till  $horizon$  steps.

---

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

---

## References

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

### Parameters

- **n\_epochs** (*int*) – the number of epochs to train.
- **batch\_size** (*int*) – mini-batch size.
- **learning\_rate** (*float*) – learning rate for dynamics model.
- **eps** (*float*) –  $\epsilon$  for Adam optimizer.
- **weight\_decay** (*float*) – weight decay rate.
- **n\_ensembles** (*int*) – the number of dynamics model for ensemble.
- **n\_transitions** (*int*) – the number of parallel trajectories to generate.
- **horizon** (*int*) – the number of steps to generate.
- **lam** (*float*) –  $\lambda$  for uncertainty penalties.
- **use\_batch\_norm** (*bool*) – flag to insert batch normalization layers.
- **discrete\_action** (*bool*) – flag to take discrete actions.
- **scaler** (*d3rlpy.preprocessing.scalers.Scaler or str*) – preprocessor. The available options are [*'pixel'*, *'min\_max'*, *'standard'*].
- **augmentation** (*d3rlpy.augmentation.AugmentationPipeline or list(str)*) – augmentation pipeline.
- **use\_gpu** (*bool or d3rlpy.gpu.Device*) – flag to use GPU or device.
- **impl** (*d3rlpy.dynamics.base.DynamicsImplBase*) – dynamics implementation.

### n\_epochs

the number of epochs to train.

**Type** *int*

### batch\_size

mini-batch size.

**Type** *int*

### learning\_rate

learning rate for dynamics model.

**Type** *float*

### eps

$\epsilon$  for Adam optimizer.

**Type** *float*

### weight\_decay

weight decay rate.



**Type** float

**n\_ensembles**  
the number of dynamics model for ensemble.

**Type** int

**n\_transitions**  
the number of parallel trajectories to generate.

**Type** int

**horizon**  
the number of steps to generate.

**Type** int

**lam**  
 $\lambda$  for uncertainty penalties.

**Type** float

**use\_batch\_norm**  
flag to insert batch normalization layers.

**Type** bool

**discrete\_action**  
flag to take discrete actions.

**Type** bool

**scaler**  
preprocessor.

**Type** d3rlpy.preprocessing.scalers.Scaler

**augmentation**  
augmentation pipeline.

**Type** d3rlpy.augmentation.AugmentationPipeline

**use\_gpu**  
flag to use GPU or device.

**Type** d3rlpy.gpu.Device

**impl**  
dynamics implementation.

**Type** d3rlpy.dynamics.base.DynamicsImplBase

## Methods

**create\_impl** (*observation\_shape*, *action\_size*)  
Instantiate implementation objects with the dataset shapes.  
This method will be used internally when *fit* method is called.

### Parameters

- **observation\_shape** (*tuple*) – observation shape.
- **action\_size** (*int*) – dimension of action-space.

**fit** (*episodes*, *experiment\_name=None*, *with\_timestamp=True*, *logdir='d3rlpy\_logs'*, *verbose=True*, *show\_progress=True*, *tensorboard=True*, *eval\_episodes=None*, *save\_interval=1*, *scorers=None*)  
Trains with the given dataset.

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

#### Parameters

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

**classmethod from\_json** (*fname*, *use\_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

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

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

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

#### Parameters

- **fname** (*str*) – file path to *params.json*.
- **use\_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.

**Returns** algorithm.

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

**generate** (*algo*, *transitions*)

Returns new transitions for data augmentation.

#### Parameters

- **algo** (*d3rlpy.algos.base.AlgoBase*) – algorithm.

- **transitions** (*list* (`d3rlpy.dataset.Transition`)) – list of transitions.

**Returns** list of generated transitions.

**Return type** `list(d3rlpy.dataset.Transition)`

**get\_params** (*deep=True*)

Returns the all attributes.

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

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

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

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

**Returns** attribute values in dictionary.

**Return type** `dict`

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

Load neural network parameters.

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

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

**predict** (*x, action, with\_variance=False*)

Returns predicted observation and reward.

**Parameters**

- **x** (*numpy.ndarray*) – observation
- **action** (*numpy.ndarray*) – action
- **with\_variance** (*bool*) – flag to return prediction variance.

**Returns** tuple of predicted observation and reward.

**Return type** `tuple`

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

Saves neural network parameters.

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

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

**set\_params** (*\*\*params*)

Sets the given arguments to the attributes if they exist.

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

```
algo.set_params(n_epochs=10, batch_size=100)
```

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

**Returns** itself.

**Return type** `d3rlpy.algos.base.AlgoBase`

**update** (*epoch*, *total\_step*, *batch*)

Update parameters with mini-batch of data.

**Parameters**

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

**Returns** loss values.

**Return type** `list`

### 3.1 Recommended Platforms

d3rlpy is only tested on Linux and macOS. However, you can possibly run d3rlpy on Windows as long as PyTorch runs since it's the core dependency.

### 3.2 from pip

*pip* is a recommended way to install d3rlpy:

```
$ pip install d3rlpy
```

### 3.3 from source

You can also install via GitHub repository:

```
$ git clone https://github.com/takuseno/d3rlpy
$ cd d3rlpy
$ pip install -e .
```



## CHAPTER 4

---

### License

---

#### MIT License

Copyright (c) 2020 Takuma Seno

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.





## CHAPTER 5

---

### Indices and tables

---

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



### d

- `d3rlpy`, 5
- `d3rlpy.algos`, 5
- `d3rlpy.augmentation`, 95
- `d3rlpy.dataset`, 80
- `d3rlpy.datasets`, 88
- `d3rlpy.dynamics`, 114
- `d3rlpy.metrics`, 99
- `d3rlpy.models.torch.q_functions`, 79
- `d3rlpy.online`, 110
- `d3rlpy.preprocessing`, 90



## Symbols

`__getitem__()` (*d3rlpy.dataset.Episode method*), 85  
`__getitem__()` (*d3rlpy.dataset.MDPDataset method*), 82  
`__getitem__()` (*d3rlpy.dataset.TransitionMiniBatch method*), 87  
`__iter__()` (*d3rlpy.dataset.Episode method*), 85  
`__iter__()` (*d3rlpy.dataset.MDPDataset method*), 82  
`__iter__()` (*d3rlpy.dataset.TransitionMiniBatch method*), 87  
`__len__()` (*d3rlpy.dataset.Episode method*), 85  
`__len__()` (*d3rlpy.dataset.MDPDataset method*), 82  
`__len__()` (*d3rlpy.dataset.TransitionMiniBatch method*), 87  
`__len__()` (*d3rlpy.online.buffers.ReplayBuffer method*), 111

## A

`action` (*d3rlpy.dataset.Transition attribute*), 86  
`action_flexibility` (*d3rlpy.algos.BCQ attribute*), 32  
`action_flexibility` (*d3rlpy.algos.DiscreteBCQ attribute*), 69  
`action_size` (*d3rlpy.online.buffers.ReplayBuffer attribute*), 111  
`actions` (*d3rlpy.dataset.Episode attribute*), 85  
`actions` (*d3rlpy.dataset.MDPDataset attribute*), 83  
`actions` (*d3rlpy.dataset.TransitionMiniBatch attribute*), 88  
`actions` (*d3rlpy.online.buffers.ReplayBuffer attribute*), 111  
`actor_learning_rate` (*d3rlpy.algos.BCQ attribute*), 31  
`actor_learning_rate` (*d3rlpy.algos.BEAR attribute*), 38  
`actor_learning_rate` (*d3rlpy.algos.CQL attribute*), 45  
`actor_learning_rate` (*d3rlpy.algos.DDPG attribute*), 11

`actor_learning_rate` (*d3rlpy.algos.SAC attribute*), 24  
`actor_learning_rate` (*d3rlpy.algos.TD3 attribute*), 17  
`alpha_learning_rate` (*d3rlpy.algos.BEAR attribute*), 38  
`alpha_learning_rate` (*d3rlpy.algos.CQL attribute*), 45  
`alpha_threshold` (*d3rlpy.algos.BEAR attribute*), 39  
`alpha_threshold` (*d3rlpy.algos.CQL attribute*), 46  
`append()` (*d3rlpy.dataset.MDPDataset method*), 82  
`append()` (*d3rlpy.online.buffers.ReplayBuffer method*), 111  
`augmentation` (*d3rlpy.algos.BC attribute*), 7  
`augmentation` (*d3rlpy.algos.BCQ attribute*), 33  
`augmentation` (*d3rlpy.algos.BEAR attribute*), 40  
`augmentation` (*d3rlpy.algos.CQL attribute*), 47  
`augmentation` (*d3rlpy.algos.DDPG attribute*), 12  
`augmentation` (*d3rlpy.algos.DiscreteBC attribute*), 52  
`augmentation` (*d3rlpy.algos.DiscreteBCQ attribute*), 70  
`augmentation` (*d3rlpy.algos.DiscreteCQL attribute*), 75  
`augmentation` (*d3rlpy.algos.DoubleDQN attribute*), 63  
`augmentation` (*d3rlpy.algos.DQN attribute*), 57  
`augmentation` (*d3rlpy.algos.SAC attribute*), 25  
`augmentation` (*d3rlpy.algos.TD3 attribute*), 19  
`augmentation` (*d3rlpy.dynamics.mopo.MOPO attribute*), 117  
`average_value_estimation_scorer()` (in module *d3rlpy.metrics.scorer*), 101

## B

`batch_size` (*d3rlpy.algos.BC attribute*), 6  
`batch_size` (*d3rlpy.algos.BCQ attribute*), 31  
`batch_size` (*d3rlpy.algos.BEAR attribute*), 38  
`batch_size` (*d3rlpy.algos.CQL attribute*), 46  
`batch_size` (*d3rlpy.algos.DDPG attribute*), 11  
`batch_size` (*d3rlpy.algos.DiscreteBC attribute*), 52

batch\_size (*d3rlpy.algos.DiscreteBCQ attribute*), 69  
 batch\_size (*d3rlpy.algos.DiscreteCQL attribute*), 75  
 batch\_size (*d3rlpy.algos.DoubleDQN attribute*), 62  
 batch\_size (*d3rlpy.algos.DQN attribute*), 57  
 batch\_size (*d3rlpy.algos.SAC attribute*), 24  
 batch\_size (*d3rlpy.algos.TD3 attribute*), 17  
 batch\_size (*d3rlpy.dynamics.mopo.MOPO attribute*), 116  
 BC (*class in d3rlpy.algos*), 5  
 BCQ (*class in d3rlpy.algos*), 29  
 BEAR (*class in d3rlpy.algos*), 36  
 beta (*d3rlpy.algos.BCQ attribute*), 32  
 beta (*d3rlpy.algos.DiscreteBC attribute*), 52  
 beta (*d3rlpy.algos.DiscreteBCQ attribute*), 69  
 bootstrap (*d3rlpy.algos.BCQ attribute*), 31  
 bootstrap (*d3rlpy.algos.BEAR attribute*), 39  
 bootstrap (*d3rlpy.algos.CQL attribute*), 46  
 bootstrap (*d3rlpy.algos.DDPG attribute*), 11  
 bootstrap (*d3rlpy.algos.DiscreteBCQ attribute*), 69  
 bootstrap (*d3rlpy.algos.DiscreteCQL attribute*), 75  
 bootstrap (*d3rlpy.algos.DoubleDQN attribute*), 63  
 bootstrap (*d3rlpy.algos.DQN attribute*), 57  
 bootstrap (*d3rlpy.algos.SAC attribute*), 24  
 bootstrap (*d3rlpy.algos.TD3 attribute*), 18

## C

clip\_reward() (*d3rlpy.dataset.MDPDataset method*), 82  
 compare\_continuous\_action\_diff() (*in module d3rlpy.metrics.comparer*), 103  
 compare\_discrete\_action\_match() (*in module d3rlpy.metrics.comparer*), 104  
 compute\_epsilon() (*d3rlpy.online.explorers.LinearDecayEpsilonGreedy method*), 112  
 compute\_return() (*d3rlpy.dataset.Episode method*), 85  
 compute\_stats() (*d3rlpy.dataset.MDPDataset method*), 82  
 continuous\_action\_diff\_scorer() (*in module d3rlpy.metrics.scorer*), 102  
 CQL (*class in d3rlpy.algos*), 44  
 create\_impl() (*d3rlpy.algos.BC method*), 7  
 create\_impl() (*d3rlpy.algos.BCQ method*), 33  
 create\_impl() (*d3rlpy.algos.BEAR method*), 40  
 create\_impl() (*d3rlpy.algos.CQL method*), 47  
 create\_impl() (*d3rlpy.algos.DDPG method*), 13  
 create\_impl() (*d3rlpy.algos.DiscreteBC method*), 53  
 create\_impl() (*d3rlpy.algos.DiscreteBCQ method*), 70  
 create\_impl() (*d3rlpy.algos.DiscreteCQL method*), 76

create\_impl() (*d3rlpy.algos.DoubleDQN method*), 64  
 create\_impl() (*d3rlpy.algos.DQN method*), 58  
 create\_impl() (*d3rlpy.algos.SAC method*), 26  
 create\_impl() (*d3rlpy.algos.TD3 method*), 19  
 create\_impl() (*d3rlpy.dynamics.mopo.MOPO method*), 117  
 critic\_learning\_rate (*d3rlpy.algos.BCQ attribute*), 31  
 critic\_learning\_rate (*d3rlpy.algos.BEAR attribute*), 38  
 critic\_learning\_rate (*d3rlpy.algos.CQL attribute*), 45  
 critic\_learning\_rate (*d3rlpy.algos.DDPG attribute*), 11  
 critic\_learning\_rate (*d3rlpy.algos.SAC attribute*), 24  
 critic\_learning\_rate (*d3rlpy.algos.TD3 attribute*), 17  
 cursor (*d3rlpy.online.buffers.ReplayBuffer attribute*), 111  
 Cutout (*class in d3rlpy.augmentation.image*), 96

## D

d3rlpy (*module*), 5  
 d3rlpy.algos (*module*), 5  
 d3rlpy.augmentation (*module*), 95  
 d3rlpy.dataset (*module*), 80  
 d3rlpy.datasets (*module*), 88  
 d3rlpy.dynamics (*module*), 114  
 d3rlpy.metrics (*module*), 99  
 d3rlpy.models.torch.q\_functions (*module*), 79  
 d3rlpy.online (*module*), 110  
 d3rlpy.preprocessing (*module*), 90  
 DDPG (*class in d3rlpy.algos*), 10  
 degree (*d3rlpy.augmentation.image.RandomRotation attribute*), 98  
 discounted\_sum\_of\_advantage\_scorer() (*in module d3rlpy.metrics.scorer*), 100  
 discrete\_action (*d3rlpy.dynamics.mopo.MOPO attribute*), 117  
 discrete\_action\_match\_scorer() (*in module d3rlpy.metrics.scorer*), 102  
 DiscreteBC (*class in d3rlpy.algos*), 51  
 DiscreteBCQ (*class in d3rlpy.algos*), 67  
 DiscreteCQL (*class in d3rlpy.algos*), 74  
 DoubleDQN (*class in d3rlpy.algos*), 61  
 DQN (*class in d3rlpy.algos*), 56  
 dump() (*d3rlpy.dataset.MDPDataset method*), 82  
 duration (*d3rlpy.online.explorers.LinearDecayEpsilonGreedy attribute*), 112  
 dynamics (*d3rlpy.algos.BC attribute*), 7  
 dynamics (*d3rlpy.algos.BCQ attribute*), 33

dynamics (*d3rlpy.algos.BEAR* attribute), 40  
 dynamics (*d3rlpy.algos.CQL* attribute), 47  
 dynamics (*d3rlpy.algos.DDPG* attribute), 12  
 dynamics (*d3rlpy.algos.DiscreteBC* attribute), 52  
 dynamics (*d3rlpy.algos.DiscreteBCQ* attribute), 70  
 dynamics (*d3rlpy.algos.DiscreteCQL* attribute), 76  
 dynamics (*d3rlpy.algos.DoubleDQN* attribute), 64  
 dynamics (*d3rlpy.algos.DQN* attribute), 58  
 dynamics (*d3rlpy.algos.SAC* attribute), 25  
 dynamics (*d3rlpy.algos.TD3* attribute), 19  
 dynamics\_observation\_prediction\_error\_scorer()  
     (in module *d3rlpy.metrics.scorer*), 104  
 dynamics\_prediction\_variance\_scorer()  
     (in module *d3rlpy.metrics.scorer*), 105  
 dynamics\_reward\_prediction\_error\_scorer()  
     (in module *d3rlpy.metrics.scorer*), 105

## E

encoder\_params (*d3rlpy.algos.BC* attribute), 7  
 encoder\_params (*d3rlpy.algos.BCQ* attribute), 33  
 encoder\_params (*d3rlpy.algos.BEAR* attribute), 40  
 encoder\_params (*d3rlpy.algos.CQL* attribute), 47  
 encoder\_params (*d3rlpy.algos.DDPG* attribute), 12  
 encoder\_params (*d3rlpy.algos.DiscreteBC* attribute), 52  
 encoder\_params (*d3rlpy.algos.DiscreteBCQ* attribute), 70  
 encoder\_params (*d3rlpy.algos.DiscreteCQL* attribute), 76  
 encoder\_params (*d3rlpy.algos.DoubleDQN* attribute), 64  
 encoder\_params (*d3rlpy.algos.DQN* attribute), 58  
 encoder\_params (*d3rlpy.algos.SAC* attribute), 25  
 encoder\_params (*d3rlpy.algos.TD3* attribute), 19  
 end\_epsilon (*d3rlpy.online.explorers.LinearDecayEpsilonGreedy* attribute), 112  
 Episode (class in *d3rlpy.dataset*), 84  
 episodes (*d3rlpy.dataset.MDPDataset* attribute), 83  
 eps (*d3rlpy.algos.BC* attribute), 6  
 eps (*d3rlpy.algos.BCQ* attribute), 32  
 eps (*d3rlpy.algos.BEAR* attribute), 39  
 eps (*d3rlpy.algos.CQL* attribute), 46  
 eps (*d3rlpy.algos.DDPG* attribute), 12  
 eps (*d3rlpy.algos.DiscreteBC* attribute), 52  
 eps (*d3rlpy.algos.DiscreteBCQ* attribute), 69  
 eps (*d3rlpy.algos.DiscreteCQL* attribute), 75  
 eps (*d3rlpy.algos.DoubleDQN* attribute), 63  
 eps (*d3rlpy.algos.DQN* attribute), 57  
 eps (*d3rlpy.algos.SAC* attribute), 25  
 eps (*d3rlpy.algos.TD3* attribute), 18  
 eps (*d3rlpy.dynamics.mopo.MOPO* attribute), 116  
 evaluate\_on\_environment() (in module *d3rlpy.metrics.scorer*), 102

## F

fit() (*d3rlpy.algos.BC* method), 7  
 fit() (*d3rlpy.algos.BCQ* method), 33  
 fit() (*d3rlpy.algos.BEAR* method), 40  
 fit() (*d3rlpy.algos.CQL* method), 47  
 fit() (*d3rlpy.algos.DDPG* method), 13  
 fit() (*d3rlpy.algos.DiscreteBC* method), 53  
 fit() (*d3rlpy.algos.DiscreteBCQ* method), 70  
 fit() (*d3rlpy.algos.DiscreteCQL* method), 76  
 fit() (*d3rlpy.algos.DoubleDQN* method), 64  
 fit() (*d3rlpy.algos.DQN* method), 58  
 fit() (*d3rlpy.algos.SAC* method), 26  
 fit() (*d3rlpy.algos.TD3* method), 19  
 fit() (*d3rlpy.dynamics.mopo.MOPO* method), 117  
 fit() (*d3rlpy.preprocessing.MinMaxScaler* method), 92  
 fit() (*d3rlpy.preprocessing.PixelScaler* method), 91  
 fit() (*d3rlpy.preprocessing.StandardScaler* method), 94  
 from\_json() (*d3rlpy.algos.BC* class method), 8  
 from\_json() (*d3rlpy.algos.BCQ* class method), 34  
 from\_json() (*d3rlpy.algos.BEAR* class method), 41  
 from\_json() (*d3rlpy.algos.CQL* class method), 48  
 from\_json() (*d3rlpy.algos.DDPG* class method), 13  
 from\_json() (*d3rlpy.algos.DiscreteBC* class method), 53  
 from\_json() (*d3rlpy.algos.DiscreteBCQ* class method), 71  
 from\_json() (*d3rlpy.algos.DiscreteCQL* class method), 77  
 from\_json() (*d3rlpy.algos.DoubleDQN* class method), 64  
 from\_json() (*d3rlpy.algos.DQN* class method), 59  
 from\_json() (*d3rlpy.algos.SAC* class method), 26  
 from\_json() (*d3rlpy.algos.TD3* class method), 20  
 from\_json() (*d3rlpy.dynamics.mopo.MOPO* class method), 118

## G

gamma (*d3rlpy.algos.BCQ* attribute), 31  
 gamma (*d3rlpy.algos.BEAR* attribute), 38  
 gamma (*d3rlpy.algos.CQL* attribute), 46  
 gamma (*d3rlpy.algos.DDPG* attribute), 11  
 gamma (*d3rlpy.algos.DiscreteBCQ* attribute), 69  
 gamma (*d3rlpy.algos.DiscreteCQL* attribute), 75  
 gamma (*d3rlpy.algos.DoubleDQN* attribute), 63  
 gamma (*d3rlpy.algos.DQN* attribute), 57  
 gamma (*d3rlpy.algos.SAC* attribute), 24  
 gamma (*d3rlpy.algos.TD3* attribute), 17  
 generate() (*d3rlpy.dynamics.mopo.MOPO* method), 118  
 get\_action\_size() (*d3rlpy.dataset.Episode* method), 85

`get_action_size()` (*d3rlpy.dataset.MDPDataset method*), 83  
`get_action_size()` (*d3rlpy.dataset.Transition method*), 86  
`get_atari()` (*in module d3rlpy.datasets*), 90  
`get_cartpole()` (*in module d3rlpy.datasets*), 89  
`get_observation_shape()` (*d3rlpy.dataset.Episode method*), 85  
`get_observation_shape()` (*d3rlpy.dataset.MDPDataset method*), 83  
`get_observation_shape()` (*d3rlpy.dataset.Transition method*), 86  
`get_params()` (*d3rlpy.algos.BC method*), 8  
`get_params()` (*d3rlpy.algos.BCQ method*), 34  
`get_params()` (*d3rlpy.algos.BEAR method*), 41  
`get_params()` (*d3rlpy.algos.CQL method*), 48  
`get_params()` (*d3rlpy.algos.DDPG method*), 14  
`get_params()` (*d3rlpy.algos.DiscreteBC method*), 54  
`get_params()` (*d3rlpy.algos.DiscreteBCQ method*), 71  
`get_params()` (*d3rlpy.algos.DiscreteCQL method*), 77  
`get_params()` (*d3rlpy.algos.DoubleDQN method*), 65  
`get_params()` (*d3rlpy.algos.DQN method*), 59  
`get_params()` (*d3rlpy.algos.SAC method*), 27  
`get_params()` (*d3rlpy.algos.TD3 method*), 20  
`get_params()` (*d3rlpy.augmentation.image.Cutout method*), 96  
`get_params()` (*d3rlpy.augmentation.image.HorizontalFlip method*), 97  
`get_params()` (*d3rlpy.augmentation.image.Intensity method*), 99  
`get_params()` (*d3rlpy.augmentation.image.RandomRotation method*), 98  
`get_params()` (*d3rlpy.augmentation.image.RandomShift method*), 95  
`get_params()` (*d3rlpy.augmentation.image.VerticalFlip method*), 97  
`get_params()` (*d3rlpy.dynamics.mopo.MOPO method*), 119  
`get_params()` (*d3rlpy.preprocessing.MinMaxScaler method*), 92  
`get_params()` (*d3rlpy.preprocessing.PixelScaler method*), 91  
`get_params()` (*d3rlpy.preprocessing.StandardScaler method*), 94  
`get_pendulum()` (*in module d3rlpy.datasets*), 89  
`get_pybullet()` (*in module d3rlpy.datasets*), 89  
`get_type()` (*d3rlpy.augmentation.image.Cutout method*), 96  
`get_type()` (*d3rlpy.augmentation.image.HorizontalFlip method*), 97  
`get_type()` (*d3rlpy.augmentation.image.Intensity method*), 99  
`get_type()` (*d3rlpy.augmentation.image.RandomRotation method*), 98  
`get_type()` (*d3rlpy.augmentation.image.RandomShift method*), 95  
`get_type()` (*d3rlpy.augmentation.image.VerticalFlip method*), 97  
`get_type()` (*d3rlpy.preprocessing.MinMaxScaler method*), 93  
`get_type()` (*d3rlpy.preprocessing.PixelScaler method*), 91  
`get_type()` (*d3rlpy.preprocessing.StandardScaler method*), 94

## H

`horizon` (*d3rlpy.dynamics.mopo.MOPO attribute*), 117  
`HorizontalFlip` (*class in d3rlpy.augmentation.image*), 96

## I

`imitator_learning_rate` (*d3rlpy.algos.BCQ attribute*), 31  
`imitator_learning_rate` (*d3rlpy.algos.BEAR attribute*), 38  
`impl` (*d3rlpy.algos.BC attribute*), 7  
`impl` (*d3rlpy.algos.BCQ attribute*), 33  
`impl` (*d3rlpy.algos.BEAR attribute*), 40  
`impl` (*d3rlpy.algos.CQL attribute*), 47  
`impl` (*d3rlpy.algos.DDPG attribute*), 12  
`impl` (*d3rlpy.algos.DiscreteBC attribute*), 52  
`impl` (*d3rlpy.algos.DiscreteBCQ attribute*), 70  
`impl` (*d3rlpy.algos.DiscreteCQL attribute*), 76  
`impl` (*d3rlpy.algos.DoubleDQN attribute*), 64  
`impl` (*d3rlpy.algos.DQN attribute*), 58  
`impl` (*d3rlpy.algos.SAC attribute*), 25  
`impl` (*d3rlpy.algos.TD3 attribute*), 19  
`impl` (*d3rlpy.dynamics.mopo.MOPO attribute*), 117  
`initial_alpha` (*d3rlpy.algos.BEAR attribute*), 39  
`initial_alpha` (*d3rlpy.algos.CQL attribute*), 46  
`initial_temperature` (*d3rlpy.algos.BEAR attribute*), 39  
`initial_temperature` (*d3rlpy.algos.CQL attribute*), 46  
`initial_temperature` (*d3rlpy.algos.SAC attribute*), 25  
`Intensity` (*class in d3rlpy.augmentation.image*), 98  
`is_action_discrete()` (*d3rlpy.dataset.MDPDataset method*), 83

## L

`lam` (*d3rlpy.algos.BCQ attribute*), 32  
`lam` (*d3rlpy.algos.BEAR attribute*), 39  
`lam` (*d3rlpy.dynamics.mopo.MOPO attribute*), 117  
`latent_size` (*d3rlpy.algos.BCQ attribute*), 32  
`learning_rate` (*d3rlpy.algos.BC attribute*), 6



- learning\_rate (*d3rlpy.algos.DiscreteBC* attribute), 52  
 learning\_rate (*d3rlpy.algos.DiscreteBCQ* attribute), 68  
 learning\_rate (*d3rlpy.algos.DiscreteCQL* attribute), 75  
 learning\_rate (*d3rlpy.algos.DoubleDQN* attribute), 62  
 learning\_rate (*d3rlpy.algos.DQN* attribute), 57  
 learning\_rate (*d3rlpy.dynamics.mopo.MOPO* attribute), 116  
 LinearDecayEpsilonGreedy (class in *d3rlpy.online.explorers*), 112  
 load() (*d3rlpy.dataset.MDPDataset* class method), 83  
 load\_model() (*d3rlpy.algos.BC* method), 8  
 load\_model() (*d3rlpy.algos.BCQ* method), 34  
 load\_model() (*d3rlpy.algos.BEAR* method), 42  
 load\_model() (*d3rlpy.algos.CQL* method), 49  
 load\_model() (*d3rlpy.algos.DDPG* method), 14  
 load\_model() (*d3rlpy.algos.DiscreteBC* method), 54  
 load\_model() (*d3rlpy.algos.DiscreteBCQ* method), 71  
 load\_model() (*d3rlpy.algos.DiscreteCQL* method), 77  
 load\_model() (*d3rlpy.algos.DoubleDQN* method), 65  
 load\_model() (*d3rlpy.algos.DQN* method), 59  
 load\_model() (*d3rlpy.algos.SAC* method), 27  
 load\_model() (*d3rlpy.algos.TD3* method), 20  
 load\_model() (*d3rlpy.dynamics.mopo.MOPO* method), 119
- ## M
- maximum (*d3rlpy.preprocessing.MinMaxScaler* attribute), 92  
 maxlen (*d3rlpy.online.buffers.ReplayBuffer* attribute), 111  
 MDPDataset (class in *d3rlpy.dataset*), 81  
 mean (*d3rlpy.online.explorers.NormalNoise* attribute), 113  
 mean (*d3rlpy.preprocessing.StandardScaler* attribute), 94  
 minimum (*d3rlpy.preprocessing.MinMaxScaler* attribute), 92  
 MinMaxScaler (class in *d3rlpy.preprocessing*), 92  
 mm\_d\_sigma (*d3rlpy.algos.BEAR* attribute), 39  
 MOPO (class in *d3rlpy.dynamics.mopo*), 115
- ## N
- n\_action\_samples (*d3rlpy.algos.BCQ* attribute), 32  
 n\_action\_samples (*d3rlpy.algos.BEAR* attribute), 39  
 n\_action\_samples (*d3rlpy.algos.CQL* attribute), 46  
 n\_augmentations (*d3rlpy.algos.BC* attribute), 7  
 n\_augmentations (*d3rlpy.algos.BCQ* attribute), 33  
 n\_augmentations (*d3rlpy.algos.BEAR* attribute), 40  
 n\_augmentations (*d3rlpy.algos.CQL* attribute), 47  
 n\_augmentations (*d3rlpy.algos.DDPG* attribute), 12  
 n\_augmentations (*d3rlpy.algos.DiscreteBC* attribute), 52  
 n\_augmentations (*d3rlpy.algos.DiscreteBCQ* attribute), 70  
 n\_augmentations (*d3rlpy.algos.DiscreteCQL* attribute), 76  
 n\_augmentations (*d3rlpy.algos.DoubleDQN* attribute), 63  
 n\_augmentations (*d3rlpy.algos.DQN* attribute), 58  
 n\_augmentations (*d3rlpy.algos.SAC* attribute), 25  
 n\_augmentations (*d3rlpy.algos.TD3* attribute), 19  
 n\_critics (*d3rlpy.algos.BCQ* attribute), 31  
 n\_critics (*d3rlpy.algos.BEAR* attribute), 39  
 n\_critics (*d3rlpy.algos.CQL* attribute), 46  
 n\_critics (*d3rlpy.algos.DDPG* attribute), 11  
 n\_critics (*d3rlpy.algos.DiscreteBCQ* attribute), 69  
 n\_critics (*d3rlpy.algos.DiscreteCQL* attribute), 75  
 n\_critics (*d3rlpy.algos.DoubleDQN* attribute), 63  
 n\_critics (*d3rlpy.algos.DQN* attribute), 57  
 n\_critics (*d3rlpy.algos.SAC* attribute), 24  
 n\_critics (*d3rlpy.algos.TD3* attribute), 18  
 n\_ensembles (*d3rlpy.dynamics.mopo.MOPO* attribute), 117  
 n\_epochs (*d3rlpy.algos.BC* attribute), 6  
 n\_epochs (*d3rlpy.algos.BCQ* attribute), 32  
 n\_epochs (*d3rlpy.algos.BEAR* attribute), 40  
 n\_epochs (*d3rlpy.algos.CQL* attribute), 47  
 n\_epochs (*d3rlpy.algos.DDPG* attribute), 12  
 n\_epochs (*d3rlpy.algos.DiscreteBC* attribute), 52  
 n\_epochs (*d3rlpy.algos.DiscreteBCQ* attribute), 69  
 n\_epochs (*d3rlpy.algos.DiscreteCQL* attribute), 75  
 n\_epochs (*d3rlpy.algos.DoubleDQN* attribute), 63  
 n\_epochs (*d3rlpy.algos.DQN* attribute), 57  
 n\_epochs (*d3rlpy.algos.SAC* attribute), 25  
 n\_epochs (*d3rlpy.algos.TD3* attribute), 18  
 n\_epochs (*d3rlpy.dynamics.mopo.MOPO* attribute), 116  
 n\_transitions (*d3rlpy.dynamics.mopo.MOPO* attribute), 117  
 next\_action (*d3rlpy.dataset.Transition* attribute), 86  
 next\_actions (*d3rlpy.dataset.TransitionMiniBatch* attribute), 88  
 next\_observation (*d3rlpy.dataset.Transition* attribute), 87  
 next\_observations (*d3rlpy.dataset.TransitionMiniBatch* attribute), 88  
 next\_reward (*d3rlpy.dataset.Transition* attribute), 87  
 next\_rewards (*d3rlpy.dataset.TransitionMiniBatch* attribute), 88  
 NormalNoise (class in *d3rlpy.online.explorers*), 113

## O

observation (*d3rlpy.dataset.Transition* attribute), 87  
 observation\_shape (*d3rlpy.online.buffers.ReplayBuffer* attribute), 111  
 observations (*d3rlpy.dataset.Episode* attribute), 85  
 observations (*d3rlpy.dataset.MDPDataset* attribute), 84  
 observations (*d3rlpy.dataset.TransitionMiniBatch* attribute), 88  
 observations (*d3rlpy.online.buffers.ReplayBuffer* attribute), 111

## P

PixelScaler (*class in d3rlpy.preprocessing*), 91  
 predict() (*d3rlpy.algos.BC* method), 8  
 predict() (*d3rlpy.algos.BCQ* method), 34  
 predict() (*d3rlpy.algos.BEAR* method), 42  
 predict() (*d3rlpy.algos.CQL* method), 49  
 predict() (*d3rlpy.algos.DDPG* method), 14  
 predict() (*d3rlpy.algos.DiscreteBC* method), 54  
 predict() (*d3rlpy.algos.DiscreteBCQ* method), 71  
 predict() (*d3rlpy.algos.DiscreteCQL* method), 77  
 predict() (*d3rlpy.algos.DoubleDQN* method), 65  
 predict() (*d3rlpy.algos.DQN* method), 59  
 predict() (*d3rlpy.algos.SAC* method), 27  
 predict() (*d3rlpy.algos.TD3* method), 20  
 predict() (*d3rlpy.dynamics.mopo.MOPO* method), 119  
 predict\_value() (*d3rlpy.algos.BC* method), 9  
 predict\_value() (*d3rlpy.algos.BCQ* method), 35  
 predict\_value() (*d3rlpy.algos.BEAR* method), 42  
 predict\_value() (*d3rlpy.algos.CQL* method), 49  
 predict\_value() (*d3rlpy.algos.DDPG* method), 14  
 predict\_value() (*d3rlpy.algos.DiscreteBC* method), 54  
 predict\_value() (*d3rlpy.algos.DiscreteBCQ* method), 72  
 predict\_value() (*d3rlpy.algos.DiscreteCQL* method), 78  
 predict\_value() (*d3rlpy.algos.DoubleDQN* method), 66  
 predict\_value() (*d3rlpy.algos.DQN* method), 60  
 predict\_value() (*d3rlpy.algos.SAC* method), 27  
 predict\_value() (*d3rlpy.algos.TD3* method), 21  
 probability (*d3rlpy.augmentation.image.Cutout* attribute), 96  
 probability (*d3rlpy.augmentation.image.HorizontalFlip* attribute), 96  
 probability (*d3rlpy.augmentation.image.VerticalFlip* attribute), 97

## Q

q\_func\_type (*d3rlpy.algos.BCQ* attribute), 32

q\_func\_type (*d3rlpy.algos.BEAR* attribute), 40  
 q\_func\_type (*d3rlpy.algos.CQL* attribute), 47  
 q\_func\_type (*d3rlpy.algos.DDPG* attribute), 12  
 q\_func\_type (*d3rlpy.algos.DiscreteBCQ* attribute), 69  
 q\_func\_type (*d3rlpy.algos.DiscreteCQL* attribute), 75  
 q\_func\_type (*d3rlpy.algos.DoubleDQN* attribute), 63  
 q\_func\_type (*d3rlpy.algos.DQN* attribute), 57  
 q\_func\_type (*d3rlpy.algos.SAC* attribute), 25  
 q\_func\_type (*d3rlpy.algos.TD3* attribute), 18

## R

RandomRotation (*class in d3rlpy.augmentation.image*), 98  
 RandomShift (*class in d3rlpy.augmentation.image*), 95  
 regularizing\_rate (*d3rlpy.algos.DDPG* attribute), 12  
 regularizing\_rate (*d3rlpy.algos.TD3* attribute), 18  
 ReplayBuffer (*class in d3rlpy.online.buffers*), 111  
 reverse\_transform() (*d3rlpy.preprocessing.MinMaxScaler* method), 93  
 reverse\_transform() (*d3rlpy.preprocessing.PixelScaler* method), 91  
 reverse\_transform() (*d3rlpy.preprocessing.StandardScaler* method), 94  
 reward (*d3rlpy.dataset.Transition* attribute), 87  
 rewards (*d3rlpy.dataset.Episode* attribute), 85  
 rewards (*d3rlpy.dataset.MDPDataset* attribute), 84  
 rewards (*d3rlpy.dataset.TransitionMiniBatch* attribute), 88  
 rewards (*d3rlpy.online.buffers.ReplayBuffer* attribute), 111  
 rl\_start\_epoch (*d3rlpy.algos.BCQ* attribute), 32  
 rl\_start\_epoch (*d3rlpy.algos.BEAR* attribute), 39

## S

SAC (*class in d3rlpy.algos*), 23  
 sample() (*d3rlpy.online.buffers.ReplayBuffer* method), 112  
 sample() (*d3rlpy.online.explorers.LinearDecayEpsilonGreedy* method), 113  
 sample() (*d3rlpy.online.explorers.NormalNoise* method), 113  
 sample\_action() (*d3rlpy.algos.BC* method), 9  
 sample\_action() (*d3rlpy.algos.BCQ* method), 35  
 sample\_action() (*d3rlpy.algos.BEAR* method), 43  
 sample\_action() (*d3rlpy.algos.CQL* method), 50  
 sample\_action() (*d3rlpy.algos.DDPG* method), 15

`sample_action()` (*d3rlpy.algos.DiscreteBC method*), 54  
`sample_action()` (*d3rlpy.algos.DiscreteBCQ method*), 72  
`sample_action()` (*d3rlpy.algos.DiscreteCQL method*), 78  
`sample_action()` (*d3rlpy.algos.DoubleDQN method*), 66  
`sample_action()` (*d3rlpy.algos.DQN method*), 60  
`sample_action()` (*d3rlpy.algos.SAC method*), 28  
`sample_action()` (*d3rlpy.algos.TD3 method*), 21  
`save_model()` (*d3rlpy.algos.BC method*), 9  
`save_model()` (*d3rlpy.algos.BCQ method*), 35  
`save_model()` (*d3rlpy.algos.BEAR method*), 43  
`save_model()` (*d3rlpy.algos.CQL method*), 50  
`save_model()` (*d3rlpy.algos.DDPG method*), 15  
`save_model()` (*d3rlpy.algos.DiscreteBC method*), 55  
`save_model()` (*d3rlpy.algos.DiscreteBCQ method*), 72  
`save_model()` (*d3rlpy.algos.DiscreteCQL method*), 78  
`save_model()` (*d3rlpy.algos.DoubleDQN method*), 66  
`save_model()` (*d3rlpy.algos.DQN method*), 60  
`save_model()` (*d3rlpy.algos.SAC method*), 28  
`save_model()` (*d3rlpy.algos.TD3 method*), 21  
`save_model()` (*d3rlpy.dynamics.mopo.MOPO method*), 119  
`save_policy()` (*d3rlpy.algos.BC method*), 9  
`save_policy()` (*d3rlpy.algos.BCQ method*), 36  
`save_policy()` (*d3rlpy.algos.BEAR method*), 43  
`save_policy()` (*d3rlpy.algos.CQL method*), 50  
`save_policy()` (*d3rlpy.algos.DDPG method*), 15  
`save_policy()` (*d3rlpy.algos.DiscreteBC method*), 55  
`save_policy()` (*d3rlpy.algos.DiscreteBCQ method*), 73  
`save_policy()` (*d3rlpy.algos.DiscreteCQL method*), 79  
`save_policy()` (*d3rlpy.algos.DoubleDQN method*), 66  
`save_policy()` (*d3rlpy.algos.DQN method*), 61  
`save_policy()` (*d3rlpy.algos.SAC method*), 28  
`save_policy()` (*d3rlpy.algos.TD3 method*), 22  
`scale` (*d3rlpy.augmentation.image.Intensity attribute*), 98  
`scaler` (*d3rlpy.algos.BC attribute*), 6  
`scaler` (*d3rlpy.algos.BCQ attribute*), 33  
`scaler` (*d3rlpy.algos.BEAR attribute*), 40  
`scaler` (*d3rlpy.algos.CQL attribute*), 47  
`scaler` (*d3rlpy.algos.DDPG attribute*), 12  
`scaler` (*d3rlpy.algos.DiscreteBC attribute*), 52  
`scaler` (*d3rlpy.algos.DiscreteBCQ attribute*), 70  
`scaler` (*d3rlpy.algos.DiscreteCQL attribute*), 75  
`scaler` (*d3rlpy.algos.DoubleDQN attribute*), 63  
`scaler` (*d3rlpy.algos.DQN attribute*), 57  
`scaler` (*d3rlpy.algos.SAC attribute*), 25  
`scaler` (*d3rlpy.algos.TD3 attribute*), 19  
`scaler` (*d3rlpy.dynamics.mopo.MOPO attribute*), 117  
`set_params()` (*d3rlpy.algos.BC method*), 9  
`set_params()` (*d3rlpy.algos.BCQ method*), 36  
`set_params()` (*d3rlpy.algos.BEAR method*), 43  
`set_params()` (*d3rlpy.algos.CQL method*), 50  
`set_params()` (*d3rlpy.algos.DDPG method*), 15  
`set_params()` (*d3rlpy.algos.DiscreteBC method*), 55  
`set_params()` (*d3rlpy.algos.DiscreteBCQ method*), 73  
`set_params()` (*d3rlpy.algos.DiscreteCQL method*), 79  
`set_params()` (*d3rlpy.algos.DoubleDQN method*), 67  
`set_params()` (*d3rlpy.algos.DQN method*), 61  
`set_params()` (*d3rlpy.algos.SAC method*), 28  
`set_params()` (*d3rlpy.algos.TD3 method*), 22  
`set_params()` (*d3rlpy.dynamics.mopo.MOPO method*), 119  
`share_encoder` (*d3rlpy.algos.BCQ attribute*), 32  
`share_encoder` (*d3rlpy.algos.BEAR attribute*), 39  
`share_encoder` (*d3rlpy.algos.CQL attribute*), 46  
`share_encoder` (*d3rlpy.algos.DDPG attribute*), 11  
`share_encoder` (*d3rlpy.algos.DiscreteBCQ attribute*), 69  
`share_encoder` (*d3rlpy.algos.DoubleDQN attribute*), 63  
`share_encoder` (*d3rlpy.algos.DQN attribute*), 57  
`share_encoder` (*d3rlpy.algos.SAC attribute*), 24  
`share_encoder` (*d3rlpy.algos.TD3 attribute*), 18  
`shift_size` (*d3rlpy.augmentation.image.RandomShift attribute*), 95  
`size()` (*d3rlpy.dataset.Episode method*), 85  
`size()` (*d3rlpy.dataset.MDPDataset method*), 83  
`size()` (*d3rlpy.dataset.TransitionMiniBatch method*), 87  
`size()` (*d3rlpy.online.buffers.ReplayBuffer method*), 112  
`StandardScaler` (*class in d3rlpy.preprocessing*), 93  
`start_epsilon` (*d3rlpy.online.explorers.LinearDecayEpsilonGreedy attribute*), 112  
`std` (*d3rlpy.online.explorers.NormalNoise attribute*), 113  
`std` (*d3rlpy.preprocessing.StandardScaler attribute*), 94

## T

`target_smoothing_clip` (*d3rlpy.algos.TD3 attribute*), 18  
`target_smoothing_sigma` (*d3rlpy.algos.TD3 attribute*), 18  
`target_update_interval` (*d3rlpy.algos.DiscreteBCQ attribute*), 69

target\_update\_interval (d3rlpy.algos.DiscreteCQL attribute), 75  
 target\_update\_interval (d3rlpy.algos.DoubleDQN attribute), 63  
 target\_update\_interval (d3rlpy.algos.DQN attribute), 57  
 tau (d3rlpy.algos.BCQ attribute), 31  
 tau (d3rlpy.algos.BEAR attribute), 38  
 tau (d3rlpy.algos.CQL attribute), 46  
 tau (d3rlpy.algos.DDPG attribute), 11  
 tau (d3rlpy.algos.SAC attribute), 24  
 tau (d3rlpy.algos.TD3 attribute), 18  
 TD3 (class in d3rlpy.algos), 16  
 td\_error\_scorer() (in module d3rlpy.metrics.scorer), 100  
 temp\_learning\_rate (d3rlpy.algos.BEAR attribute), 38  
 temp\_learning\_rate (d3rlpy.algos.CQL attribute), 45  
 temp\_learning\_rate (d3rlpy.algos.SAC attribute), 24  
 terminal (d3rlpy.dataset.Transition attribute), 87  
 terminals (d3rlpy.dataset.MDPDataset attribute), 84  
 terminals (d3rlpy.dataset.TransitionMiniBatch attribute), 88  
 terminals (d3rlpy.online.buffers.ReplayBuffer attribute), 111  
 train() (in module d3rlpy.online.iterators), 114  
 transform() (d3rlpy.augmentation.image.Cutout method), 96  
 transform() (d3rlpy.augmentation.image.HorizontalFlip method), 97  
 transform() (d3rlpy.augmentation.image.Intensity method), 99  
 transform() (d3rlpy.augmentation.image.RandomRotation method), 98  
 transform() (d3rlpy.augmentation.image.RandomShift method), 96  
 transform() (d3rlpy.augmentation.image.VerticalFlip method), 97  
 transform() (d3rlpy.preprocessing.MinMaxScaler method), 93  
 transform() (d3rlpy.preprocessing.PixelScaler method), 91  
 transform() (d3rlpy.preprocessing.StandardScaler method), 94  
 Transition (class in d3rlpy.dataset), 86  
 TransitionMiniBatch (class in d3rlpy.dataset), 87  
 transitions (d3rlpy.dataset.Episode attribute), 85  
 transitions (d3rlpy.dataset.TransitionMiniBatch attribute), 88  
 update() (d3rlpy.algos.BC method), 9  
 update() (d3rlpy.algos.BCQ method), 36  
 update() (d3rlpy.algos.BEAR method), 43  
 update() (d3rlpy.algos.CQL method), 50  
 update() (d3rlpy.algos.DDPG method), 16  
 update() (d3rlpy.algos.DiscreteBC method), 55  
 update() (d3rlpy.algos.DiscreteBCQ method), 73  
 update() (d3rlpy.algos.DiscreteCQL method), 79  
 update() (d3rlpy.algos.DoubleDQN method), 67  
 update() (d3rlpy.algos.DQN method), 61  
 update() (d3rlpy.algos.SAC method), 29  
 update() (d3rlpy.algos.TD3 method), 22  
 update() (d3rlpy.dynamics.mopo.MOPO method), 120  
 update\_actor\_interval (d3rlpy.algos.BCQ attribute), 32  
 update\_actor\_interval (d3rlpy.algos.BEAR attribute), 39  
 update\_actor\_interval (d3rlpy.algos.CQL attribute), 46  
 update\_actor\_interval (d3rlpy.algos.SAC attribute), 25  
 update\_actor\_interval (d3rlpy.algos.TD3 attribute), 18  
 use\_batch\_norm (d3rlpy.algos.BC attribute), 6  
 use\_batch\_norm (d3rlpy.algos.BCQ attribute), 32  
 use\_batch\_norm (d3rlpy.algos.BEAR attribute), 39  
 use\_batch\_norm (d3rlpy.algos.CQL attribute), 46  
 use\_batch\_norm (d3rlpy.algos.DDPG attribute), 12  
 use\_batch\_norm (d3rlpy.algos.DiscreteBC attribute), 52  
 use\_batch\_norm (d3rlpy.algos.DiscreteBCQ attribute), 69  
 use\_batch\_norm (d3rlpy.algos.DiscreteCQL attribute), 75  
 use\_batch\_norm (d3rlpy.algos.DoubleDQN attribute), 63  
 use\_batch\_norm (d3rlpy.algos.DQN attribute), 57  
 use\_batch\_norm (d3rlpy.algos.SAC attribute), 25  
 use\_batch\_norm (d3rlpy.algos.TD3 attribute), 18  
 use\_batch\_norm (d3rlpy.dynamics.mopo.MOPO attribute), 117  
 use\_gpu (d3rlpy.algos.BC attribute), 6  
 use\_gpu (d3rlpy.algos.BCQ attribute), 32  
 use\_gpu (d3rlpy.algos.BEAR attribute), 40  
 use\_gpu (d3rlpy.algos.CQL attribute), 47  
 use\_gpu (d3rlpy.algos.DDPG attribute), 12  
 use\_gpu (d3rlpy.algos.DiscreteBC attribute), 52  
 use\_gpu (d3rlpy.algos.DiscreteBCQ attribute), 69  
 use\_gpu (d3rlpy.algos.DiscreteCQL attribute), 75  
 use\_gpu (d3rlpy.algos.DoubleDQN attribute), 63  
 use\_gpu (d3rlpy.algos.DQN attribute), 57  
 use\_gpu (d3rlpy.algos.SAC attribute), 25  
 use\_gpu (d3rlpy.algos.TD3 attribute), 18  
 use\_gpu (d3rlpy.dynamics.mopo.MOPO attribute), 117

## U

## V

`value_estimation_std_scorer()` (in module *d3rlpy.metrics.scorer*), [101](#)

`VerticalFlip` (class in *d3rlpy.augmentation.image*), [97](#)

## W

`weight_decay` (*d3rlpy.dynamics.mopo.MOPO* attribute), [116](#)