Stable Baselines3 Documentation

Release 1.8.0

Stable Baselines3 Contributors

Apr 08, 2023

USER GUIDE

1 Installation **3**
- 1.1 Prerequisites . . . 3
- 1.2 Bleeding-edge version . . . 3
- 1.3 Development version . . . 3

2 RL Algorithms **5**

3 Examples **7**
- 3.1 TQC . . . 7
- 3.2 QR-DQN . . . 7
- 3.3 MaskablePPO . . . 7
- 3.4 TRPO . . . 8
- 3.5 ARS . . . 8
- 3.6 RecurrentPPO . . . 8

4 ARS **9**
- 4.1 Notes . . . 9
- 4.2 Can I use? . . . 9
- 4.3 Example . . . 10
- 4.4 Results . . . 10
- 4.5 Parameters . . . 11
- 4.6 ARS Policies . . . 15

5 Maskable PPO **17**
- 5.1 Notes . . . 17
- 5.2 Can I use? . . . 17
- 5.3 Example . . . 18
- 5.4 Results . . . 19
- 5.5 Parameters . . . 24
- 5.6 MaskablePPO Policies . . . 29

6 Recurrent PPO **35**
- 6.1 Notes . . . 35
- 6.2 Can I use? . . . 35
- 6.3 Example . . . 36
- 6.4 Results . . . 36
- 6.5 Parameters . . . 37
- 6.6 RecurrentPPO Policies . . . 42

7 QR-DQN **49**
- 7.1 Notes . . . 49

7.2 Can I use? . . . 49
7.3 Example . . . 50
7.4 Results . . . 50
7.5 Parameters . . . 51
7.6 QR-DQN Policies . . . 56

8 TQC **59**
8.1 Notes . . . 59
8.2 Can I use? . . . 59
8.3 Example . . . 60
8.4 Results . . . 60
8.5 Comments . . . 61
8.6 Parameters . . . 61
8.7 TQC Policies . . . 67

9 TRPO **71**
9.1 Notes . . . 71
9.2 Can I use? . . . 71
9.3 Example . . . 72
9.4 Results . . . 72
9.5 Parameters . . . 73
9.6 TRPO Policies . . . 78

10 Utils **83**

11 Gym Wrappers **85**
11.1 TimeFeatureWrapper . . . 85

12 Changelog **87**
12.1 Release 1.8.0 (2023-04-07) . . . 87
12.2 Release 1.7.0 (2023-01-10) . . . 88
12.3 Release 1.6.2 (2022-10-10) . . . 89
12.4 Release 1.6.1 (2022-09-29) . . . 89
12.5 Release 1.6.0 (2022-07-11) . . . 90
12.6 Release 1.5.0 (2022-03-25) . . . 91
12.7 Release 1.4.0 (2022-01-19) . . . 91
12.8 Release 1.3.0 (2021-10-23) . . . 92
12.9 Release 1.2.0 (2021-09-08) . . . 93
12.10 Release 1.1.0 (2021-07-01) . . . 93
12.11 Release 1.0 (2021-03-17) . . . 94
12.12 Pre-Release 0.11.1 (2021-02-27) . . . 94
12.13 Pre-Release 0.11.0 (2021-02-27) . . . 95
12.14 Pre-Release 0.10.0 (2020-10-28) . . . 95
12.15 Maintainers . . . 96
12.16 Contributors: . . . 96

13 Citing Stable Baselines3 **97**

14 Contributing **99**

15 Indices and tables **101**

Python Module Index **103**

Index **105**

Contrib package for Stable Baselines3 (SB3) - Experimental code.

Github repository: https://github.com/Stable-Baselines-Team/stable-baselines3-contrib

SB3 repository: https://github.com/DLR-RM/stable-baselines3

RL Baselines3 Zoo (collection of pre-trained agents): https://github.com/DLR-RM/rl-baselines3-zoo

RL Baselines3 Zoo also offers a simple interface to train, evaluate agents and do hyperparameter tuning.

CHAPTER

ONE

INSTALLATION

1.1 Prerequisites

Please read Stable-Baselines3 installation guide first.

1.1.1 Stable Release

To install Stable Baselines3 contrib with pip, execute:

```
pip install sb3-contrib
```

1.2 Bleeding-edge version

```
pip install git+https://github.com/Stable-Baselines-Team/stable-baselines3-contrib/
```

1.3 Development version

To contribute to Stable-Baselines3, with support for running tests and building the documentation.

```
git clone https://github.com/Stable-Baselines-Team/stable-baselines3-contrib/ && cd␣
↪stable-baselines3-contrib
pip install -e .
```

CHAPTER

TWO

RL ALGORITHMS

This table displays the rl algorithms that are implemented in the Stable Baselines3 contrib project, along with some useful characteristics: support for discrete/continuous actions, multiprocessing.

Name	`Box`	`Discrete`	`MultiDiscrete`	`MultiBinary`	Multi Processing
ARS	✓				✓
MaskablePPO		✓	✓	✓	✓
QR-DQN		✓			✓
RecurrentPPO	✓	✓	✓	✓	✓
TQC	✓				✓
TRPO	✓	✓	✓	✓	✓

Note: `Tuple` observation spaces are not supported by any environment, however, single-level `Dict` spaces are

Actions `gym.spaces`:

- `Box`: A N-dimensional box that contains every point in the action space.
- `Discrete`: A list of possible actions, where each timestep only one of the actions can be used.
- `MultiDiscrete`: A list of possible actions, where each timestep only one action of each discrete set can be used.
- `MultiBinary`: A list of possible actions, where each timestep any of the actions can be used in any combination.

CHAPTER

THREE

EXAMPLES

3.1 TQC

Train a Truncated Quantile Critics (TQC) agent on the Pendulum environment.

```
from sb3_contrib import TQC

model = TQC("MlpPolicy", "Pendulum-v1", top_quantiles_to_drop_per_net=2, verbose=1)
model.learn(total_timesteps=10_000, log_interval=4)
model.save("tqc_pendulum")
```

3.2 QR-DQN

Train a Quantile Regression DQN (QR-DQN) agent on the CartPole environment.

```
from sb3_contrib import QRDQN

policy_kwargs = dict(n_quantiles=50)
model = QRDQN("MlpPolicy", "CartPole-v1", policy_kwargs=policy_kwargs, verbose=1)
model.learn(total_timesteps=10_000, log_interval=4)
model.save("qrdqn_cartpole")
```

3.3 MaskablePPO

Train a PPO with invalid action masking agent on a toy environment.

```
from sb3_contrib import MaskablePPO
from sb3_contrib.common.envs import InvalidActionEnvDiscrete

env = InvalidActionEnvDiscrete(dim=80, n_invalid_actions=60)
model = MaskablePPO("MlpPolicy", env, verbose=1)
model.learn(5000)
model.save("maskable_toy_env")
```

3.4 TRPO

Train a Trust Region Policy Optimization (TRPO) agent on the Pendulum environment.

```
from sb3_contrib import TRPO

model = TRPO("MlpPolicy", "Pendulum-v1", gamma=0.9, verbose=1)
model.learn(total_timesteps=100_000, log_interval=4)
model.save("trpo_pendulum")
```

3.5 ARS

Train an agent using Augmented Random Search (ARS) agent on the Pendulum environment

```
from sb3_contrib import ARS

model = ARS("LinearPolicy", "Pendulum-v1", verbose=1)
model.learn(total_timesteps=10000, log_interval=4)
model.save("ars_pendulum")
```

3.6 RecurrentPPO

Train a PPO agent with a recurrent policy on the CartPole environment.

Note: It is particularly important to pass the `lstm_states` and `episode_start` argument to the `predict()` method, so the cell and hidden states of the LSTM are correctly updated.

```
import numpy as np

from sb3_contrib import RecurrentPPO

model = RecurrentPPO("MlpLstmPolicy", "CartPole-v1", verbose=1)
model.learn(5000)

env = model.get_env()
obs = env.reset()
# cell and hidden state of the LSTM
lstm_states = None
num_envs = 1
# Episode start signals are used to reset the lstm states
episode_starts = np.ones((num_envs,), dtype=bool)
while True:
    action, lstm_states = model.predict(obs, state=lstm_states, episode_start=episode_
↪starts, deterministic=True)
    obs, rewards, dones, info = env.step(action)
    episode_starts = dones
    env.render()
```

CHAPTER

FOUR

ARS

Augmented Random Search (ARS) is a simple reinforcement algorithm that uses a direct random search over policy parameters. It can be surprisingly effective compared to more sophisticated algorithms. In the original paper the authors showed that linear policies trained with ARS were competitive with deep reinforcement learning for the MuJuCo locomotion tasks.

SB3s implementation allows for linear policies without bias or squashing function, it also allows for training MLP policies, which include linear policies with bias and squashing functions as a special case.

Normally one wants to train ARS with several seeds to properly evaluate.

Warning: ARS multi-processing is different from the classic Stable-Baselines3 multi-processing: it runs n environments in parallel but asynchronously. This asynchronous multi-processing is considered experimental and does not fully support callbacks: the `on_step()` event is called artificially after the evaluation episodes are over.

Available Policies

LinearPolicy	alias of `ARSLinearPolicy`
MlpPolicy	alias of `ARSPolicy`

4.1 Notes

- Original paper: https://arxiv.org/abs/1803.07055
- Original Implementation: https://github.com/modestyachts/ARS

4.2 Can I use?

- Recurrent policies:
- Multi processing: ✓ (cf. example)
- Gym spaces:

Space	Action	Observation
Discrete	✓	✓
Box	✓	✓
MultiDiscrete		✓
MultiBinary		✓
Dict		

4.3 Example

```
from sb3_contrib import ARS

# Policy can be LinearPolicy or MlpPolicy
model = ARS("LinearPolicy", "Pendulum-v1", verbose=1)
model.learn(total_timesteps=10000, log_interval=4)
model.save("ars_pendulum")
```

With experimental asynchronous multi-processing:

```
from sb3_contrib import ARS
from sb3_contrib.common.vec_env import AsyncEval

from stable_baselines3.common.env_util import make_vec_env

env_id = "CartPole-v1"
n_envs = 2

model = ARS("LinearPolicy", env_id, n_delta=2, n_top=1, verbose=1)
# Create env for asynchronous evaluation (run in different processes)
async_eval = AsyncEval([lambda: make_vec_env(env_id) for _ in range(n_envs)], model.
↪policy)

model.learn(total_timesteps=200_000, log_interval=4, async_eval=async_eval)
```

4.4 Results

Replicating results from the original paper, which used the Mujoco benchmarks. Same parameters from the original paper, using 8 seeds.

Environments	ARS
HalfCheetah	4398 +/- 320
Swimmer	241 +/- 51
Hopper	3320 +/- 120

4.4.1 How to replicate the results?

Clone RL-Zoo and checkout the branch `feat/ars`

```
git clone https://github.com/DLR-RM/rl-baselines3-zoo
cd rl-baselines3-zoo/
git checkout feat/ars
```

Run the benchmark. The following code snippet trains 8 seeds in parallel

```
for ENV_ID in Swimmer-v3 HalfCheetah-v3 Hopper-v3
do
    for SEED_NUM in {1..8}
    do
        SEED=$RANDOM
        python train.py --algo ars --env $ENV_ID --eval-episodes 10 --eval-freq 10000 -n␣
↪20000000 --seed  $SEED &
        sleep 1
    done
    wait
done
```

Plot the results:

```
python scripts/all_plots.py -a ars -e HalfCheetah Swimmer Hopper -f logs/ -o logs/ars_
↪results -max 20000000
python scripts/plot_from_file.py -i logs/ars_results.pkl -l ARS
```

4.5 Parameters

class sb3_contrib.ars.**ARS**(*policy*, *env*, *n_delta=8*, *n_top=None*, *learning_rate=0.02*, *delta_std=0.05*, *zero_policy=True*, *alive_bonus_offset=0*, *n_eval_episodes=1*, *policy_kwargs=None*, *stats_window_size=100*, *tensorboard_log=None*, *seed=None*, *verbose=0*, *device='cpu'*, *_init_setup_model=True*)

Augmented Random Search: https://arxiv.org/abs/1803.07055

Original implementation: https://github.com/modestyachts/ARS C++/Cuda Implementation: https://github.com/google-research/tiny-differentiable-simulator/ 150 LOC Numpy Implementation: https://github.com/alexis-jacq/numpy_ARS/blob/master/asr.py

Parameters

- **policy** (Union[str, Type[ARSPolicy]]) – The policy to train, can be an instance of ARSPolicy, or a string from ["LinearPolicy", "MlpPolicy"]
- **env** (Union[Env, VecEnv, str]) – The environment to train on, may be a string if registered with gym
- **n_delta** (int) – How many random perturbations of the policy to try at each update step.
- **n_top** (Optional[int]) – How many of the top delta to use in each update step. Default is n_delta
- **learning_rate** (Union[float, Callable[[float], float]]) – Float or schedule for the step size

- **delta_std** (Union[float, Callable[[float], float]]) – Float or schedule for the exploration noise
- **zero_policy** (bool) – Boolean determining if the passed policy should have it's weights zeroed before training.
- **alive_bonus_offset** (float) – Constant added to the reward at each step, used to cancel out alive bonuses.
- **n_eval_episodes** (int) – Number of episodes to evaluate each candidate.
- **policy_kwargs** (Optional[Dict[str, Any]]) – Keyword arguments to pass to the policy on creation
- **stats_window_size** (int) – Window size for the rollout logging, specifying the number of episodes to average the reported success rate, mean episode length, and mean reward over
- **tensorboard_log** (Optional[str]) – String with the directory to put tensorboard logs:
- **seed** (Optional[int]) – Random seed for the training
- **verbose** (int) – Verbosity level: 0 no output, 1 info, 2 debug
- **device** (Union[device, str]) – Torch device to use for training, defaults to "cpu"
- **_init_setup_model** (bool) – Whether or not to build the network at the creation of the instance

evaluate_candidates(*candidate_weights*, *callback*, *async_eval*)

Evaluate each candidate.

Parameters

- **candidate_weights** (Tensor) – The candidate weights to be evaluated.
- **callback** (BaseCallback) – Callback that will be called at each step (or after evaluation in the multiprocess version)
- **async_eval** (Optional[AsyncEval]) – The object for asynchronous evaluation of candidates.

Return type
Tensor

Returns
The episodic return for each candidate.

get_env()

Returns the current environment (can be None if not defined).

Return type
Optional[VecEnv]

Returns
The current environment

get_parameters()

Return the parameters of the agent. This includes parameters from different networks, e.g. critics (value functions) and policies (pi functions).

Return type
Dict[str, Dict]

Returns
Mapping of from names of the objects to PyTorch state-dicts.

`get_vec_normalize_env()`

Return the `VecNormalize` wrapper of the training env if it exists.

Return type
`Optional[VecNormalize]`

Returns
The `VecNormalize` env.

`learn(total_timesteps, callback=None, log_interval=1, tb_log_name='ARS', reset_num_timesteps=True, async_eval=None, progress_bar=False)`

Return a trained model.

Parameters

- **total_timesteps** (`int`) – The total number of samples (env steps) to train on
- **callback** (`Union[None, Callable, List[BaseCallback], BaseCallback]`) – callback(s) called at every step with state of the algorithm.
- **log_interval** (`int`) – The number of timesteps before logging.
- **tb_log_name** (`str`) – the name of the run for TensorBoard logging
- **reset_num_timesteps** (`bool`) – whether or not to reset the current timestep number (used in logging)
- **async_eval** (`Optional[AsyncEval]`) – The object for asynchronous evaluation of candidates.
- **progress_bar** (`bool`) – Display a progress bar using tqdm and rich.

Return type
`TypeVar(SelfARS, bound= ARS)`

Returns
the trained model

`classmethod load(path, env=None, device='auto', custom_objects=None, print_system_info=False, force_reset=True, **kwargs)`

Load the model from a zip-file. Warning: `load` re-creates the model from scratch, it does not update it in-place! For an in-place load use `set_parameters` instead.

Parameters

- **path** (`Union[str, Path, BufferedIOBase]`) – path to the file (or a file-like) where to load the agent from
- **env** (`Union[Env, VecEnv, None]`) – the new environment to run the loaded model on (can be None if you only need prediction from a trained model) has priority over any saved environment
- **device** (`Union[device, str]`) – Device on which the code should run.
- **custom_objects** (`Optional[Dict[str, Any]]`) – Dictionary of objects to replace upon loading. If a variable is present in this dictionary as a key, it will not be deserialized and the corresponding item will be used instead. Similar to custom_objects in `keras.models.load_model`. Useful when you have an object in file that can not be deserialized.
- **print_system_info** (`bool`) – Whether to print system info from the saved model and the current system info (useful to debug loading issues)
- **force_reset** (`bool`) – Force call to `reset()` before training to avoid unexpected behavior. See https://github.com/DLR-RM/stable-baselines3/issues/597

- **kwargs** – extra arguments to change the model when loading

Return type
TypeVar(SelfBaseAlgorithm, bound= BaseAlgorithm)

Returns
new model instance with loaded parameters

property logger: Logger
Getter for the logger object.

predict(*observation*, *state=None*, *episode_start=None*, *deterministic=False*)
Get the policy action from an observation (and optional hidden state). Includes sugar-coating to handle different observations (e.g. normalizing images).

Parameters

- **observation** (Union[ndarray, Dict[str, ndarray]]) – the input observation
- **state** (Optional[Tuple[ndarray, ...]]) – The last hidden states (can be None, used in recurrent policies)
- **episode_start** (Optional[ndarray]) – The last masks (can be None, used in recurrent policies) this correspond to beginning of episodes, where the hidden states of the RNN must be reset.
- **deterministic** (bool) – Whether or not to return deterministic actions.

Return type
Tuple[ndarray, Optional[Tuple[ndarray, ...]]]

Returns
the model's action and the next hidden state (used in recurrent policies)

save(*path*, *exclude=None*, *include=None*)
Save all the attributes of the object and the model parameters in a zip-file.

Parameters

- **path** (Union[str, Path, BufferedIOBase]) – path to the file where the rl agent should be saved
- **exclude** (Optional[Iterable[str]]) – name of parameters that should be excluded in addition to the default ones
- **include** (Optional[Iterable[str]]) – name of parameters that might be excluded but should be included anyway

Return type
None

set_env(*env*, *force_reset=True*)
Checks the validity of the environment, and if it is coherent, set it as the current environment. Furthermore wrap any non vectorized env into a vectorized checked parameters: - observation_space - action_space

Parameters

- **env** (Union[Env, VecEnv]) – The environment for learning a policy
- **force_reset** (bool) – Force call to `reset()` before training to avoid unexpected behavior. See issue https://github.com/DLR-RM/stable-baselines3/issues/597

Return type
None

set_logger(*logger*)

Setter for for logger object. :rtype: None

> **Warning:** When passing a custom logger object, this will overwrite tensorboard_log and verbose settings passed to the constructor.

set_parameters(*load_path_or_dict*, *exact_match=True*, *device='auto'*)

Load parameters from a given zip-file or a nested dictionary containing parameters for different modules (see get_parameters).

Parameters

- **load_path_or_iter** – Location of the saved data (path or file-like, see save), or a nested dictionary containing nn.Module parameters used by the policy. The dictionary maps object names to a state-dictionary returned by torch.nn.Module.state_dict().
- **exact_match** (bool) – If True, the given parameters should include parameters for each module and each of their parameters, otherwise raises an Exception. If set to False, this can be used to update only specific parameters.
- **device** (Union[device, str]) – Device on which the code should run.

Return type
None

set_random_seed(*seed=None*)

Set the seed of the pseudo-random generators (python, numpy, pytorch, gym, action_space)

Parameters
seed (Optional[int]) –

Return type
None

4.6 ARS Policies

class sb3_contrib.ars.policies.**ARSPolicy**(*observation_space*, *action_space*, *net_arch=None*, *activation_fn=<class 'torch.nn.modules.activation.ReLU'>*, *with_bias=True*, *squash_output=True*)

Policy network for ARS.

Parameters

- **observation_space** (Space) – The observation space of the environment
- **action_space** (Space) – The action space of the environment
- **net_arch** (Optional[List[int]]) – Network architecture, defaults to a 2 layers MLP with 64 hidden nodes.
- **activation_fn** (Type[Module]) – Activation function
- **with_bias** (bool) – If set to False, the layers will not learn an additive bias
- **squash_output** (bool) – For continuous actions, whether the output is squashed or not using a tanh() function. If not squashed with tanh the output will instead be clipped.

forward(*obs*)

Defines the computation performed at every call.

Should be overridden by all subclasses. :rtype: Tensor

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

sb3_contrib.ars.**LinearPolicy**

alias of ARSLinearPolicy

sb3_contrib.ars.**MlpPolicy**

alias of ARSPolicy

CHAPTER

FIVE

MASKABLE PPO

Implementation of invalid action masking for the Proximal Policy Optimization (PPO) algorithm. Other than adding support for action masking, the behavior is the same as in SB3's core PPO algorithm.

Available Policies

MlpPolicy	alias of `MaskableActorCriticPolicy`
CnnPolicy	alias of `MaskableActorCriticCnnPolicy`
MultiInputPolicy	alias of `MaskableMultiInputActorCriticPolicy`

5.1 Notes

- Paper: https://arxiv.org/abs/2006.14171
- Blog post: https://costa.sh/blog-a-closer-look-at-invalid-action-masking-in-policy-gradient-algorithms.html
- Additional Blog post: https://boring-guy.sh/posts/masking-rl/

5.2 Can I use?

- Recurrent policies:
- Multi processing: ✓
- Gym spaces:

Space	Action	Observation
Discrete	✓	✓
Box		✓
MultiDiscrete	✓	✓
MultiBinary	✓	✓
Dict		✓

5.3 Example

Train a PPO agent on `InvalidActionEnvDiscrete`. `InvalidActionEnvDiscrete` has a `action_masks` method that returns the invalid action mask (`True` if the action is valid, `False` otherwise).

```
from sb3_contrib import MaskablePPO
from sb3_contrib.common.envs import InvalidActionEnvDiscrete
from sb3_contrib.common.maskable.evaluation import evaluate_policy
from sb3_contrib.common.maskable.utils import get_action_masks


env = InvalidActionEnvDiscrete(dim=80, n_invalid_actions=60)
model = MaskablePPO("MlpPolicy", env, gamma=0.4, seed=32, verbose=1)
model.learn(5000)

evaluate_policy(model, env, n_eval_episodes=20, reward_threshold=90, warn=False)

model.save("ppo_mask")
del model # remove to demonstrate saving and loading

model = MaskablePPO.load("ppo_mask")

obs = env.reset()
while True:
    # Retrieve current action mask
    action_masks = get_action_masks(env)
    action, _states = model.predict(obs, action_masks=action_masks)
    obs, rewards, dones, info = env.step(action)
    env.render()
```

If the environment implements the invalid action mask but using a different name, you can use the `ActionMasker` to specify the name (see PR #25):

Note: If you are using a custom environment and you want to debug it with `check_env`, it will execute the method `step` passing a random action to it (using `action_space.sample()`), without taking into account the invalid actions mask (see issue #145).

```
import gym
import numpy as np

from sb3_contrib.common.maskable.policies import MaskableActorCriticPolicy
from sb3_contrib.common.wrappers import ActionMasker
from sb3_contrib.ppo_mask import MaskablePPO


def mask_fn(env: gym.Env) -> np.ndarray:
    # Do whatever you'd like in this function to return the action mask
    # for the current env. In this example, we assume the env has a
    # helpful method we can rely on.
    return env.valid_action_mask()
```

(continues on next page)

(continued from previous page)

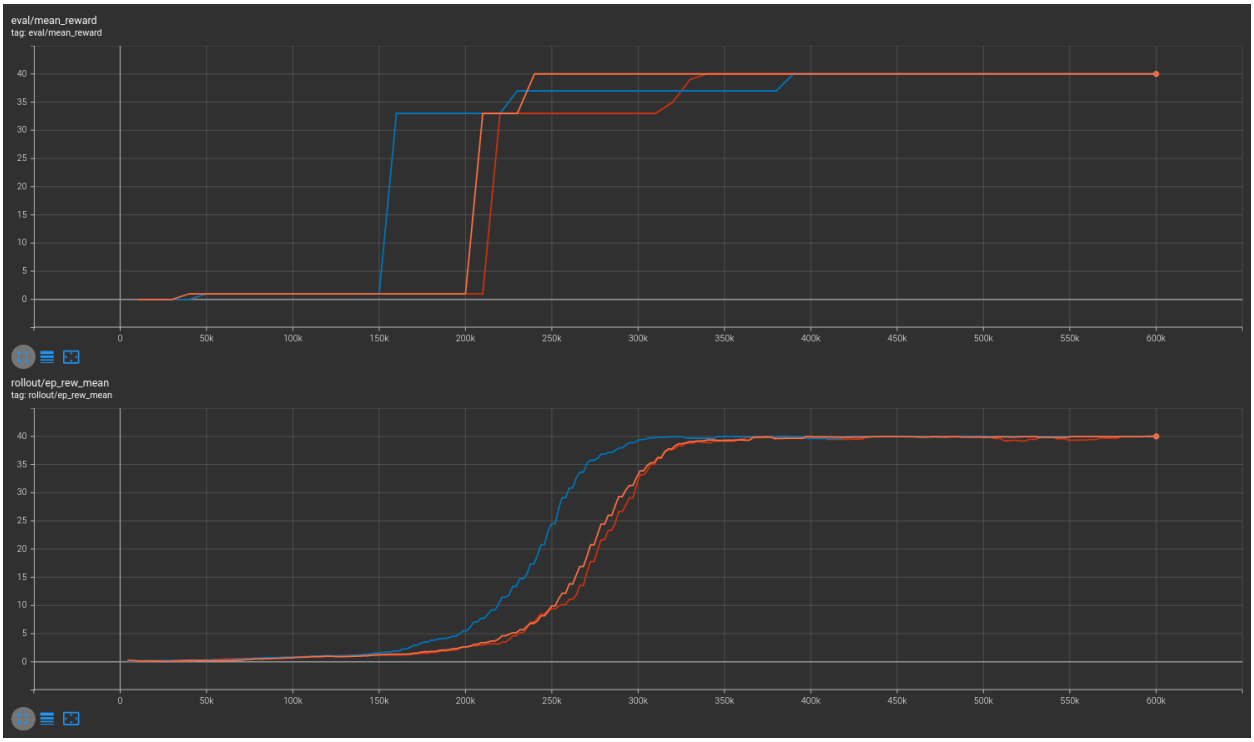
```
env = ...  # Initialize env
env = ActionMasker(env, mask_fn)  # Wrap to enable masking

# MaskablePPO behaves the same as SB3's PPO unless the env is wrapped
# with ActionMasker. If the wrapper is detected, the masks are automatically
# retrieved and used when learning. Note that MaskablePPO does not accept
# a new action_mask_fn kwarg, as it did in an earlier draft.
model = MaskablePPO(MaskableActorCriticPolicy, env, verbose=1)
model.learn()

# Note that use of masks is manual and optional outside of learning,
# so masking can be "removed" at testing time
model.predict(observation, action_masks=valid_action_array)
```

5.4 Results

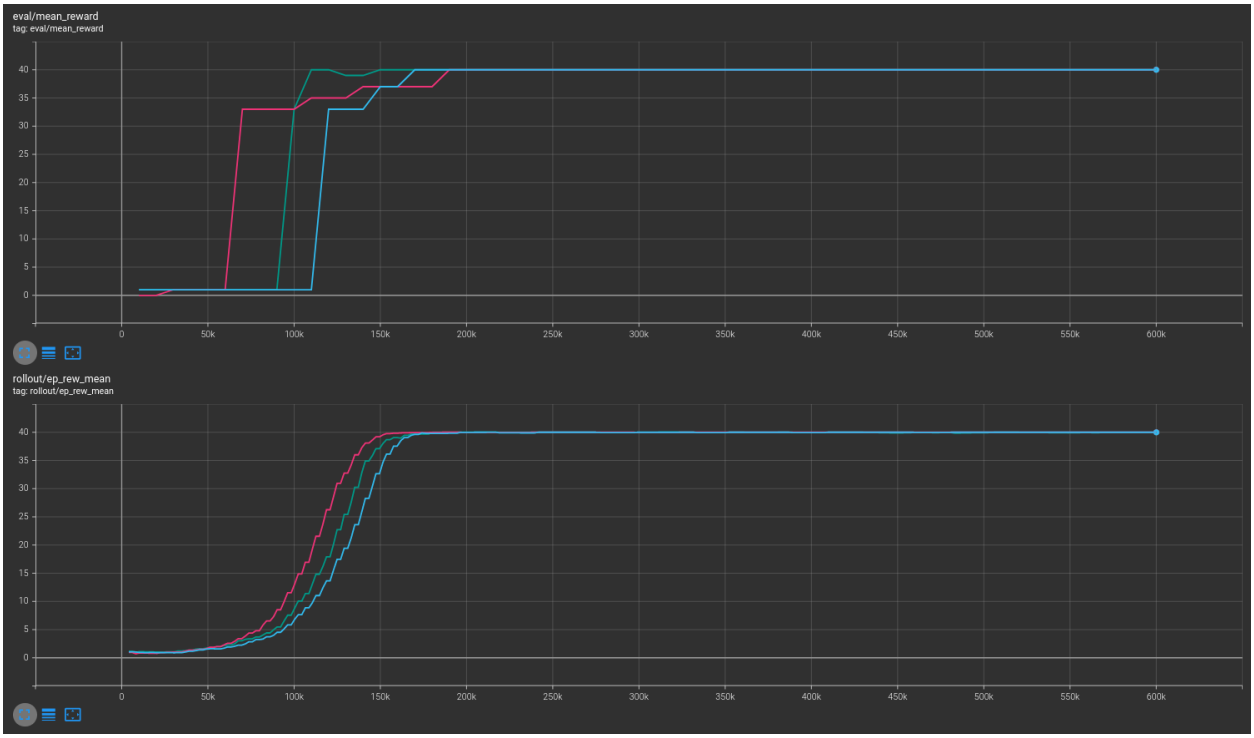
Results are shown for two MicroRTS benchmarks: MicrortsMining4x4F9-v0 (600K steps) and MicrortsMining10x10F9-v0 (1.5M steps). For each, models were trained with and without masking, using 3 seeds.

5.4.1 4x4

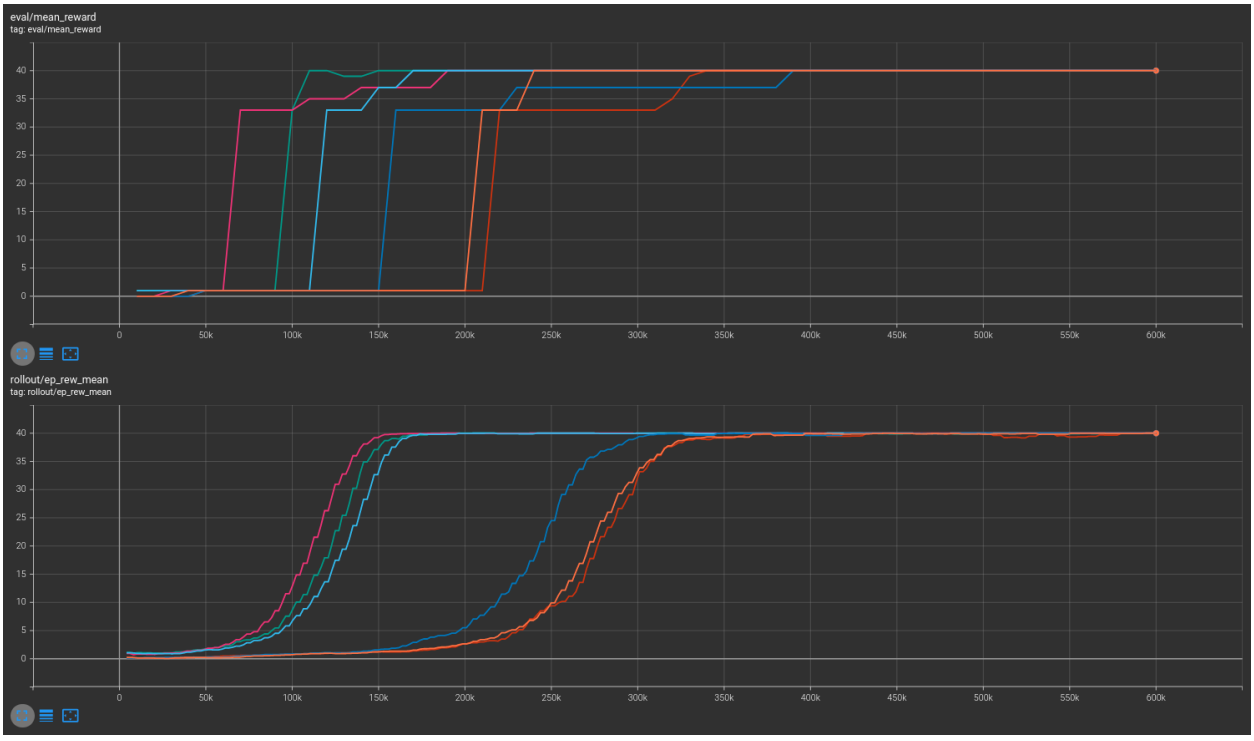
No masking



With masking

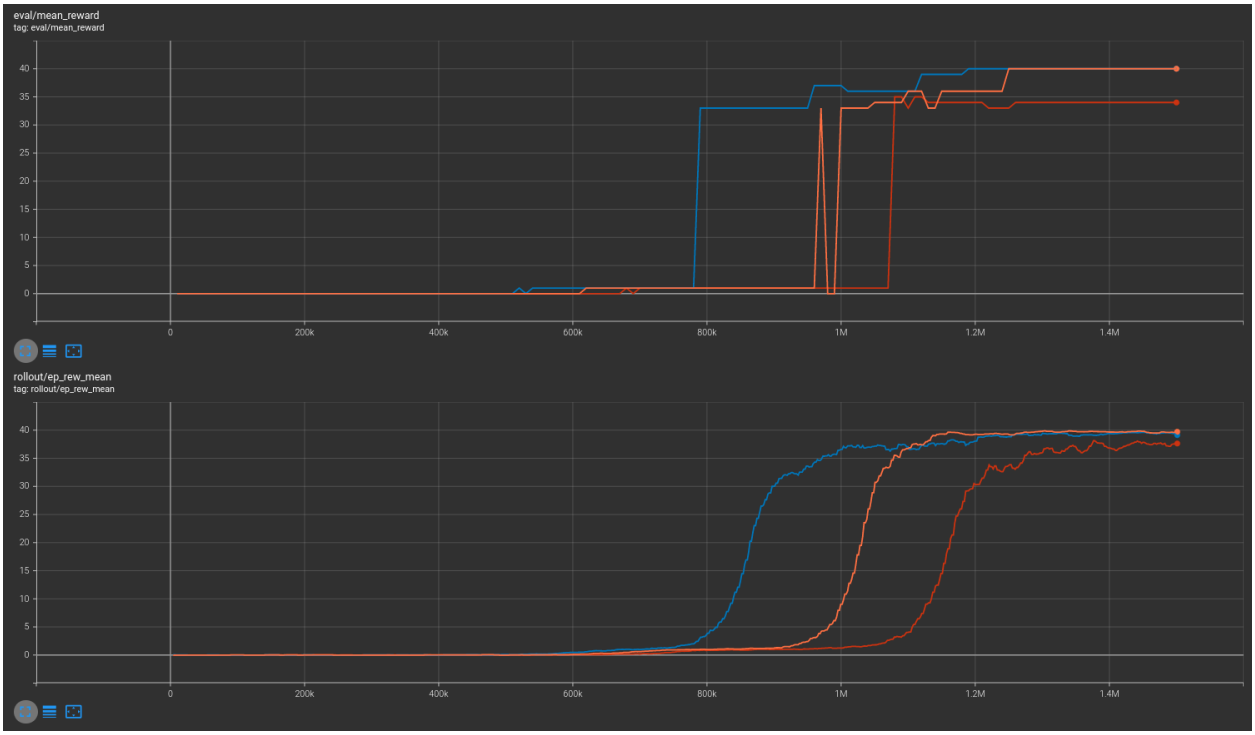


Combined

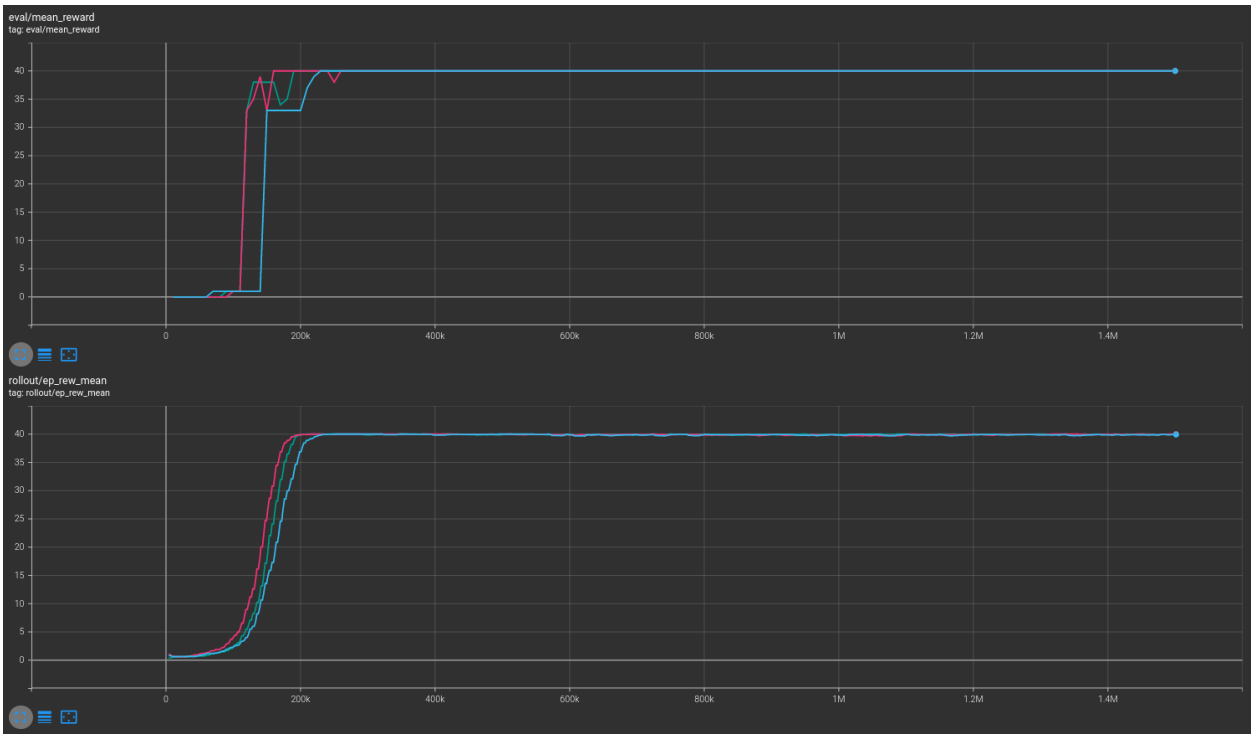


5.4.2 10x10

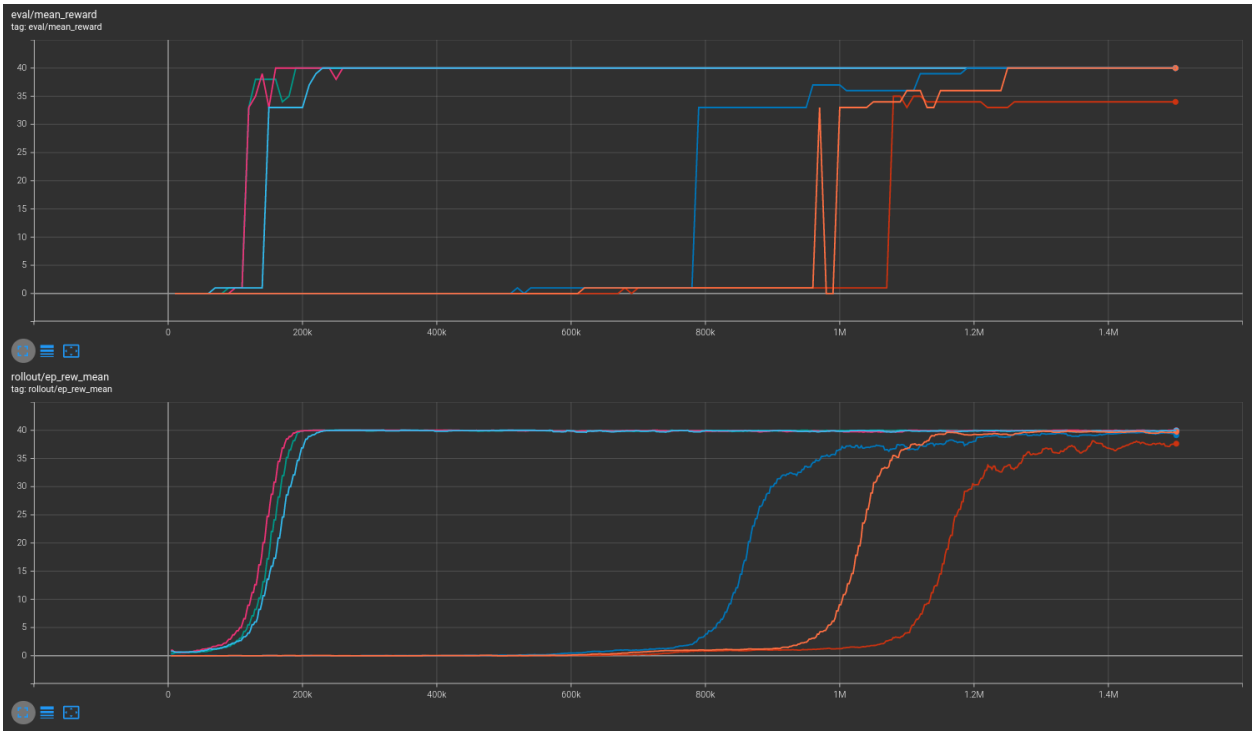
No masking



With masking



Combined



More information may be found in the associated PR.

5.4.3 How to replicate the results?

Clone the repo for the experiment:

```
git clone git@github.com:kronion/microrts-ppo-comparison.git
cd microrts-ppo-comparison
```

Install dependencies:

```
# Install MicroRTS:
rm -fR ~/microrts && mkdir ~/microrts && \
 wget -O ~/microrts/microrts.zip http://microrts.s3.amazonaws.com/microrts/artifacts/
↪202004222224.microrts.zip && \
 unzip ~/microrts/microrts.zip -d ~/microrts/

# You may want to make a venv before installing packages
pip install -r requirements.txt
```

Train several times with various seeds, with and without masking:

```
# python sb/train_ppo.py [output dir] [MicroRTS map size] [--mask] [--seed int]

# 4x4 unmasked
python sb3/train_ppo.py zoo 4 --seed 42
python sb3/train_ppo.py zoo 4 --seed 43
python sb3/train_ppo.py zoo 4 --seed 44

# 4x4 masked
python sb3/train_ppo.py zoo 4 --mask --seed 42
python sb3/train_ppo.py zoo 4 --mask --seed 43
python sb3/train_ppo.py zoo 4 --mask --seed 44

# 10x10 unmasked
python sb3/train_ppo.py zoo 10 --seed 42
python sb3/train_ppo.py zoo 10 --seed 43
python sb3/train_ppo.py zoo 10 --seed 44

# 10x10 masked
python sb3/train_ppo.py zoo 10 --mask --seed 42
python sb3/train_ppo.py zoo 10 --mask --seed 43
python sb3/train_ppo.py zoo 10 --mask --seed 44
```

View the tensorboard log output:

```
# For 4x4 environment
 tensorboard --logdir zoo/4x4/runs

 # For 10x10 environment
 tensorboard --logdir zoo/10x10/runs
```

5.5 Parameters

class sb3_contrib.ppo_mask.**MaskablePPO**(*policy*, *env*, *learning_rate=0.0003*, *n_steps=2048*, *batch_size=64*, *n_epochs=10*, *gamma=0.99*, *gae_lambda=0.95*, *clip_range=0.2*, *clip_range_vf=None*, *normalize_advantage=True*, *ent_coef=0.0*, *vf_coef=0.5*, *max_grad_norm=0.5*, *target_kl=None*, *stats_window_size=100*, *tensorboard_log=None*, *policy_kwargs=None*, *verbose=0*, *seed=None*, *device='auto'*, *_init_setup_model=True*)

Proximal Policy Optimization algorithm (PPO) (clip version) with Invalid Action Masking.

Based on the original Stable Baselines 3 implementation.

Introduction to PPO: https://spinningup.openai.com/en/latest/algorithms/ppo.html Background on Invalid Action Masking: https://arxiv.org/abs/2006.14171

Parameters

- **policy** (Union[str, Type[MaskableActorCriticPolicy]]) – The policy model to use (MlpPolicy, CnnPolicy, ...)
- **env** (Union[Env, VecEnv, str]) – The environment to learn from (if registered in Gym, can be str)
- **learning_rate** (Union[float, Callable[[float], float]]) – The learning rate, it can be a function of the current progress remaining (from 1 to 0)
- **n_steps** (int) – The number of steps to run for each environment per update (i.e. batch size is n_steps * n_env where n_env is number of environment copies running in parallel)
- **batch_size** (Optional[int]) – Minibatch size
- **n_epochs** (int) – Number of epoch when optimizing the surrogate loss
- **gamma** (float) – Discount factor
- **gae_lambda** (float) – Factor for trade-off of bias vs variance for Generalized Advantage Estimator
- **clip_range** (Union[float, Callable[[float], float]]) – Clipping parameter, it can be a function of the current progress remaining (from 1 to 0).
- **clip_range_vf** (Union[None, float, Callable[[float], float]]) – Clipping parameter for the value function, it can be a function of the current progress remaining (from 1 to 0). This is a parameter specific to the OpenAI implementation. If None is passed (default), no clipping will be done on the value function. IMPORTANT: this clipping depends on the reward scaling.
- **normalize_advantage** (bool) – Whether to normalize or not the advantage
- **ent_coef** (float) – Entropy coefficient for the loss calculation
- **vf_coef** (float) – Value function coefficient for the loss calculation
- **max_grad_norm** (float) – The maximum value for the gradient clipping
- **target_kl** (Optional[float]) – Limit the KL divergence between updates, because the clipping is not enough to prevent large update see issue #213 (cf https://github.com/hill-a/stable-baselines/issues/213) By default, there is no limit on the kl div.
- **stats_window_size** (int) – Window size for the rollout logging, specifying the number of episodes to average the reported success rate, mean episode length, and mean reward over

- **tensorboard_log** (Optional[str]) – the log location for tensorboard (if None, no logging)
- **policy_kwargs** (Optional[Dict[str, Any]]) – additional arguments to be passed to the policy on creation
- **verbose** (int) – the verbosity level: 0 no output, 1 info, 2 debug
- **seed** (Optional[int]) – Seed for the pseudo random generators
- **device** (Union[device, str]) – Device (cpu, cuda, ...) on which the code should be run. Setting it to auto, the code will be run on the GPU if possible.
- **_init_setup_model** (bool) – Whether or not to build the network at the creation of the instance

collect_rollouts(*env*, *callback*, *rollout_buffer*, *n_rollout_steps*, *use_masking=True*)

Collect experiences using the current policy and fill a RolloutBuffer. The term rollout here refers to the model-free notion and should not be used with the concept of rollout used in model-based RL or planning.

This method is largely identical to the implementation found in the parent class.

Parameters

- **env** (VecEnv) – The training environment
- **callback** (BaseCallback) – Callback that will be called at each step (and at the beginning and end of the rollout)
- **rollout_buffer** (RolloutBuffer) – Buffer to fill with rollouts
- **n_steps** – Number of experiences to collect per environment
- **use_masking** (bool) – Whether or not to use invalid action masks during training

Return type
bool

Returns
True if function returned with at least *n_rollout_steps* collected, False if callback terminated rollout prematurely.

get_env()

Returns the current environment (can be None if not defined).

Return type
Optional[VecEnv]

Returns
The current environment

get_parameters()

Return the parameters of the agent. This includes parameters from different networks, e.g. critics (value functions) and policies (pi functions).

Return type
Dict[str, Dict]

Returns
Mapping of from names of the objects to PyTorch state-dicts.

get_vec_normalize_env()

Return the VecNormalize wrapper of the training env if it exists.

Return type
Optional[VecNormalize]

Returns
The VecNormalize env.

learn(*total_timesteps*, *callback=None*, *log_interval=1*, *tb_log_name='PPO'*, *reset_num_timesteps=True*, *use_masking=True*, *progress_bar=False*)

Return a trained model.

Parameters

- **total_timesteps** (int) – The total number of samples (env steps) to train on
- **callback** (Union[None, Callable, List[BaseCallback], BaseCallback]) – callback(s) called at every step with state of the algorithm.
- **log_interval** (int) – The number of episodes before logging.
- **tb_log_name** (str) – the name of the run for TensorBoard logging
- **reset_num_timesteps** (bool) – whether or not to reset the current timestep number (used in logging)
- **progress_bar** (bool) – Display a progress bar using tqdm and rich.

Return type
TypeVar(SelfMaskablePPO, bound= MaskablePPO)

Returns
the trained model

classmethod load(*path*, *env=None*, *device='auto'*, *custom_objects=None*, *print_system_info=False*, *force_reset=True*, ***kwargs*)

Load the model from a zip-file. Warning: load re-creates the model from scratch, it does not update it in-place! For an in-place load use set_parameters instead.

Parameters

- **path** (Union[str, Path, BufferedIOBase]) – path to the file (or a file-like) where to load the agent from
- **env** (Union[Env, VecEnv, None]) – the new environment to run the loaded model on (can be None if you only need prediction from a trained model) has priority over any saved environment
- **device** (Union[device, str]) – Device on which the code should run.
- **custom_objects** (Optional[Dict[str, Any]]) – Dictionary of objects to replace upon loading. If a variable is present in this dictionary as a key, it will not be deserialized and the corresponding item will be used instead. Similar to custom_objects in keras.models.load_model. Useful when you have an object in file that can not be deserialized.
- **print_system_info** (bool) – Whether to print system info from the saved model and the current system info (useful to debug loading issues)
- **force_reset** (bool) – Force call to reset() before training to avoid unexpected behavior. See https://github.com/DLR-RM/stable-baselines3/issues/597
- **kwargs** – extra arguments to change the model when loading

Return type
TypeVar(SelfBaseAlgorithm, bound= BaseAlgorithm)

Returns
new model instance with loaded parameters

`property logger: Logger`

Getter for the logger object.

`predict`(*observation*, *state=None*, *episode_start=None*, *deterministic=False*, *action_masks=None*)

Get the policy action from an observation (and optional hidden state). Includes sugar-coating to handle different observations (e.g. normalizing images).

Parameters

- **observation** (ndarray) – the input observation
- **state** (Optional[Tuple[ndarray, ...]]) – The last hidden states (can be None, used in recurrent policies)
- **episode_start** (Optional[ndarray]) – The last masks (can be None, used in recurrent policies) this correspond to beginning of episodes, where the hidden states of the RNN must be reset.
- **deterministic** (bool) – Whether or not to return deterministic actions.

Return type
Tuple[ndarray, Optional[Tuple[ndarray, ...]]]

Returns
the model's action and the next hidden state (used in recurrent policies)

`save`(*path*, *exclude=None*, *include=None*)

Save all the attributes of the object and the model parameters in a zip-file.

Parameters

- **path** (Union[str, Path, BufferedIOBase]) – path to the file where the rl agent should be saved
- **exclude** (Optional[Iterable[str]]) – name of parameters that should be excluded in addition to the default ones
- **include** (Optional[Iterable[str]]) – name of parameters that might be excluded but should be included anyway

Return type
None

`set_env`(*env*, *force_reset=True*)

Checks the validity of the environment, and if it is coherent, set it as the current environment. Furthermore wrap any non vectorized env into a vectorized checked parameters: - observation_space - action_space

Parameters

- **env** (Union[Env, VecEnv]) – The environment for learning a policy
- **force_reset** (bool) – Force call to `reset()` before training to avoid unexpected behavior. See issue https://github.com/DLR-RM/stable-baselines3/issues/597

Return type
None

`set_logger`(*logger*)

Setter for for logger object. :rtype: None

Warning: When passing a custom logger object, this will overwrite `tensorboard_log` and `verbose` settings passed to the constructor.

`set_parameters`(*load_path_or_dict*, *exact_match=True*, *device='auto'*)

Load parameters from a given zip-file or a nested dictionary containing parameters for different modules (see `get_parameters`).

Parameters

- **`load_path_or_iter`** – Location of the saved data (path or file-like, see `save`), or a nested dictionary containing nn.Module parameters used by the policy. The dictionary maps object names to a state-dictionary returned by `torch.nn.Module.state_dict()`.
- **`exact_match`** (`bool`) – If True, the given parameters should include parameters for each module and each of their parameters, otherwise raises an Exception. If set to False, this can be used to update only specific parameters.
- **`device`** (`Union[device, str]`) – Device on which the code should run.

Return type
`None`

`set_random_seed`(*seed=None*)

Set the seed of the pseudo-random generators (python, numpy, pytorch, gym, action_space)

Parameters
`seed` (`Optional[int]`) –

Return type
`None`

`train`()

Update policy using the currently gathered rollout buffer.

Return type
`None`

5.6 MaskablePPO Policies

sb3_contrib.ppo_mask.**MlpPolicy**

alias of MaskableActorCriticPolicy

class sb3_contrib.common.maskable.policies.**MaskableActorCriticPolicy**(*observation_space*, *action_space*, *lr_schedule*, *net_arch=None*, *activation_fn=<class 'torch.nn.modules.activation.Tanh'>*, *ortho_init=True*, *features_extractor_class=<class 'stable_baselines3.common.torch_layers.Flatten*, *features_extractor_kwargs=None*, *share_features_extractor=True*, *normalize_images=True*, *optimizer_class=<class 'torch.optim.adam.Adam'>*, *optimizer_kwargs=None*)

Policy class for actor-critic algorithms (has both policy and value prediction). Used by A2C, PPO and the likes.

Parameters

- **observation_space** (Space) – Observation space
- **action_space** (Space) – Action space
- **lr_schedule** (Callable[[float], float]) – Learning rate schedule (could be constant)
- **net_arch** (Union[List[int], Dict[str, List[int]], None]) – The specification of the policy and value networks.
- **activation_fn** (Type[Module]) – Activation function
- **ortho_init** (bool) – Whether to use or not orthogonal initialization
- **features_extractor_class** (Type[BaseFeaturesExtractor]) – Features extractor to use.
- **features_extractor_kwargs** (Optional[Dict[str, Any]]) – Keyword arguments to pass to the features extractor.
- **share_features_extractor** (bool) – If True, the features extractor is shared between the policy and value networks.
- **normalize_images** (bool) – Whether to normalize images or not, dividing by 255.0 (True by default)
- **optimizer_class** (Type[Optimizer]) – The optimizer to use, th.optim.Adam by default
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional keyword arguments, excluding the learning rate, to pass to the optimizer

evaluate_actions(*obs*, *actions*, *action_masks=None*)

Evaluate actions according to the current policy, given the observations.

Parameters

- **obs** (Tensor) – Observation

- **actions** (Tensor) – Actions

Return type
Tuple[Tensor, Tensor, Tensor]

Returns
estimated value, log likelihood of taking those actions and entropy of the action distribution.

extract_features(*obs*)

Preprocess the observation if needed and extract features. :type obs: Tensor :param obs: Observation :rtype: Union[Tensor, Tuple[Tensor, Tensor]] :return: the output of the features extractor(s)

forward(*obs*, *deterministic=False*, *action_masks=None*)

Forward pass in all the networks (actor and critic)

Parameters

- **obs** (Tensor) – Observation
- **deterministic** (bool) – Whether to sample or use deterministic actions
- **action_masks** (Optional[ndarray]) – Action masks to apply to the action distribution

Return type
Tuple[Tensor, Tensor, Tensor]

Returns
action, value and log probability of the action

get_distribution(*obs*, *action_masks=None*)

Get the current policy distribution given the observations.

Parameters

- **obs** (Tensor) – Observation
- **action_masks** (Optional[ndarray]) – Actions' mask

Return type
MaskableDistribution

Returns
the action distribution.

predict(*observation*, *state=None*, *episode_start=None*, *deterministic=False*, *action_masks=None*)

Get the policy action from an observation (and optional hidden state). Includes sugar-coating to handle different observations (e.g. normalizing images).

Parameters

- **observation** (Union[ndarray, Dict[str, ndarray]]) – the input observation
- **state** (Optional[Tuple[ndarray, ...]]) – The last states (can be None, used in recurrent policies)
- **episode_start** (Optional[ndarray]) – The last masks (can be None, used in recurrent policies)
- **deterministic** (bool) – Whether or not to return deterministic actions.
- **action_masks** (Optional[ndarray]) – Action masks to apply to the action distribution

Return type
Tuple[ndarray, Optional[Tuple[ndarray, ...]]]

Returns
the model's action and the next state (used in recurrent policies)

predict_values(*obs*)

Get the estimated values according to the current policy given the observations.

Parameters
obs (Tensor) – Observation

Return type
Tensor

Returns
the estimated values.

sb3_contrib.ppo_mask.**CnnPolicy**

alias of MaskableActorCriticCnnPolicy

class sb3_contrib.common.maskable.policies.**MaskableActorCriticCnnPolicy**(*observation_space*, *action_space*, *lr_schedule*, *net_arch=None*, *activation_fn=<class 'torch.nn.modules.activation.Tanh'>*, *ortho_init=True*, *features_extractor_class=<class 'stable_baselines3.common.torch_layers.Nat*
features_extractor_kwargs=None, *share_features_extractor=True*, *normalize_images=True*, *optimizer_class=<class 'torch.optim.adam.Adam'>*, *optimizer_kwargs=None*)

CNN policy class for actor-critic algorithms (has both policy and value prediction). Used by A2C, PPO and the likes.

Parameters

- **observation_space** (Space) – Observation space
- **action_space** (Space) – Action space
- **lr_schedule** (Callable[[float], float]) – Learning rate schedule (could be constant)
- **net_arch** (Union[List[int], Dict[str, List[int]], None]) – The specification of the policy and value networks.
- **activation_fn** (Type[Module]) – Activation function
- **ortho_init** (bool) – Whether to use or not orthogonal initialization
- **features_extractor_class** (Type[BaseFeaturesExtractor]) – Features extractor to use.
- **features_extractor_kwargs** (Optional[Dict[str, Any]]) – Keyword arguments to pass to the features extractor.

- **share_features_extractor** (bool) – If True, the features extractor is shared between the policy and value networks.
- **normalize_images** (bool) – Whether to normalize images or not, dividing by 255.0 (True by default)
- **optimizer_class** (Type[Optimizer]) – The optimizer to use, th.optim.Adam by default
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional keyword arguments, excluding the learning rate, to pass to the optimizer

sb3_contrib.ppo_mask.**MultiInputPolicy**

alias of MaskableMultiInputActorCriticPolicy

class sb3_contrib.common.maskable.policies.**MaskableMultiInputActorCriticPolicy**(*observation_space*, *action_space*, *lr_schedule*, *net_arch=None*, *activation_fn=<class 'torch.nn.modules.activation.Tan*
ortho_init=True, *features_extractor_class=<class 'stable_baselines3.common.torch_l*
features_extractor_kwargs=None, *share_features_extractor=True*, *normalize_images=True*, *optimizer_class=<class 'torch.optim.adam.Adam'>*, *optimizer_kwargs=None*)

MultiInputActorClass policy class for actor-critic algorithms (has both policy and value prediction). Used by A2C, PPO and the likes.

Parameters

- **observation_space** (Dict) – Observation space (Tuple)
- **action_space** (Space) – Action space
- **lr_schedule** (Callable[[float], float]) – Learning rate schedule (could be constant)
- **net_arch** (Union[List[int], Dict[str, List[int]], None]) – The specification of the policy and value networks.
- **activation_fn** (Type[Module]) – Activation function
- **ortho_init** (bool) – Whether to use or not orthogonal initialization
- **features_extractor_class** (Type[BaseFeaturesExtractor]) – Uses the CombinedExtractor

- **features_extractor_kwargs** (Optional[Dict[str, Any]]) – Keyword arguments to pass to the feature extractor.
- **share_features_extractor** (bool) – If True, the features extractor is shared between the policy and value networks.
- **normalize_images** (bool) – Whether to normalize images or not, dividing by 255.0 (True by default)
- **optimizer_class** (Type[Optimizer]) – The optimizer to use, th.optim.Adam by default
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional keyword arguments, excluding the learning rate, to pass to the optimizer

CHAPTER

SIX

RECURRENT PPO

Implementation of recurrent policies for the Proximal Policy Optimization (PPO) algorithm. Other than adding support for recurrent policies (LSTM here), the behavior is the same as in SB3's core PPO algorithm.

Available Policies

MlpLstmPolicy	alias of `RecurrentActorCriticPolicy`
CnnLstmPolicy	alias of `RecurrentActorCriticCnnPolicy`
MultiInputLstmPolicy	alias of `RecurrentMultiInputActorCriticPolicy`

6.1 Notes

- Blog post: https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/

6.2 Can I use?

- Recurrent policies: ✓
- Multi processing: ✓
- Gym spaces:

Space	Action	Observation
Discrete	✓	✓
Box	✓	✓
MultiDiscrete	✓	✓
MultiBinary	✓	✓
Dict		✓

6.3 Example

Note: It is particularly important to pass the `lstm_states` and `episode_start` argument to the `predict()` method, so the cell and hidden states of the LSTM are correctly updated.

```
import numpy as np

from sb3_contrib import RecurrentPPO
from stable_baselines3.common.evaluation import evaluate_policy

model = RecurrentPPO("MlpLstmPolicy", "CartPole-v1", verbose=1)
model.learn(5000)

env = model.get_env()
mean_reward, std_reward = evaluate_policy(model, env, n_eval_episodes=20, warn=False)
print(mean_reward)

model.save("ppo_recurrent")
del model # remove to demonstrate saving and loading

model = RecurrentPPO.load("ppo_recurrent")

obs = env.reset()
# cell and hidden state of the LSTM
lstm_states = None
num_envs = 1
# Episode start signals are used to reset the lstm states
episode_starts = np.ones((num_envs,), dtype=bool)
while True:
    action, lstm_states = model.predict(obs, state=lstm_states, episode_start=episode_
↪starts, deterministic=True)
    obs, rewards, dones, info = env.step(action)
    episode_starts = dones
    env.render()
```

6.4 Results

Report on environments with masked velocity (with and without framestack) can be found here: https://wandb.ai/sb3/no-vel-envs/reports/PPO-vs-RecurrentPPO-aka-PPO-LSTM-on-environments-with-masked-velocity--VmlldzoxOTI4NjE4

`RecurrentPPO` was evaluated against PPO on:

- PendulumNoVel-v1
- LunarLanderNoVel-v2
- CartPoleNoVel-v1
- MountainCarContinuousNoVel-v0
- CarRacing-v0

6.4.1 How to replicate the results?

Clone the repo for the experiment:

```
git clone https://github.com/DLR-RM/rl-baselines3-zoo
cd rl-baselines3-zoo
git checkout feat/recurrent-ppo
```

Run the benchmark (replace `$ENV_ID` by the envs mentioned above):

```
python train.py --algo ppo_lstm --env $ENV_ID --eval-episodes 10 --eval-freq 10000
```

6.5 Parameters

class sb3_contrib.ppo_recurrent.**RecurrentPPO**(*policy*, *env*, *learning_rate=0.0003*, *n_steps=128*, *batch_size=128*, *n_epochs=10*, *gamma=0.99*, *gae_lambda=0.95*, *clip_range=0.2*, *clip_range_vf=None*, *normalize_advantage=True*, *ent_coef=0.0*, *vf_coef=0.5*, *max_grad_norm=0.5*, *use_sde=False*, *sde_sample_freq=-1*, *target_kl=None*, *stats_window_size=100*, *tensorboard_log=None*, *policy_kwargs=None*, *verbose=0*, *seed=None*, *device='auto'*, *_init_setup_model=True*)

Proximal Policy Optimization algorithm (PPO) (clip version) with support for recurrent policies (LSTM).

Based on the original Stable Baselines 3 implementation.

Introduction to PPO: https://spinningup.openai.com/en/latest/algorithms/ppo.html

Parameters

- **policy** (Union[str, Type[RecurrentActorCriticPolicy]]) – The policy model to use (MlpPolicy, CnnPolicy, ...)
- **env** (Union[Env, VecEnv, str]) – The environment to learn from (if registered in Gym, can be str)
- **learning_rate** (Union[float, Callable[[float], float]]) – The learning rate, it can be a function of the current progress remaining (from 1 to 0)
- **n_steps** (int) – The number of steps to run for each environment per update (i.e. batch size is n_steps * n_env where n_env is number of environment copies running in parallel)
- **batch_size** (Optional[int]) – Minibatch size
- **n_epochs** (int) – Number of epoch when optimizing the surrogate loss
- **gamma** (float) – Discount factor
- **gae_lambda** (float) – Factor for trade-off of bias vs variance for Generalized Advantage Estimator
- **clip_range** (Union[float, Callable[[float], float]]) – Clipping parameter, it can be a function of the current progress remaining (from 1 to 0).
- **clip_range_vf** (Union[None, float, Callable[[float], float]]) – Clipping parameter for the value function, it can be a function of the current progress remaining (from 1 to 0). This is a parameter specific to the OpenAI implementation. If None is passed (default), no

clipping will be done on the value function. IMPORTANT: this clipping depends on the reward scaling.

- **normalize_advantage** (bool) – Whether to normalize or not the advantage
- **ent_coef** (float) – Entropy coefficient for the loss calculation
- **vf_coef** (float) – Value function coefficient for the loss calculation
- **max_grad_norm** (float) – The maximum value for the gradient clipping
- **target_kl** (Optional[float]) – Limit the KL divergence between updates, because the clipping is not enough to prevent large update see issue #213 (cf https://github.com/hill-a/stable-baselines/issues/213) By default, there is no limit on the kl div.
- **stats_window_size** (int) – Window size for the rollout logging, specifying the number of episodes to average the reported success rate, mean episode length, and mean reward over
- **tensorboard_log** (Optional[str]) – the log location for tensorboard (if None, no logging)
- **policy_kwargs** (Optional[Dict[str, Any]]) – additional arguments to be passed to the policy on creation
- **verbose** (int) – the verbosity level: 0 no output, 1 info, 2 debug
- **seed** (Optional[int]) – Seed for the pseudo random generators
- **device** (Union[device, str]) – Device (cpu, cuda, ...) on which the code should be run. Setting it to auto, the code will be run on the GPU if possible.
- **_init_setup_model** (bool) – Whether or not to build the network at the creation of the instance

collect_rollouts(*env*, *callback*, *rollout_buffer*, *n_rollout_steps*)

Collect experiences using the current policy and fill a RolloutBuffer. The term rollout here refers to the model-free notion and should not be used with the concept of rollout used in model-based RL or planning.

Parameters

- **env** (VecEnv) – The training environment
- **callback** (BaseCallback) – Callback that will be called at each step (and at the beginning and end of the rollout)
- **rollout_buffer** (RolloutBuffer) – Buffer to fill with rollouts
- **n_steps** – Number of experiences to collect per environment

Return type
bool

Returns
True if function returned with at least *n_rollout_steps* collected, False if callback terminated rollout prematurely.

get_env()

Returns the current environment (can be None if not defined).

Return type
Optional[VecEnv]

Returns
The current environment

`get_parameters()`

Return the parameters of the agent. This includes parameters from different networks, e.g. critics (value functions) and policies (pi functions).

Return type
`Dict[str, Dict]`

Returns
Mapping of from names of the objects to PyTorch state-dicts.

`get_vec_normalize_env()`

Return the `VecNormalize` wrapper of the training env if it exists.

Return type
`Optional[VecNormalize]`

Returns
The `VecNormalize` env.

`learn(total_timesteps, callback=None, log_interval=1, tb_log_name='RecurrentPPO', reset_num_timesteps=True, progress_bar=False)`

Return a trained model.

Parameters

- **total_timesteps** (`int`) – The total number of samples (env steps) to train on
- **callback** (`Union[None, Callable, List[BaseCallback], BaseCallback]`) – callback(s) called at every step with state of the algorithm.
- **log_interval** (`int`) – The number of episodes before logging.
- **tb_log_name** (`str`) – the name of the run for TensorBoard logging
- **reset_num_timesteps** (`bool`) – whether or not to reset the current timestep number (used in logging)
- **progress_bar** (`bool`) – Display a progress bar using tqdm and rich.

Return type
`TypeVar(SelfRecurrentPPO, bound= RecurrentPPO)`

Returns
the trained model

`classmethod load(path, env=None, device='auto', custom_objects=None, print_system_info=False, force_reset=True, **kwargs)`

Load the model from a zip-file. Warning: `load` re-creates the model from scratch, it does not update it in-place! For an in-place load use `set_parameters` instead.

Parameters

- **path** (`Union[str, Path, BufferedIOBase]`) – path to the file (or a file-like) where to load the agent from
- **env** (`Union[Env, VecEnv, None]`) – the new environment to run the loaded model on (can be None if you only need prediction from a trained model) has priority over any saved environment
- **device** (`Union[device, str]`) – Device on which the code should run.
- **custom_objects** (`Optional[Dict[str, Any]]`) – Dictionary of objects to replace upon loading. If a variable is present in this dictionary as a key, it will not be deserialized and

the corresponding item will be used instead. Similar to custom_objects in `keras.models.load_model`. Useful when you have an object in file that can not be deserialized.

- **print_system_info** (`bool`) – Whether to print system info from the saved model and the current system info (useful to debug loading issues)
- **force_reset** (`bool`) – Force call to `reset()` before training to avoid unexpected behavior. See https://github.com/DLR-RM/stable-baselines3/issues/597
- **kwargs** – extra arguments to change the model when loading

Return type
`TypeVar(SelfBaseAlgorithm, bound= BaseAlgorithm)`

Returns
new model instance with loaded parameters

property logger: Logger

Getter for the logger object.

predict(*observation*, *state=None*, *episode_start=None*, *deterministic=False*)

Get the policy action from an observation (and optional hidden state). Includes sugar-coating to handle different observations (e.g. normalizing images).

Parameters

- **observation** (`Union[ndarray, Dict[str, ndarray]]`) – the input observation
- **state** (`Optional[Tuple[ndarray, ...]]`) – The last hidden states (can be None, used in recurrent policies)
- **episode_start** (`Optional[ndarray]`) – The last masks (can be None, used in recurrent policies) this correspond to beginning of episodes, where the hidden states of the RNN must be reset.
- **deterministic** (`bool`) – Whether or not to return deterministic actions.

Return type
`Tuple[ndarray, Optional[Tuple[ndarray, ...]]]`

Returns
the model's action and the next hidden state (used in recurrent policies)

save(*path*, *exclude=None*, *include=None*)

Save all the attributes of the object and the model parameters in a zip-file.

Parameters

- **path** (`Union[str, Path, BufferedIOBase]`) – path to the file where the rl agent should be saved
- **exclude** (`Optional[Iterable[str]]`) – name of parameters that should be excluded in addition to the default ones
- **include** (`Optional[Iterable[str]]`) – name of parameters that might be excluded but should be included anyway

Return type
`None`

set_env(*env*, *force_reset=True*)

Checks the validity of the environment, and if it is coherent, set it as the current environment. Furthermore wrap any non vectorized env into a vectorized checked parameters: - observation_space - action_space

Parameters

- **env** (Union[Env, VecEnv]) – The environment for learning a policy
- **force_reset** (bool) – Force call to `reset()` before training to avoid unexpected behavior. See issue https://github.com/DLR-RM/stable-baselines3/issues/597

Return type
None

`set_logger`(*logger*)

Setter for for logger object. :rtype: None

> **Warning:** When passing a custom logger object, this will overwrite `tensorboard_log` and `verbose` settings passed to the constructor.

`set_parameters`(*load_path_or_dict*, *exact_match=True*, *device='auto'*)

Load parameters from a given zip-file or a nested dictionary containing parameters for different modules (see `get_parameters`).

Parameters

- **load_path_or_iter** – Location of the saved data (path or file-like, see `save`), or a nested dictionary containing nn.Module parameters used by the policy. The dictionary maps object names to a state-dictionary returned by `torch.nn.Module.state_dict()`.
- **exact_match** (bool) – If True, the given parameters should include parameters for each module and each of their parameters, otherwise raises an Exception. If set to False, this can be used to update only specific parameters.
- **device** (Union[device, str]) – Device on which the code should run.

Return type
None

`set_random_seed`(*seed=None*)

Set the seed of the pseudo-random generators (python, numpy, pytorch, gym, action_space)

Parameters
seed (Optional[int]) –

Return type
None

`train`()

Update policy using the currently gathered rollout buffer.

Return type
None

6.6 RecurrentPPO Policies

sb3_contrib.ppo_recurrent.**MlpLstmPolicy**

alias of RecurrentActorCriticPolicy

class sb3_contrib.common.recurrent.policies.**RecurrentActorCriticPolicy**(*observation_space*, *action_space*, *lr_schedule*, *net_arch=None*, *activation_fn=<class 'torch.nn.modules.activation.Tanh'>*, *ortho_init=True*, *use_sde=False*, *log_std_init=0.0*, *full_std=True*, *use_expln=False*, *squash_output=False*, *features_extractor_class=<class 'stable_baselines3.common.torch_layers.Flatt*
features_extractor_kwargs=None, *share_features_extractor=True*, *normalize_images=True*, *optimizer_class=<class 'torch.optim.adam.Adam'>*, *optimizer_kwargs=None*, *lstm_hidden_size=256*, *n_lstm_layers=1*, *shared_lstm=False*, *enable_critic_lstm=True*, *lstm_kwargs=None*)

Recurrent policy class for actor-critic algorithms (has both policy and value prediction). To be used with A2C, PPO and the likes. It assumes that both the actor and the critic LSTM have the same architecture.

Parameters

- **observation_space** (Space) – Observation space
- **action_space** (Space) – Action space
- **lr_schedule** (Callable[[float], float]) – Learning rate schedule (could be constant)
- **net_arch** (Union[List[int], Dict[str, List[int]], None]) – The specification of the policy and value networks.
- **activation_fn** (Type[Module]) – Activation function
- **ortho_init** (bool) – Whether to use or not orthogonal initialization
- **use_sde** (bool) – Whether to use State Dependent Exploration or not
- **log_std_init** (float) – Initial value for the log standard deviation

- **full_std** (bool) – Whether to use (n_features x n_actions) parameters for the std instead of only (n_features,) when using gSDE
- **use_expln** (bool) – Use expln() function instead of exp() to ensure a positive standard deviation (cf paper). It allows to keep variance above zero and prevent it from growing too fast. In practice, exp() is usually enough.
- **squash_output** (bool) – Whether to squash the output using a tanh function, this allows to ensure boundaries when using gSDE.
- **features_extractor_class** (Type[BaseFeaturesExtractor]) – Features extractor to use.
- **features_extractor_kwargs** (Optional[Dict[str, Any]]) – Keyword arguments to pass to the features extractor.
- **share_features_extractor** (bool) – If True, the features extractor is shared between the policy and value networks.
- **normalize_images** (bool) – Whether to normalize images or not, dividing by 255.0 (True by default)
- **optimizer_class** (Type[Optimizer]) – The optimizer to use, th.optim.Adam by default
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional keyword arguments, excluding the learning rate, to pass to the optimizer
- **lstm_hidden_size** (int) – Number of hidden units for each LSTM layer.
- **n_lstm_layers** (int) – Number of LSTM layers.
- **shared_lstm** (bool) – Whether the LSTM is shared between the actor and the critic (in that case, only the actor gradient is used) By default, the actor and the critic have two separate LSTM.
- **enable_critic_lstm** (bool) – Use a seperate LSTM for the critic.
- **lstm_kwargs** (Optional[Dict[str, Any]]) – Additional keyword arguments to pass the the LSTM constructor.

evaluate_actions(*obs*, *actions*, *lstm_states*, *episode_starts*)

Evaluate actions according to the current policy, given the observations.

Parameters

- **obs** (Tensor) – Observation.
- **actions** (Tensor) –
- **lstm_states** (RNNStates) – The last hidden and memory states for the LSTM.
- **episode_starts** (Tensor) – Whether the observations correspond to new episodes or not (we reset the lstm states in that case).

Return type
Tuple[Tensor, Tensor, Tensor]

Returns
estimated value, log likelihood of taking those actions and entropy of the action distribution.

forward(*obs*, *lstm_states*, *episode_starts*, *deterministic=False*)

Forward pass in all the networks (actor and critic)

Parameters

- **obs** (Tensor) – Observation. Observation
- **lstm_states** (RNNStates) – The last hidden and memory states for the LSTM.
- **episode_starts** (Tensor) – Whether the observations correspond to new episodes or not (we reset the lstm states in that case).
- **deterministic** (bool) – Whether to sample or use deterministic actions

Return type
Tuple[Tensor, Tensor, Tensor, RNNStates]

Returns
action, value and log probability of the action

get_distribution(*obs*, *lstm_states*, *episode_starts*)

Get the current policy distribution given the observations.

Parameters

- **obs** (Tensor) – Observation.
- **lstm_states** (Tuple[Tensor, Tensor]) – The last hidden and memory states for the LSTM.
- **episode_starts** (Tensor) – Whether the observations correspond to new episodes or not (we reset the lstm states in that case).

Return type
Tuple[Distribution, Tuple[Tensor, ...]]

Returns
the action distribution and new hidden states.

predict(*observation*, *state=None*, *episode_start=None*, *deterministic=False*)

Get the policy action from an observation (and optional hidden state). Includes sugar-coating to handle different observations (e.g. normalizing images).

Parameters

- **observation** (Union[ndarray, Dict[str, ndarray]]) – the input observation
- **lstm_states** – The last hidden and memory states for the LSTM.
- **episode_starts** – Whether the observations correspond to new episodes or not (we reset the lstm states in that case).
- **deterministic** (bool) – Whether or not to return deterministic actions.

Return type
Tuple[ndarray, Optional[Tuple[ndarray, ...]]]

Returns
the model's action and the next hidden state (used in recurrent policies)

predict_values(*obs*, *lstm_states*, *episode_starts*)

Get the estimated values according to the current policy given the observations.

Parameters

- **obs** (Tensor) – Observation.
- **lstm_states** (Tuple[Tensor, Tensor]) – The last hidden and memory states for the LSTM.

- **episode_starts** (Tensor) – Whether the observations correspond to new episodes or not (we reset the lstm states in that case).

Return type
Tensor

Returns
the estimated values.

sb3_contrib.ppo_recurrent.**CnnLstmPolicy**

alias of RecurrentActorCriticCnnPolicy

class sb3_contrib.common.recurrent.policies.**RecurrentActorCriticCnnPolicy**(*observation_space*, *action_space*, *lr_schedule*, *net_arch=None*, *activation_fn=<class 'torch.nn.modules.activation.Tanh'>*, *ortho_init=True*, *use_sde=False*, *log_std_init=0.0*, *full_std=True*, *use_expln=False*, *squash_output=False*, *features_extractor_class=<class 'stable_baselines3.common.torch_layers.N*, *features_extractor_kwargs=None*, *share_features_extractor=True*, *normalize_images=True*, *optimizer_class=<class 'torch.optim.adam.Adam'>*, *optimizer_kwargs=None*, *lstm_hidden_size=256*, *n_lstm_layers=1*, *shared_lstm=False*, *enable_critic_lstm=True*, *lstm_kwargs=None*)

CNN recurrent policy class for actor-critic algorithms (has both policy and value prediction). Used by A2C, PPO and the likes.

Parameters

- **observation_space** (Space) – Observation space
- **action_space** (Space) – Action space
- **lr_schedule** (Callable[[float], float]) – Learning rate schedule (could be constant)
- **net_arch** (Union[List[int], Dict[str, List[int]], None]) – The specification of the policy and value networks.

- **activation_fn** (Type[Module]) – Activation function
- **ortho_init** (bool) – Whether to use or not orthogonal initialization
- **use_sde** (bool) – Whether to use State Dependent Exploration or not
- **log_std_init** (float) – Initial value for the log standard deviation
- **full_std** (bool) – Whether to use (n_features x n_actions) parameters for the std instead of only (n_features,) when using gSDE
- **use_expln** (bool) – Use `expln()` function instead of `exp()` to ensure a positive standard deviation (cf paper). It allows to keep variance above zero and prevent it from growing too fast. In practice, `exp()` is usually enough.
- **squash_output** (bool) – Whether to squash the output using a tanh function, this allows to ensure boundaries when using gSDE.
- **features_extractor_class** (Type[BaseFeaturesExtractor]) – Features extractor to use.
- **features_extractor_kwargs** (Optional[Dict[str, Any]]) – Keyword arguments to pass to the features extractor.
- **share_features_extractor** (bool) – If True, the features extractor is shared between the policy and value networks.
- **normalize_images** (bool) – Whether to normalize images or not, dividing by 255.0 (True by default)
- **optimizer_class** (Type[Optimizer]) – The optimizer to use, `th.optim.Adam` by default
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional keyword arguments, excluding the learning rate, to pass to the optimizer
- **lstm_hidden_size** (int) – Number of hidden units for each LSTM layer.
- **n_lstm_layers** (int) – Number of LSTM layers.
- **shared_lstm** (bool) – Whether the LSTM is shared between the actor and the critic. By default, only the actor has a recurrent network.
- **enable_critic_lstm** (bool) – Use a seperate LSTM for the critic.
- **lstm_kwargs** (Optional[Dict[str, Any]]) – Additional keyword arguments to pass the the LSTM constructor.

sb3_contrib.ppo_recurrent.**MultiInputLstmPolicy**

alias of RecurrentMultiInputActorCriticPolicy

class sb3_contrib.common.recurrent.policies.**RecurrentMultiInputActorCriticPolicy**(*observation_space*, *action_space*, *lr_schedule*, *net_arch=None*, *activation_fn=<class 'torch.nn.modules.activation.*, *ortho_init=True*, *use_sde=False*, *log_std_init=0.0*, *full_std=True*, *use_expln=False*, *squash_output=False*, *features_extractor_class=<class 'stable_baselines3.common.torch*, *features_extractor_kwargs=Non*, *share_features_extractor=Tr*, *normalize_images=True*, *optimizer_class=<class 'torch.optim.adam.Adam'>*, *optimizer_kwargs=None*, *lstm_hidden_size=256*, *n_lstm_layers=1*, *shared_lstm=False*, *enable_critic_lstm=True*, *lstm_kwargs=None*)

MultiInputActorClass policy class for actor-critic algorithms (has both policy and value prediction). Used by A2C, PPO and the likes.

Parameters

- **observation_space** (Space) – Observation space
- **action_space** (Space) – Action space
- **lr_schedule** (Callable[[float], float]) – Learning rate schedule (could be constant)
- **net_arch** (Union[List[int], Dict[str, List[int]], None]) – The specification of the policy and value networks.
- **activation_fn** (Type[Module]) – Activation function
- **ortho_init** (bool) – Whether to use or not orthogonal initialization
- **use_sde** (bool) – Whether to use State Dependent Exploration or not
- **log_std_init** (float) – Initial value for the log standard deviation
- **full_std** (bool) – Whether to use (n_features x n_actions) parameters for the std instead of only (n_features,) when using gSDE

- **use_expln** (bool) – Use expln() function instead of exp() to ensure a positive standard deviation (cf paper). It allows to keep variance above zero and prevent it from growing too fast. In practice, exp() is usually enough.
- **squash_output** (bool) – Whether to squash the output using a tanh function, this allows to ensure boundaries when using gSDE.
- **features_extractor_class** (Type[BaseFeaturesExtractor]) – Features extractor to use.
- **features_extractor_kwargs** (Optional[Dict[str, Any]]) – Keyword arguments to pass to the features extractor.
- **share_features_extractor** (bool) – If True, the features extractor is shared between the policy and value networks.
- **normalize_images** (bool) – Whether to normalize images or not, dividing by 255.0 (True by default)
- **optimizer_class** (Type[Optimizer]) – The optimizer to use, th.optim.Adam by default
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional keyword arguments, excluding the learning rate, to pass to the optimizer
- **lstm_hidden_size** (int) – Number of hidden units for each LSTM layer.
- **n_lstm_layers** (int) – Number of LSTM layers.
- **shared_lstm** (bool) – Whether the LSTM is shared between the actor and the critic. By default, only the actor has a recurrent network.
- **enable_critic_lstm** (bool) – Use a seperate LSTM for the critic.
- **lstm_kwargs** (Optional[Dict[str, Any]]) – Additional keyword arguments to pass the the LSTM constructor.

CHAPTER

SEVEN

QR-DQN

Quantile Regression DQN (QR-DQN) builds on Deep Q-Network (DQN) and make use of quantile regression to explicitly model the distribution over returns, instead of predicting the mean return (DQN).

Available Policies

MlpPolicy	alias of `QRDQNPolicy`
CnnPolicy	Policy class for QR-DQN when using images as input.
MultiInputPolicy	Policy class for QR-DQN when using dict observations as input.

7.1 Notes

- Original paper: https://arxiv.org/abs/1710.100442
- Distributional RL (C51): https://arxiv.org/abs/1707.06887
- Further reference: https://github.com/amy12xx/ml_notes_and_reports/blob/master/distributional_rl/QRDQN.pdf

7.2 Can I use?

- Recurrent policies:
- Multi processing: ✓
- Gym spaces:

Space	Action	Observation
Discrete	✓	✓
Box		✓
MultiDiscrete		✓
MultiBinary		✓
Dict		✓

7.3 Example

```
import gym

from sb3_contrib import QRDQN

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

policy_kwargs = dict(n_quantiles=50)
model = QRDQN("MlpPolicy", env, policy_kwargs=policy_kwargs, verbose=1)
model.learn(total_timesteps=10000, log_interval=4)
model.save("qrdqn_cartpole")

del model # remove to demonstrate saving and loading

model = QRDQN.load("qrdqn_cartpole")

obs = env.reset()
while True:
    action, _states = model.predict(obs, deterministic=True)
    obs, reward, done, info = env.step(action)
    env.render()
    if done:
      obs = env.reset()
```

7.4 Results

Result on Atari environments (10M steps, Pong and Breakout) and classic control tasks using 3 and 5 seeds.

The complete learning curves are available in the associated PR.

Note: QR-DQN implementation was validated against Intel Coach one which roughly compare to the original paper results (we trained the agent with a smaller budget).

Environments	QR-DQN	DQN
Breakout	413 +/- 21	~300
Pong	20 +/- 0	~20
CartPole	386 +/- 64	500 +/- 0
MountainCar	-111 +/- 4	-107 +/- 4
LunarLander	168 +/- 39	195 +/- 28
Acrobot	-73 +/- 2	-74 +/- 2

7.4.1 How to replicate the results?

Clone RL-Zoo fork and checkout the branch `feat/qrdqn`:

```
git clone https://github.com/ku2482/rl-baselines3-zoo/
cd rl-baselines3-zoo/
git checkout feat/qrdqn
```

Run the benchmark (replace `$ENV_ID` by the envs mentioned above):

```
python train.py --algo qrdqn --env $ENV_ID --eval-episodes 10 --eval-freq 10000
```

Plot the results:

```
python scripts/all_plots.py -a qrdqn -e Breakout Pong -f logs/ -o logs/qrdqn_results
python scripts/plot_from_file.py -i logs/qrdqn_results.pkl -latex -l QR-DQN
```

7.5 Parameters

class sb3_contrib.qrdqn.**QRDQN**(*policy*, *env*, *learning_rate=5e-05*, *buffer_size=1000000*, *learning_starts=50000*, *batch_size=32*, *tau=1.0*, *gamma=0.99*, *train_freq=4*, *gradient_steps=1*, *replay_buffer_class=None*, *replay_buffer_kwargs=None*, *optimize_memory_usage=False*, *target_update_interval=10000*, *exploration_fraction=0.005*, *exploration_initial_eps=1.0*, *exploration_final_eps=0.01*, *max_grad_norm=None*, *stats_window_size=100*, *tensorboard_log=None*, *policy_kwargs=None*, *verbose=0*, *seed=None*, *device='auto'*, *_init_setup_model=True*)

Quantile Regression Deep Q-Network (QR-DQN) Paper: https://arxiv.org/abs/1710.10044 Default hyperparameters are taken from the paper and are tuned for Atari games.

Parameters

- **policy** (`Union[str, Type[QRDQNPolicy]]`) – The policy model to use (MlpPolicy, CnnPolicy, ...)
- **env** (`Union[Env, VecEnv, str]`) – The environment to learn from (if registered in Gym, can be str)
- **learning_rate** (`Union[float, Callable[[float], float]]`) – The learning rate, it can be a function of the current progress remaining (from 1 to 0)
- **buffer_size** (`int`) – size of the replay buffer
- **learning_starts** (`int`) – how many steps of the model to collect transitions for before learning starts
- **batch_size** (`int`) – Minibatch size for each gradient update
- **tau** (`float`) – the soft update coefficient ("Polyak update", between 0 and 1) default 1 for hard update
- **gamma** (`float`) – the discount factor
- **train_freq** (`int`) – Update the model every `train_freq` steps. Alternatively pass a tuple of frequency and unit like `(5, "step")` or `(2, "episode")`.

- **gradient_steps** (int) – How many gradient steps to do after each rollout (see train_freq and n_episodes_rollout) Set to -1 means to do as many gradient steps as steps done in the environment during the rollout.
- **replay_buffer_class** (Optional[Type[ReplayBuffer]]) – Replay buffer class to use (for instance HerReplayBuffer). If None, it will be automatically selected.
- **replay_buffer_kwargs** (Optional[Dict[str, Any]]) – Keyword arguments to pass to the replay buffer on creation.
- **optimize_memory_usage** (bool) – Enable a memory efficient variant of the replay buffer at a cost of more complexity. See https://github.com/DLR-RM/stable-baselines3/issues/37#issuecomment-637501195
- **target_update_interval** (int) – update the target network every target_update_interval environment steps.
- **exploration_fraction** (float) – fraction of entire training period over which the exploration rate is reduced
- **exploration_initial_eps** (float) – initial value of random action probability
- **exploration_final_eps** (float) – final value of random action probability
- **max_grad_norm** (Optional[float]) – The maximum value for the gradient clipping (if None, no clipping)
- **stats_window_size** (int) – Window size for the rollout logging, specifying the number of episodes to average the reported success rate, mean episode length, and mean reward over
- **tensorboard_log** (Optional[str]) – the log location for tensorboard (if None, no logging)
- **policy_kwargs** (Optional[Dict[str, Any]]) – additional arguments to be passed to the policy on creation
- **verbose** (int) – the verbosity level: 0 no output, 1 info, 2 debug
- **seed** (Optional[int]) – Seed for the pseudo random generators
- **device** (Union[device, str]) – Device (cpu, cuda, ...) on which the code should be run. Setting it to auto, the code will be run on the GPU if possible.
- **_init_setup_model** (bool) – Whether or not to build the network at the creation of the instance

collect_rollouts(*env*, *callback*, *train_freq*, *replay_buffer*, *action_noise=None*, *learning_starts=0*, *log_interval=None*)

Collect experiences and store them into a ReplayBuffer.

Parameters

- **env** (VecEnv) – The training environment
- **callback** (BaseCallback) – Callback that will be called at each step (and at the beginning and end of the rollout)
- **train_freq** (TrainFreq) – How much experience to collect by doing rollouts of current policy. Either TrainFreq(<n>, TrainFrequencyUnit.STEP) or TrainFreq(<n>, TrainFrequencyUnit.EPISODE) with <n> being an integer greater than 0.
- **action_noise** (Optional[ActionNoise]) – Action noise that will be used for exploration Required for deterministic policy (e.g. TD3). This can also be used in addition to the stochastic policy for SAC.

- **learning_starts** (int) – Number of steps before learning for the warm-up phase.
- **replay_buffer** (ReplayBuffer) –
- **log_interval** (Optional[int]) – Log data every log_interval episodes

Return type
RolloutReturn

Returns

get_env()

Returns the current environment (can be None if not defined).

Return type
Optional[VecEnv]

Returns
The current environment

get_parameters()

Return the parameters of the agent. This includes parameters from different networks, e.g. critics (value functions) and policies (pi functions).

Return type
Dict[str, Dict]

Returns
Mapping of from names of the objects to PyTorch state-dicts.

get_vec_normalize_env()

Return the VecNormalize wrapper of the training env if it exists.

Return type
Optional[VecNormalize]

Returns
The VecNormalize env.

learn(*total_timesteps*, *callback=None*, *log_interval=4*, *tb_log_name='QRDQN'*, *reset_num_timesteps=True*, *progress_bar=False*)

Return a trained model.

Parameters

- **total_timesteps** (int) – The total number of samples (env steps) to train on
- **callback** (Union[None, Callable, List[BaseCallback], BaseCallback]) – callback(s) called at every step with state of the algorithm.
- **log_interval** (int) – The number of episodes before logging.
- **tb_log_name** (str) – the name of the run for TensorBoard logging
- **reset_num_timesteps** (bool) – whether or not to reset the current timestep number (used in logging)
- **progress_bar** (bool) – Display a progress bar using tqdm and rich.

Return type
TypeVar(SelfQRDQN, bound= QRDQN)

Returns
the trained model

classmethod load(*path*, *env=None*, *device='auto'*, *custom_objects=None*, *print_system_info=False*, *force_reset=True*, ***kwargs*)

Load the model from a zip-file. Warning: `load` re-creates the model from scratch, it does not update it in-place! For an in-place load use `set_parameters` instead.

Parameters

- **path** (`Union[str, Path, BufferedIOBase]`) – path to the file (or a file-like) where to load the agent from
- **env** (`Union[Env, VecEnv, None]`) – the new environment to run the loaded model on (can be None if you only need prediction from a trained model) has priority over any saved environment
- **device** (`Union[device, str]`) – Device on which the code should run.
- **custom_objects** (`Optional[Dict[str, Any]]`) – Dictionary of objects to replace upon loading. If a variable is present in this dictionary as a key, it will not be deserialized and the corresponding item will be used instead. Similar to custom_objects in `keras.models.load_model`. Useful when you have an object in file that can not be deserialized.
- **print_system_info** (`bool`) – Whether to print system info from the saved model and the current system info (useful to debug loading issues)
- **force_reset** (`bool`) – Force call to `reset()` before training to avoid unexpected behavior. See https://github.com/DLR-RM/stable-baselines3/issues/597
- **kwargs** – extra arguments to change the model when loading

Return type
`TypeVar(SelfBaseAlgorithm, bound= BaseAlgorithm)`

Returns
new model instance with loaded parameters

load_replay_buffer(*path*, *truncate_last_traj=True*)

Load a replay buffer from a pickle file.

Parameters

- **path** (`Union[str, Path, BufferedIOBase]`) – Path to the pickled replay buffer.
- **truncate_last_traj** (`bool`) – When using `HerReplayBuffer` with online sampling: If set to `True`, we assume that the last trajectory in the replay buffer was finished (and truncate it). If set to `False`, we assume that we continue the same trajectory (same episode).

Return type
`None`

property logger: Logger

Getter for the logger object.

predict(*observation*, *state=None*, *episode_start=None*, *deterministic=False*)

Get the policy action from an observation (and optional hidden state). Includes sugar-coating to handle different observations (e.g. normalizing images).

Parameters

- **observation** (`Union[ndarray, Dict[str, ndarray]]`) – the input observation
- **state** (`Optional[Tuple[ndarray, ...]]`) – The last hidden states (can be None, used in recurrent policies)

- **episode_start** (Optional[ndarray]) – The last masks (can be None, used in recurrent policies) this correspond to beginning of episodes, where the hidden states of the RNN must be reset.
- **deterministic** (bool) – Whether or not to return deterministic actions.

Return type
Tuple[ndarray, Optional[Tuple[ndarray, ...]]]

Returns
the model's action and the next hidden state (used in recurrent policies)

save(*path*, *exclude=None*, *include=None*)

Save all the attributes of the object and the model parameters in a zip-file.

Parameters

- **path** (Union[str, Path, BufferedIOBase]) – path to the file where the rl agent should be saved
- **exclude** (Optional[Iterable[str]]) – name of parameters that should be excluded in addition to the default ones
- **include** (Optional[Iterable[str]]) – name of parameters that might be excluded but should be included anyway

Return type
None

save_replay_buffer(*path*)

Save the replay buffer as a pickle file.

Parameters
path (Union[str, Path, BufferedIOBase]) – Path to the file where the replay buffer should be saved. if path is a str or pathlib.Path, the path is automatically created if necessary.

Return type
None

set_env(*env*, *force_reset=True*)

Checks the validity of the environment, and if it is coherent, set it as the current environment. Furthermore wrap any non vectorized env into a vectorized checked parameters: - observation_space - action_space

Parameters

- **env** (Union[Env, VecEnv]) – The environment for learning a policy
- **force_reset** (bool) – Force call to `reset()` before training to avoid unexpected behavior. See issue https://github.com/DLR-RM/stable-baselines3/issues/597

Return type
None

set_logger(*logger*)

Setter for for logger object. :rtype: None

> **Warning:** When passing a custom logger object, this will overwrite `tensorboard_log` and `verbose` settings passed to the constructor.

set_parameters(*load_path_or_dict*, *exact_match=True*, *device='auto'*)

Load parameters from a given zip-file or a nested dictionary containing parameters for different modules (see get_parameters).

Parameters

- **load_path_or_iter** – Location of the saved data (path or file-like, see save), or a nested dictionary containing nn.Module parameters used by the policy. The dictionary maps object names to a state-dictionary returned by torch.nn.Module.state_dict().
- **exact_match** (bool) – If True, the given parameters should include parameters for each module and each of their parameters, otherwise raises an Exception. If set to False, this can be used to update only specific parameters.
- **device** (Union[device, str]) – Device on which the code should run.

Return type
None

set_random_seed(*seed=None*)

Set the seed of the pseudo-random generators (python, numpy, pytorch, gym, action_space)

Parameters
seed (Optional[int]) –

Return type
None

train(*gradient_steps*, *batch_size=100*)

Sample the replay buffer and do the updates (gradient descent and update target networks)

Return type
None

7.6 QR-DQN Policies

sb3_contrib.qrdqn.**MlpPolicy**

alias of QRDQNPolicy

class sb3_contrib.qrdqn.policies.**QRDQNPolicy**(*observation_space*, *action_space*, *lr_schedule*, *n_quantiles=200*, *net_arch=None*, *activation_fn=<class 'torch.nn.modules.activation.ReLU'>*, *features_extractor_class=<class 'stable_baselines3.common.torch_layers.FlattenExtractor'>*, *features_extractor_kwargs=None*, *normalize_images=True*, *optimizer_class=<class 'torch.optim.adam.Adam'>*, *optimizer_kwargs=None*)

Policy class with quantile and target networks for QR-DQN.

Parameters

- **observation_space** (Space) – Observation space
- **action_space** (Space) – Action space
- **lr_schedule** (Callable[[float], float]) – Learning rate schedule (could be constant)
- **n_quantiles** (int) – Number of quantiles

- **net_arch** (Optional[List[int]]) – The specification of the network architecture.
- **activation_fn** (Type[Module]) – Activation function
- **features_extractor_class** (Type[BaseFeaturesExtractor]) – Features extractor to use.
- **features_extractor_kwargs** (Optional[Dict[str, Any]]) – Keyword arguments to pass to the features extractor.
- **normalize_images** (bool) – Whether to normalize images or not, dividing by 255.0 (True by default)
- **optimizer_class** (Type[Optimizer]) – The optimizer to use, th.optim.Adam by default
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional keyword arguments, excluding the learning rate, to pass to the optimizer

forward(*obs*, *deterministic=True*)

Defines the computation performed at every call.

Should be overridden by all subclasses. :rtype: Tensor

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

set_training_mode(*mode*)

Put the policy in either training or evaluation mode. This affects certain modules, such as batch normalisation and dropout. :type mode: bool :param mode: if true, set to training mode, else set to evaluation mode

Return type
None

class sb3_contrib.qrdqn.**CnnPolicy**(*observation_space*, *action_space*, *lr_schedule*, *n_quantiles=200*, *net_arch=None*, *activation_fn=<class 'torch.nn.modules.activation.ReLU'>*, *features_extractor_class=<class 'stable_baselines3.common.torch_layers.NatureCNN'>*, *features_extractor_kwargs=None*, *normalize_images=True*, *optimizer_class=<class 'torch.optim.adam.Adam'>*, *optimizer_kwargs=None*)

Policy class for QR-DQN when using images as input.

Parameters

- **observation_space** (Space) – Observation space
- **action_space** (Space) – Action space
- **lr_schedule** (Callable[[float], float]) – Learning rate schedule (could be constant)
- **n_quantiles** (int) – Number of quantiles
- **net_arch** (Optional[List[int]]) – The specification of the network architecture.
- **activation_fn** (Type[Module]) – Activation function
- **features_extractor_class** (Type[BaseFeaturesExtractor]) – Features extractor to use.

- **normalize_images** (bool) – Whether to normalize images or not, dividing by 255.0 (True by default)
- **optimizer_class** (Type[Optimizer]) – The optimizer to use, th.optim.Adam by default
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional keyword arguments, excluding the learning rate, to pass to the optimizer

class sb3_contrib.qrdqn.**MultiInputPolicy**(*observation_space*, *action_space*, *lr_schedule*, *n_quantiles=200*, *net_arch=None*, *activation_fn=<class 'torch.nn.modules.activation.ReLU'>*, *features_extractor_class=<class 'stable_baselines3.common.torch_layers.CombinedExtractor'>*, *features_extractor_kwargs=None*, *normalize_images=True*, *optimizer_class=<class 'torch.optim.adam.Adam'>*, *optimizer_kwargs=None*)

Policy class for QR-DQN when using dict observations as input.

Parameters

- **observation_space** (Space) – Observation space
- **action_space** (Space) – Action space
- **lr_schedule** (Callable[[float], float]) – Learning rate schedule (could be constant)
- **n_quantiles** (int) – Number of quantiles
- **net_arch** (Optional[List[int]]) – The specification of the network architecture.
- **activation_fn** (Type[Module]) – Activation function
- **features_extractor_class** (Type[BaseFeaturesExtractor]) – Features extractor to use.
- **normalize_images** (bool) – Whether to normalize images or not, dividing by 255.0 (True by default)
- **optimizer_class** (Type[Optimizer]) – The optimizer to use, th.optim.Adam by default
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional keyword arguments, excluding the learning rate, to pass to the optimizer

CHAPTER

EIGHT

TQC

Controlling Overestimation Bias with Truncated Mixture of Continuous Distributional Quantile Critics (TQC). Truncated Quantile Critics (TQC) builds on SAC, TD3 and QR-DQN, making use of quantile regression to predict a distribution for the value function (instead of a mean value). It truncates the quantiles predicted by different networks (a bit as it is done in TD3).

Available Policies

MlpPolicy	alias of `TQCPolicy`
CnnPolicy	Policy class (with both actor and critic) for TQC.
`MultiInputPolicy`	Policy class (with both actor and critic) for TQC.

8.1 Notes

- Original paper: https://arxiv.org/abs/2005.04269
- Original Implementation: https://github.com/bayesgroup/tqc_pytorch

8.2 Can I use?

- Recurrent policies:
- Multi processing: ✓
- Gym spaces:

Space	Action	Observation
Discrete		✓
Box	✓	✓
MultiDiscrete		✓
MultiBinary		✓
Dict		✓

8.3 Example

```
import gym
import numpy as np

from sb3_contrib import TQC

env = gym.make("Pendulum-v1")

policy_kwargs = dict(n_critics=2, n_quantiles=25)
model = TQC("MlpPolicy", env, top_quantiles_to_drop_per_net=2, verbose=1, policy_
↪kwargs=policy_kwargs)
model.learn(total_timesteps=10000, log_interval=4)
model.save("tqc_pendulum")

del model # remove to demonstrate saving and loading

model = TQC.load("tqc_pendulum")

obs = env.reset()
while True:
    action, _states = model.predict(obs, deterministic=True)
    obs, reward, done, info = env.step(action)
    env.render()
    if done:
      obs = env.reset()
```

8.4 Results

Result on the PyBullet benchmark (1M steps) and on BipedalWalkerHardcore-v3 (2M steps) using 3 seeds. The complete learning curves are available in the associated PR.

The main difference with SAC is on harder environments (BipedalWalkerHardcore, Walker2D).

Note: Hyperparameters from the gSDE paper were used (as they are tuned for SAC on PyBullet envs), including using gSDE for the exploration and not the unstructured Gaussian noise but this should not affect results in simulation.

Note: We are using the open source PyBullet environments and not the MuJoCo simulator (as done in the original paper). You can find a complete benchmark on PyBullet envs in the gSDE paper if you want to compare TQC results to those of A2C/PPO/SAC/TD3.

Environments	SAC	TQC
	gSDE	gSDE
HalfCheetah	2984 +/- 202	3041 +/- 157
Ant	3102 +/- 37	3700 +/- 37
Hopper	2262 +/- 1	2401 +/- 62*
Walker2D	2136 +/- 67	2535 +/- 94
BipedalWalkerHardcore	13 +/- 18	228 +/- 18

* with tuned hyperparameter `top_quantiles_to_drop_per_net` taken from the original paper

8.4.1 How to replicate the results?

Clone RL-Zoo and checkout the branch `feat/tqc`:

```
git clone https://github.com/DLR-RM/rl-baselines3-zoo
cd rl-baselines3-zoo/
git checkout feat/tqc
```

Run the benchmark (replace `$ENV_ID` by the envs mentioned above):

```
python train.py --algo tqc --env $ENV_ID --eval-episodes 10 --eval-freq 10000
```

Plot the results:

```
python scripts/all_plots.py -a tqc -e HalfCheetah Ant Hopper Walker2D␣
↪BipedalWalkerHardcore -f logs/ -o logs/tqc_results
python scripts/plot_from_file.py -i logs/tqc_results.pkl -latex -l TQC
```

8.5 Comments

This implementation is based on SB3 SAC implementation and uses the code from the original TQC implementation for the quantile huber loss.

8.6 Parameters

class sb3_contrib.tqc.**TQC**(*policy*, *env*, *learning_rate=0.0003*, *buffer_size=1000000*, *learning_starts=100*, *batch_size=256*, *tau=0.005*, *gamma=0.99*, *train_freq=1*, *gradient_steps=1*, *action_noise=None*, *replay_buffer_class=None*, *replay_buffer_kwargs=None*, *optimize_memory_usage=False*, *ent_coef='auto'*, *target_update_interval=1*, *target_entropy='auto'*, *top_quantiles_to_drop_per_net=2*, *use_sde=False*, *sde_sample_freq=-1*, *use_sde_at_warmup=False*, *stats_window_size=100*, *tensorboard_log=None*, *policy_kwargs=None*, *verbose=0*, *seed=None*, *device='auto'*, *_init_setup_model=True*)

Controlling Overestimation Bias with Truncated Mixture of Continuous Distributional Quantile Critics. Paper: https://arxiv.org/abs/2005.04269 This implementation uses SB3 SAC implementation as base.

Parameters

- **policy** (Union[str, Type[TQCPolicy]]) – The policy model to use (MlpPolicy, CnnPolicy, ...)
- **env** (Union[Env, VecEnv, str]) – The environment to learn from (if registered in Gym, can be str)
- **learning_rate** (Union[float, Callable]) – learning rate for adam optimizer, the same learning rate will be used for all networks (Q-Values, Actor and Value function) it can be a function of the current progress remaining (from 1 to 0)
- **buffer_size** (int) – size of the replay buffer
- **learning_starts** (int) – how many steps of the model to collect transitions for before learning starts
- **batch_size** (int) – Minibatch size for each gradient update
- **tau** (float) – the soft update coefficient ("Polyak update", between 0 and 1)
- **gamma** (float) – the discount factor
- **train_freq** (int) – Update the model every train_freq steps. Alternatively pass a tuple of frequency and unit like (5, "step") or (2, "episode").
- **gradient_steps** (int) – How many gradient update after each step
- **action_noise** (Optional[ActionNoise]) – the action noise type (None by default), this can help for hard exploration problem. Cf common.noise for the different action noise type.
- **replay_buffer_class** (Optional[ReplayBuffer]) – Replay buffer class to use (for instance HerReplayBuffer). If None, it will be automatically selected.
- **replay_buffer_kwargs** (Optional[Dict[str, Any]]) – Keyword arguments to pass to the replay buffer on creation.
- **optimize_memory_usage** (bool) – Enable a memory efficient variant of the replay buffer at a cost of more complexity. See https://github.com/DLR-RM/stable-baselines3/issues/37#issuecomment-637501195
- **ent_coef** (Union[str, float]) – Entropy regularization coefficient. (Equivalent to inverse of reward scale in the original SAC paper.) Controlling exploration/exploitation trade-off. Set it to 'auto' to learn it automatically (and 'auto_0.1' for using 0.1 as initial value)
- **target_update_interval** (int) – update the target network every target_network_update_freq gradient steps.
- **target_entropy** (Union[str, float]) – target entropy when learning ent_coef (ent_coef = 'auto')
- **top_quantiles_to_drop_per_net** (int) – Number of quantiles to drop per network
- **use_sde** (bool) – Whether to use generalized State Dependent Exploration (gSDE) instead of action noise exploration (default: False)
- **sde_sample_freq** (int) – Sample a new noise matrix every n steps when using gSDE Default: -1 (only sample at the beginning of the rollout)
- **use_sde_at_warmup** (bool) – Whether to use gSDE instead of uniform sampling during the warm up phase (before learning starts)
- **stats_window_size** (int) – Window size for the rollout logging, specifying the number of episodes to average the reported success rate, mean episode length, and mean reward over
- **tensorboard_log** (Optional[str]) – the log location for tensorboard (if None, no logging)

- **policy_kwargs** (Optional[Dict[str, Any]]) – additional arguments to be passed to the policy on creation
- **verbose** (int) – the verbosity level: 0 no output, 1 info, 2 debug
- **seed** (Optional[int]) – Seed for the pseudo random generators
- **device** (Union[device, str]) – Device (cpu, cuda, ...) on which the code should be run. Setting it to auto, the code will be run on the GPU if possible.
- **_init_setup_model** (bool) – Whether or not to build the network at the creation of the instance

collect_rollouts(*env*, *callback*, *train_freq*, *replay_buffer*, *action_noise=None*, *learning_starts=0*, *log_interval=None*)

Collect experiences and store them into a ReplayBuffer.

Parameters

- **env** (VecEnv) – The training environment
- **callback** (BaseCallback) – Callback that will be called at each step (and at the beginning and end of the rollout)
- **train_freq** (TrainFreq) – How much experience to collect by doing rollouts of current policy. Either TrainFreq(<n>, TrainFrequencyUnit.STEP) or TrainFreq(<n>, TrainFrequencyUnit.EPISODE) with <n> being an integer greater than 0.
- **action_noise** (Optional[ActionNoise]) – Action noise that will be used for exploration Required for deterministic policy (e.g. TD3). This can also be used in addition to the stochastic policy for SAC.
- **learning_starts** (int) – Number of steps before learning for the warm-up phase.
- **replay_buffer** (ReplayBuffer) –
- **log_interval** (Optional[int]) – Log data every log_interval episodes

Return type
RolloutReturn

Returns

get_env()

Returns the current environment (can be None if not defined).

Return type
Optional[VecEnv]

Returns
The current environment

get_parameters()

Return the parameters of the agent. This includes parameters from different networks, e.g. critics (value functions) and policies (pi functions).

Return type
Dict[str, Dict]

Returns
Mapping of from names of the objects to PyTorch state-dicts.

`get_vec_normalize_env()`

Return the `VecNormalize` wrapper of the training env if it exists.

Return type
`Optional[VecNormalize]`

Returns
The `VecNormalize` env.

`learn(total_timesteps, callback=None, log_interval=4, tb_log_name='TQC', reset_num_timesteps=True, progress_bar=False)`

Return a trained model.

Parameters

- **total_timesteps** (`int`) – The total number of samples (env steps) to train on
- **callback** (`Union[None, Callable, List[BaseCallback], BaseCallback]`) – callback(s) called at every step with state of the algorithm.
- **log_interval** (`int`) – The number of episodes before logging.
- **tb_log_name** (`str`) – the name of the run for TensorBoard logging
- **reset_num_timesteps** (`bool`) – whether or not to reset the current timestep number (used in logging)
- **progress_bar** (`bool`) – Display a progress bar using tqdm and rich.

Return type
`TypeVar(SelfTQC, bound= TQC)`

Returns
the trained model

`classmethod load(path, env=None, device='auto', custom_objects=None, print_system_info=False, force_reset=True, **kwargs)`

Load the model from a zip-file. Warning: `load` re-creates the model from scratch, it does not update it in-place! For an in-place load use `set_parameters` instead.

Parameters

- **path** (`Union[str, Path, BufferedIOBase]`) – path to the file (or a file-like) where to load the agent from
- **env** (`Union[Env, VecEnv, None]`) – the new environment to run the loaded model on (can be None if you only need prediction from a trained model) has priority over any saved environment
- **device** (`Union[device, str]`) – Device on which the code should run.
- **custom_objects** (`Optional[Dict[str, Any]]`) – Dictionary of objects to replace upon loading. If a variable is present in this dictionary as a key, it will not be deserialized and the corresponding item will be used instead. Similar to custom_objects in `keras.models.load_model`. Useful when you have an object in file that can not be deserialized.
- **print_system_info** (`bool`) – Whether to print system info from the saved model and the current system info (useful to debug loading issues)
- **force_reset** (`bool`) – Force call to `reset()` before training to avoid unexpected behavior. See https://github.com/DLR-RM/stable-baselines3/issues/597
- **kwargs** – extra arguments to change the model when loading

Return type
TypeVar(SelfBaseAlgorithm, bound= BaseAlgorithm)

Returns
new model instance with loaded parameters

load_replay_buffer(*path*, *truncate_last_traj=True*)

Load a replay buffer from a pickle file.

Parameters

- **path** (Union[str, Path, BufferedIOBase]) – Path to the pickled replay buffer.
- **truncate_last_traj** (bool) – When using HerReplayBuffer with online sampling: If set to True, we assume that the last trajectory in the replay buffer was finished (and truncate it). If set to False, we assume that we continue the same trajectory (same episode).

Return type
None

property logger: Logger

Getter for the logger object.

predict(*observation*, *state=None*, *episode_start=None*, *deterministic=False*)

Get the policy action from an observation (and optional hidden state). Includes sugar-coating to handle different observations (e.g. normalizing images).

Parameters

- **observation** (Union[ndarray, Dict[str, ndarray]]) – the input observation
- **state** (Optional[Tuple[ndarray, ...]]) – The last hidden states (can be None, used in recurrent policies)
- **episode_start** (Optional[ndarray]) – The last masks (can be None, used in recurrent policies) this correspond to beginning of episodes, where the hidden states of the RNN must be reset.
- **deterministic** (bool) – Whether or not to return deterministic actions.

Return type
Tuple[ndarray, Optional[Tuple[ndarray, ...]]]

Returns
the model's action and the next hidden state (used in recurrent policies)

save(*path*, *exclude=None*, *include=None*)

Save all the attributes of the object and the model parameters in a zip-file.

Parameters

- **path** (Union[str, Path, BufferedIOBase]) – path to the file where the rl agent should be saved
- **exclude** (Optional[Iterable[str]]) – name of parameters that should be excluded in addition to the default ones
- **include** (Optional[Iterable[str]]) – name of parameters that might be excluded but should be included anyway

Return type
None

save_replay_buffer(*path*)

Save the replay buffer as a pickle file.

Parameters
path (Union[str, Path, BufferedIOBase]) – Path to the file where the replay buffer should be saved. if path is a str or pathlib.Path, the path is automatically created if necessary.

Return type
None

set_env(*env*, *force_reset=True*)

Checks the validity of the environment, and if it is coherent, set it as the current environment. Furthermore wrap any non vectorized env into a vectorized checked parameters: - observation_space - action_space

Parameters

- **env** (Union[Env, VecEnv]) – The environment for learning a policy
- **force_reset** (bool) – Force call to `reset()` before training to avoid unexpected behavior. See issue https://github.com/DLR-RM/stable-baselines3/issues/597

Return type
None

set_logger(*logger*)

Setter for for logger object. :rtype: None

> **Warning:** When passing a custom logger object, this will overwrite `tensorboard_log` and `verbose` settings passed to the constructor.

set_parameters(*load_path_or_dict*, *exact_match=True*, *device='auto'*)

Load parameters from a given zip-file or a nested dictionary containing parameters for different modules (see `get_parameters`).

Parameters

- **load_path_or_iter** – Location of the saved data (path or file-like, see `save`), or a nested dictionary containing nn.Module parameters used by the policy. The dictionary maps object names to a state-dictionary returned by `torch.nn.Module.state_dict()`.
- **exact_match** (bool) – If True, the given parameters should include parameters for each module and each of their parameters, otherwise raises an Exception. If set to False, this can be used to update only specific parameters.
- **device** (Union[device, str]) – Device on which the code should run.

Return type
None

set_random_seed(*seed=None*)

Set the seed of the pseudo-random generators (python, numpy, pytorch, gym, action_space)

Parameters
seed (Optional[int]) –

Return type
None

train(*gradient_steps*, *batch_size=64*)

Sample the replay buffer and do the updates (gradient descent and update target networks)

Return type
None

8.7 TQC Policies

sb3_contrib.tqc.**MlpPolicy**

alias of TQCPolicy

class sb3_contrib.tqc.policies.**TQCPolicy**(*observation_space*, *action_space*, *lr_schedule*, *net_arch=None*, *activation_fn=<class 'torch.nn.modules.activation.ReLU'>*, *use_sde=False*, *log_std_init=-3*, *use_expln=False*, *clip_mean=2.0*, *features_extractor_class=<class 'stable_baselines3.common.torch_layers.FlattenExtractor'>*, *features_extractor_kwargs=None*, *normalize_images=True*, *optimizer_class=<class 'torch.optim.adam.Adam'>*, *optimizer_kwargs=None*, *n_quantiles=25*, *n_critics=2*, *share_features_extractor=False*)

Policy class (with both actor and critic) for TQC.

Parameters

- **observation_space** (Space) – Observation space
- **action_space** (Space) – Action space
- **lr_schedule** (Callable[[float], float]) – Learning rate schedule (could be constant)
- **net_arch** (Union[List[int], Dict[str, List[int]], None]) – The specification of the policy and value networks.
- **activation_fn** (Type[Module]) – Activation function
- **use_sde** (bool) – Whether to use State Dependent Exploration or not
- **log_std_init** (float) – Initial value for the log standard deviation
- **use_expln** (bool) – Use expln() function instead of exp() when using gSDE to ensure a positive standard deviation (cf paper). It allows to keep variance above zero and prevent it from growing too fast. In practice, exp() is usually enough.
- **clip_mean** (float) – Clip the mean output when using gSDE to avoid numerical instability.
- **features_extractor_class** (Type[BaseFeaturesExtractor]) – Features extractor to use.
- **features_extractor_kwargs** (Optional[Dict[str, Any]]) – Keyword arguments to pass to the feature extractor.
- **normalize_images** (bool) – Whether to normalize images or not, dividing by 255.0 (True by default)
- **optimizer_class** (Type[Optimizer]) – The optimizer to use, th.optim.Adam by default
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional keyword arguments, excluding the learning rate, to pass to the optimizer
- **n_quantiles** (int) – Number of quantiles for the critic.

- **n_critics** (int) – Number of critic networks to create.
- **share_features_extractor** (bool) – Whether to share or not the features extractor between the actor and the critic (this saves computation time)

forward(*obs*, *deterministic=False*)

Defines the computation performed at every call.

Should be overridden by all subclasses. :rtype: Tensor

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

reset_noise(*batch_size=1*)

Sample new weights for the exploration matrix, when using gSDE.

Parameters
batch_size (int) –

Return type
None

set_training_mode(*mode*)

Put the policy in either training or evaluation mode. This affects certain modules, such as batch normalisation and dropout. :type mode: bool :param mode: if true, set to training mode, else set to evaluation mode

Return type
None

class sb3_contrib.tqc.**CnnPolicy**(*observation_space*, *action_space*, *lr_schedule*, *net_arch=None*, *activation_fn=<class 'torch.nn.modules.activation.ReLU'>*, *use_sde=False*, *log_std_init=-3*, *use_expln=False*, *clip_mean=2.0*, *features_extractor_class=<class 'stable_baselines3.common.torch_layers.NatureCNN'>*, *features_extractor_kwargs=None*, *normalize_images=True*, *optimizer_class=<class 'torch.optim.adam.Adam'>*, *optimizer_kwargs=None*, *n_quantiles=25*, *n_critics=2*, *share_features_extractor=False*)

Policy class (with both actor and critic) for TQC.

Parameters

- **observation_space** (Space) – Observation space
- **action_space** (Space) – Action space
- **lr_schedule** (Callable[[float], float]) – Learning rate schedule (could be constant)
- **net_arch** (Union[List[int], Dict[str, List[int]], None]) – The specification of the policy and value networks.
- **activation_fn** (Type[Module]) – Activation function
- **use_sde** (bool) – Whether to use State Dependent Exploration or not
- **log_std_init** (float) – Initial value for the log standard deviation

- **use_expln** (bool) – Use expln() function instead of exp() when using gSDE to ensure a positive standard deviation (cf paper). It allows to keep variance above zero and prevent it from growing too fast. In practice, exp() is usually enough.
- **clip_mean** (float) – Clip the mean output when using gSDE to avoid numerical instability.
- **features_extractor_class** (Type[BaseFeaturesExtractor]) – Features extractor to use.
- **normalize_images** (bool) – Whether to normalize images or not, dividing by 255.0 (True by default)
- **optimizer_class** (Type[Optimizer]) – The optimizer to use, th.optim.Adam by default
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional keyword arguments, excluding the learning rate, to pass to the optimizer
- **n_quantiles** (int) – Number of quantiles for the critic.
- **n_critics** (int) – Number of critic networks to create.
- **share_features_extractor** (bool) – Whether to share or not the features extractor between the actor and the critic (this saves computation time)

CHAPTER

NINE

TRPO

Trust Region Policy Optimization (TRPO) is an iterative approach for optimizing policies with guaranteed monotonic improvement.

Available Policies

MlpPolicy	alias of `ActorCriticPolicy`
CnnPolicy	alias of `ActorCriticCnnPolicy`
MultiInputPolicy	alias of `MultiInputActorCriticPolicy`

9.1 Notes

- Original paper: https://arxiv.org/abs/1502.05477
- OpenAI blog post: https://blog.openai.com/openai-baselines-ppo/

9.2 Can I use?

- Recurrent policies:
- Multi processing: ✓
- Gym spaces:

Space	Action	Observation
Discrete	✓	✓
Box	✓	✓
MultiDiscrete	✓	✓
MultiBinary	✓	✓
Dict		✓

9.3 Example

```
import gym
import numpy as np

from sb3_contrib import TRPO

env = gym.make("Pendulum-v1")

model = TRPO("MlpPolicy", env, verbose=1)
model.learn(total_timesteps=10000, log_interval=4)
model.save("trpo_pendulum")

del model # remove to demonstrate saving and loading

model = TRPO.load("trpo_pendulum")

obs = env.reset()
while True:
    action, _states = model.predict(obs, deterministic=True)
    obs, reward, done, info = env.step(action)
    env.render()
    if done:
      obs = env.reset()
```

9.4 Results

Result on the MuJoCo benchmark (1M steps on `-v3` envs with MuJoCo v2.1.0) using 3 seeds. The complete learning curves are available in the associated PR.

Environments	TRPO
HalfCheetah	1803 +/- 46
Ant	3554 +/- 591
Hopper	3372 +/- 215
Walker2d	4502 +/- 234
Swimmer	359 +/- 2

9.4.1 How to replicate the results?

Clone RL-Zoo and checkout the branch `feat/trpo`:

```
git clone https://github.com/cyprienc/rl-baselines3-zoo
cd rl-baselines3-zoo/
```

Run the benchmark (replace `$ENV_ID` by the envs mentioned above):

```
python train.py --algo tqc --env $ENV_ID --n-eval-envs 10 --eval-episodes 20 --eval-freq␣
↪50000
```

Plot the results:

```
python scripts/all_plots.py -a trpo -e HalfCheetah Ant Hopper Walker2d Swimmer -f logs/ -
↪o logs/trpo_results
python scripts/plot_from_file.py -i logs/trpo_results.pkl -latex -l TRPO
```

9.5 Parameters

class sb3_contrib.trpo.**TRPO**(*policy*, *env*, *learning_rate=0.001*, *n_steps=2048*, *batch_size=128*, *gamma=0.99*, *cg_max_steps=15*, *cg_damping=0.1*, *line_search_shrinking_factor=0.8*, *line_search_max_iter=10*, *n_critic_updates=10*, *gae_lambda=0.95*, *use_sde=False*, *sde_sample_freq=-1*, *normalize_advantage=True*, *target_kl=0.01*, *sub_sampling_factor=1*, *stats_window_size=100*, *tensorboard_log=None*, *policy_kwargs=None*, *verbose=0*, *seed=None*, *device='auto'*, *_init_setup_model=True*)

Trust Region Policy Optimization (TRPO)

Paper: https://arxiv.org/abs/1502.05477 Code: This implementation borrows code from OpenAI Spinning Up (https://github.com/openai/spinningup/) and Stable Baselines (TRPO from https://github.com/hill-a/stable-baselines)

Introduction to TRPO: https://spinningup.openai.com/en/latest/algorithms/trpo.html

Parameters

- **policy** (Union[str, Type[ActorCriticPolicy]]) – The policy model to use (MlpPolicy, CnnPolicy, ...)
- **env** (Union[Env, VecEnv, str]) – The environment to learn from (if registered in Gym, can be str)
- **learning_rate** (Union[float, Callable[[float], float]]) – The learning rate for the value function, it can be a function of the current progress remaining (from 1 to 0)
- **n_steps** (int) – The number of steps to run for each environment per update (i.e. rollout buffer size is n_steps * n_envs where n_envs is number of environment copies running in parallel) NOTE: n_steps * n_envs must be greater than 1 (because of the advantage normalization) See https://github.com/pytorch/pytorch/issues/29372
- **batch_size** (int) – Minibatch size for the value function
- **gamma** (float) – Discount factor
- **cg_max_steps** (int) – maximum number of steps in the Conjugate Gradient algorithm for computing the Hessian vector product
- **cg_damping** (float) – damping in the Hessian vector product computation
- **line_search_shrinking_factor** (float) – step-size reduction factor for the line-search (i.e., theta_new = theta + alpha^i * step)
- **line_search_max_iter** (int) – maximum number of iteration for the backtracking line-search
- **n_critic_updates** (int) – number of critic updates per policy update
- **gae_lambda** (float) – Factor for trade-off of bias vs variance for Generalized Advantage Estimator
- **use_sde** (bool) – Whether to use generalized State Dependent Exploration (gSDE) instead of action noise exploration (default: False)

- **sde_sample_freq** (int) – Sample a new noise matrix every n steps when using gSDE Default: -1 (only sample at the beginning of the rollout)
- **normalize_advantage** (bool) – Whether to normalize or not the advantage
- **target_kl** (float) – Target Kullback-Leibler divergence between updates. Should be small for stability. Values like 0.01, 0.05.
- **sub_sampling_factor** (int) – Sub-sample the batch to make computation faster see p40-42 of John Schulman thesis http://joschu.net/docs/thesis.pdf
- **stats_window_size** (int) – Window size for the rollout logging, specifying the number of episodes to average the reported success rate, mean episode length, and mean reward over
- **tensorboard_log** (Optional[str]) – the log location for tensorboard (if None, no logging)
- **policy_kwargs** (Optional[Dict[str, Any]]) – additional arguments to be passed to the policy on creation
- **verbose** (int) – the verbosity level: 0 no output, 1 info, 2 debug
- **seed** (Optional[int]) – Seed for the pseudo random generators
- **device** (Union[device, str]) – Device (cpu, cuda, ...) on which the code should be run. Setting it to auto, the code will be run on the GPU if possible.
- **_init_setup_model** (bool) – Whether or not to build the network at the creation of the instance

collect_rollouts(*env*, *callback*, *rollout_buffer*, *n_rollout_steps*)

Collect experiences using the current policy and fill a RolloutBuffer. The term rollout here refers to the model-free notion and should not be used with the concept of rollout used in model-based RL or planning.

Parameters

- **env** (VecEnv) – The training environment
- **callback** (BaseCallback) – Callback that will be called at each step (and at the beginning and end of the rollout)
- **rollout_buffer** (RolloutBuffer) – Buffer to fill with rollouts
- **n_rollout_steps** (int) – Number of experiences to collect per environment

Return type
bool

Returns
True if function returned with at least *n_rollout_steps* collected, False if callback terminated rollout prematurely.

get_env()

Returns the current environment (can be None if not defined).

Return type
Optional[VecEnv]

Returns
The current environment

get_parameters()

Return the parameters of the agent. This includes parameters from different networks, e.g. critics (value functions) and policies (pi functions).

Return type
Dict[str, Dict]

Returns
Mapping of from names of the objects to PyTorch state-dicts.

get_vec_normalize_env()

Return the VecNormalize wrapper of the training env if it exists.

Return type
Optional[VecNormalize]

Returns
The VecNormalize env.

hessian_vector_product(*params*, *grad_kl*, *vector*, *retain_graph=True*)

Computes the matrix-vector product with the Fisher information matrix.

Parameters

- **params** (List[Parameter]) – list of parameters used to compute the Hessian
- **grad_kl** (Tensor) – flattened gradient of the KL divergence between the old and new policy
- **vector** (Tensor) – vector to compute the dot product the hessian-vector dot product with
- **retain_graph** (bool) – if True, the graph will be kept after computing the Hessian

Return type
Tensor

Returns
Hessian-vector dot product (with damping)

learn(*total_timesteps*, *callback=None*, *log_interval=1*, *tb_log_name='TRPO'*, *reset_num_timesteps=True*, *progress_bar=False*)

Return a trained model.

Parameters

- **total_timesteps** (int) – The total number of samples (env steps) to train on
- **callback** (Union[None, Callable, List[BaseCallback], BaseCallback]) – callback(s) called at every step with state of the algorithm.
- **log_interval** (int) – The number of episodes before logging.
- **tb_log_name** (str) – the name of the run for TensorBoard logging
- **reset_num_timesteps** (bool) – whether or not to reset the current timestep number (used in logging)
- **progress_bar** (bool) – Display a progress bar using tqdm and rich.

Return type
TypeVar(SelfTRPO, bound= TRPO)

Returns
the trained model

classmethod load(*path*, *env=None*, *device='auto'*, *custom_objects=None*, *print_system_info=False*, *force_reset=True*, ***kwargs*)

Load the model from a zip-file. Warning: `load` re-creates the model from scratch, it does not update it in-place! For an in-place load use `set_parameters` instead.

Parameters

- **path** (Union[str, Path, BufferedIOBase]) – path to the file (or a file-like) where to load the agent from
- **env** (Union[Env, VecEnv, None]) – the new environment to run the loaded model on (can be None if you only need prediction from a trained model) has priority over any saved environment
- **device** (Union[device, str]) – Device on which the code should run.
- **custom_objects** (Optional[Dict[str, Any]]) – Dictionary of objects to replace upon loading. If a variable is present in this dictionary as a key, it will not be deserialized and the corresponding item will be used instead. Similar to custom_objects in `keras.models.load_model`. Useful when you have an object in file that can not be deserialized.
- **print_system_info** (bool) – Whether to print system info from the saved model and the current system info (useful to debug loading issues)
- **force_reset** (bool) – Force call to `reset()` before training to avoid unexpected behavior. See https://github.com/DLR-RM/stable-baselines3/issues/597
- **kwargs** – extra arguments to change the model when loading

Return type
TypeVar(SelfBaseAlgorithm, bound= BaseAlgorithm)

Returns
new model instance with loaded parameters

property logger: Logger

Getter for the logger object.

predict(*observation*, *state=None*, *episode_start=None*, *deterministic=False*)

Get the policy action from an observation (and optional hidden state). Includes sugar-coating to handle different observations (e.g. normalizing images).

Parameters

- **observation** (Union[ndarray, Dict[str, ndarray]]) – the input observation
- **state** (Optional[Tuple[ndarray, ...]]) – The last hidden states (can be None, used in recurrent policies)
- **episode_start** (Optional[ndarray]) – The last masks (can be None, used in recurrent policies) this correspond to beginning of episodes, where the hidden states of the RNN must be reset.
- **deterministic** (bool) – Whether or not to return deterministic actions.

Return type
Tuple[ndarray, Optional[Tuple[ndarray, ...]]]

Returns
the model's action and the next hidden state (used in recurrent policies)

save(*path*, *exclude=None*, *include=None*)

Save all the attributes of the object and the model parameters in a zip-file.

Parameters

- **path** (Union[str, Path, BufferedIOBase]) – path to the file where the rl agent should be saved
- **exclude** (Optional[Iterable[str]]) – name of parameters that should be excluded in addition to the default ones
- **include** (Optional[Iterable[str]]) – name of parameters that might be excluded but should be included anyway

Return type
None

set_env(*env*, *force_reset=True*)

Checks the validity of the environment, and if it is coherent, set it as the current environment. Furthermore wrap any non vectorized env into a vectorized checked parameters: - observation_space - action_space

Parameters

- **env** (Union[Env, VecEnv]) – The environment for learning a policy
- **force_reset** (bool) – Force call to `reset()` before training to avoid unexpected behavior. See issue https://github.com/DLR-RM/stable-baselines3/issues/597

Return type
None

set_logger(*logger*)

Setter for for logger object. :rtype: None

> **Warning:** When passing a custom logger object, this will overwrite `tensorboard_log` and `verbose` settings passed to the constructor.

set_parameters(*load_path_or_dict*, *exact_match=True*, *device='auto'*)

Load parameters from a given zip-file or a nested dictionary containing parameters for different modules (see `get_parameters`).

Parameters

- **load_path_or_iter** – Location of the saved data (path or file-like, see `save`), or a nested dictionary containing nn.Module parameters used by the policy. The dictionary maps object names to a state-dictionary returned by `torch.nn.Module.state_dict()`.
- **exact_match** (bool) – If True, the given parameters should include parameters for each module and each of their parameters, otherwise raises an Exception. If set to False, this can be used to update only specific parameters.
- **device** (Union[device, str]) – Device on which the code should run.

Return type
None

set_random_seed(*seed=None*)

Set the seed of the pseudo-random generators (python, numpy, pytorch, gym, action_space)

Parameters
seed (Optional[int]) –

Return type
None

train()

Update policy using the currently gathered rollout buffer.

Return type
None

9.6 TRPO Policies

sb3_contrib.trpo.**MlpPolicy**

alias of ActorCriticPolicy

class stable_baselines3.common.policies.**ActorCriticPolicy**(*observation_space*, *action_space*, *lr_schedule*, *net_arch=None*, *activation_fn=<class 'torch.nn.modules.activation.Tanh'>*, *ortho_init=True*, *use_sde=False*, *log_std_init=0.0*, *full_std=True*, *use_expln=False*, *squash_output=False*, *features_extractor_class=<class 'stable_baselines3.common.torch_layers.FlattenExtractor'>*, *features_extractor_kwargs=None*, *share_features_extractor=True*, *normalize_images=True*, *optimizer_class=<class 'torch.optim.adam.Adam'>*, *optimizer_kwargs=None*)

Policy class for actor-critic algorithms (has both policy and value prediction). Used by A2C, PPO and the likes.

Parameters

- **observation_space** (Space) – Observation space
- **action_space** (Space) – Action space
- **lr_schedule** (Callable[[float], float]) – Learning rate schedule (could be constant)
- **net_arch** (Union[List[int], Dict[str, List[int]], None]) – The specification of the policy and value networks.
- **activation_fn** (Type[Module]) – Activation function
- **ortho_init** (bool) – Whether to use or not orthogonal initialization
- **use_sde** (bool) – Whether to use State Dependent Exploration or not
- **log_std_init** (float) – Initial value for the log standard deviation
- **full_std** (bool) – Whether to use (n_features x n_actions) parameters for the std instead of only (n_features,) when using gSDE
- **use_expln** (bool) – Use expln() function instead of exp() to ensure a positive standard deviation (cf paper). It allows to keep variance above zero and prevent it from growing too fast. In practice, exp() is usually enough.
- **squash_output** (bool) – Whether to squash the output using a tanh function, this allows to ensure boundaries when using gSDE.
- **features_extractor_class** (Type[BaseFeaturesExtractor]) – Features extractor to use.

- **features_extractor_kwargs** (Optional[Dict[str, Any]]) – Keyword arguments to pass to the features extractor.
- **share_features_extractor** (bool) – If True, the features extractor is shared between the policy and value networks.
- **normalize_images** (bool) – Whether to normalize images or not, dividing by 255.0 (True by default)
- **optimizer_class** (Type[Optimizer]) – The optimizer to use, th.optim.Adam by default
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional keyword arguments, excluding the learning rate, to pass to the optimizer

evaluate_actions(*obs*, *actions*)

Evaluate actions according to the current policy, given the observations.

Parameters

- **obs** (Tensor) – Observation
- **actions** (Tensor) – Actions

Return type
Tuple[Tensor, Tensor, Optional[Tensor]]

Returns
estimated value, log likelihood of taking those actions and entropy of the action distribution.

extract_features(*obs*)

Preprocess the observation if needed and extract features.

Parameters
obs (Tensor) – Observation

Return type
Union[Tensor, Tuple[Tensor, Tensor]]

Returns
the output of the features extractor(s)

forward(*obs*, *deterministic=False*)

Forward pass in all the networks (actor and critic)

Parameters

- **obs** (Tensor) – Observation
- **deterministic** (bool) – Whether to sample or use deterministic actions

Return type
Tuple[Tensor, Tensor, Tensor]

Returns
action, value and log probability of the action

get_distribution(*obs*)

Get the current policy distribution given the observations.

Parameters
obs (Tensor) –

Return type
Distribution

Returns
the action distribution.

`predict_values`(*obs*)

Get the estimated values according to the current policy given the observations.

Parameters
obs (Tensor) – Observation

Return type
Tensor

Returns
the estimated values.

`reset_noise`(*n_envs=1*)

Sample new weights for the exploration matrix.

Parameters
n_envs (int) –

Return type
None

sb3_contrib.trpo.**CnnPolicy**

alias of ActorCriticCnnPolicy

class stable_baselines3.common.policies.**ActorCriticCnnPolicy**(*observation_space*, *action_space*, *lr_schedule*, *net_arch=None*, *activation_fn=<class 'torch.nn.modules.activation.Tanh'>*, *ortho_init=True*, *use_sde=False*, *log_std_init=0.0*, *full_std=True*, *use_expln=False*, *squash_output=False*, *features_extractor_class=<class 'stable_baselines3.common.torch_layers.NatureCNN'>*, *features_extractor_kwargs=None*, *share_features_extractor=True*, *normalize_images=True*, *optimizer_class=<class 'torch.optim.adam.Adam'>*, *optimizer_kwargs=None*)

CNN policy class for actor-critic algorithms (has both policy and value prediction). Used by A2C, PPO and the likes.

Parameters

- **observation_space** (Space) – Observation space
- **action_space** (Space) – Action space
- **lr_schedule** (Callable[[float], float]) – Learning rate schedule (could be constant)
- **net_arch** (Union[List[int], Dict[str, List[int]], None]) – The specification of the policy and value networks.
- **activation_fn** (Type[Module]) – Activation function
- **ortho_init** (bool) – Whether to use or not orthogonal initialization

- **use_sde** (bool) – Whether to use State Dependent Exploration or not
- **log_std_init** (float) – Initial value for the log standard deviation
- **full_std** (bool) – Whether to use (n_features x n_actions) parameters for the std instead of only (n_features,) when using gSDE
- **use_expln** (bool) – Use expln() function instead of exp() to ensure a positive standard deviation (cf paper). It allows to keep variance above zero and prevent it from growing too fast. In practice, exp() is usually enough.
- **squash_output** (bool) – Whether to squash the output using a tanh function, this allows to ensure boundaries when using gSDE.
- **features_extractor_class** (Type[BaseFeaturesExtractor]) – Features extractor to use.
- **features_extractor_kwargs** (Optional[Dict[str, Any]]) – Keyword arguments to pass to the features extractor.
- **share_features_extractor** (bool) – If True, the features extractor is shared between the policy and value networks.
- **normalize_images** (bool) – Whether to normalize images or not, dividing by 255.0 (True by default)
- **optimizer_class** (Type[Optimizer]) – The optimizer to use, th.optim.Adam by default
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional keyword arguments, excluding the learning rate, to pass to the optimizer

sb3_contrib.trpo.**MultiInputPolicy**

alias of MultiInputActorCriticPolicy

class stable_baselines3.common.policies.**MultiInputActorCriticPolicy**(*observation_space*, *action_space*, *lr_schedule*, *net_arch=None*, *activation_fn=<class 'torch.nn.modules.activation.Tanh'>*, *ortho_init=True*, *use_sde=False*, *log_std_init=0.0*, *full_std=True*, *use_expln=False*, *squash_output=False*, *features_extractor_class=<class 'stable_baselines3.common.torch_layers.Combine*, *features_extractor_kwargs=None*, *share_features_extractor=True*, *normalize_images=True*, *optimizer_class=<class 'torch.optim.adam.Adam'>*, *optimizer_kwargs=None*)

MultiInputActorClass policy class for actor-critic algorithms (has both policy and value prediction). Used by A2C, PPO and the likes.

Parameters

- **observation_space** (Dict) – Observation space (Tuple)
- **action_space** (Space) – Action space
- **lr_schedule** (Callable[[float], float]) – Learning rate schedule (could be constant)
- **net_arch** (Union[List[int], Dict[str, List[int]], None]) – The specification of the policy and value networks.
- **activation_fn** (Type[Module]) – Activation function
- **ortho_init** (bool) – Whether to use or not orthogonal initialization
- **use_sde** (bool) – Whether to use State Dependent Exploration or not
- **log_std_init** (float) – Initial value for the log standard deviation
- **full_std** (bool) – Whether to use (n_features x n_actions) parameters for the std instead of only (n_features,) when using gSDE
- **use_expln** (bool) – Use expln() function instead of exp() to ensure a positive standard deviation (cf paper). It allows to keep variance above zero and prevent it from growing too fast. In practice, exp() is usually enough.
- **squash_output** (bool) – Whether to squash the output using a tanh function, this allows to ensure boundaries when using gSDE.
- **features_extractor_class** (Type[BaseFeaturesExtractor]) – Uses the CombinedExtractor
- **features_extractor_kwargs** (Optional[Dict[str, Any]]) – Keyword arguments to pass to the features extractor.
- **share_features_extractor** (bool) – If True, the features extractor is shared between the policy and value networks.
- **normalize_images** (bool) – Whether to normalize images or not, dividing by 255.0 (True by default)
- **optimizer_class** (Type[Optimizer]) – The optimizer to use, th.optim.Adam by default
- **optimizer_kwargs** (Optional[Dict[str, Any]]) – Additional keyword arguments, excluding the learning rate, to pass to the optimizer

CHAPTER

TEN

UTILS

sb3_contrib.common.utils.**conjugate_gradient_solver**(*matrix_vector_dot_fn*, *b*, *max_iter=10*, *residual_tol=1e-10*)

Finds an approximate solution to a set of linear equations Ax = b

Sources:

- https://github.com/ajlangley/trpo-pytorch/blob/master/conjugate_gradient.py
- https://github.com/joschu/modular_rl/blob/master/modular_rl/trpo.py#L122

Reference:

- https://epubs.siam.org/doi/abs/10.1137/1.9781611971446.ch6

Parameters

- **matrix_vector_dot_fn** (Callable[[Tensor], Tensor]) – a function that right multiplies a matrix A by a vector v
- **b** – the right hand term in the set of linear equations Ax = b
- **max_iter** – the maximum number of iterations (default is 10)
- **residual_tol** – residual tolerance for early stopping of the solving (default is 1e-10)

Return x
the approximate solution to the system of equations defined by *matrix_vector_dot_fn* and b

Return type
Tensor

sb3_contrib.common.utils.**flat_grad**(*output*, *parameters*, *create_graph=False*, *retain_graph=False*)

Returns the gradients of the passed sequence of parameters into a flat gradient. Order of parameters is preserved.

Parameters

- **output** – functional output to compute the gradient for
- **parameters** (Sequence[Parameter]) – sequence of Parameter
- **retain_graph** (bool) – If False, the graph used to compute the grad will be freed. Defaults to the value of create_graph.
- **create_graph** (bool) – If True, graph of the derivative will be constructed, allowing to compute higher order derivative products. Default: False.

Return type
Tensor

Returns
Tensor containing the flattened gradients

sb3_contrib.common.utils.**quantile_huber_loss**(*current_quantiles*, *target_quantiles*, *cum_prob=None*, *sum_over_quantiles=True*)

The quantile-regression loss, as described in the QR-DQN and TQC papers. Partially taken from https://github.com/bayesgroup/tqc_pytorch.

Parameters

- **current_quantiles** (Tensor) – current estimate of quantiles, must be either (batch_size, n_quantiles) or (batch_size, n_critics, n_quantiles)
- **target_quantiles** (Tensor) – target of quantiles, must be either (batch_size, n_target_quantiles), (batch_size, 1, n_target_quantiles), or (batch_size, n_critics, n_target_quantiles)
- **cum_prob** (Optional[Tensor]) – cumulative probabilities to calculate quantiles (also called midpoints in QR-DQN paper), must be either (batch_size, n_quantiles), (batch_size, 1, n_quantiles), or (batch_size, n_critics, n_quantiles). (if None, calculating unit quantiles)
- **sum_over_quantiles** (bool) – if summing over the quantile dimension or not

Return type
Tensor

Returns
the loss

CHAPTER

ELEVEN

GYM WRAPPERS

Additional Gym Wrappers to enhance Gym environments.

11.1 TimeFeatureWrapper

class sb3_contrib.common.wrappers.**TimeFeatureWrapper**(*env*, *max_steps=1000*, *test_mode=False*)

Add remaining, normalized time to observation space for fixed length episodes. See https://arxiv.org/abs/1712.00378 and https://github.com/aravindr93/mjrl/issues/13.

Note: Only `gym.spaces.Box` and `gym.spaces.Dict` (`gym.GoalEnv`) 1D observation spaces are supported for now.

Parameters

- **env** (Env) – Gym env to wrap.
- **max_steps** (int) – Max number of steps of an episode if it is not wrapped in a `TimeLimit` object.
- **test_mode** (bool) – In test mode, the time feature is constant, equal to zero. This allow to check that the agent did not overfit this feature, learning a deterministic pre-defined sequence of actions.

reset()

Resets the environment to an initial state and returns an initial observation.

Note that this function should not reset the environment's random number generator(s); random variables in the environment's state should be sampled independently between multiple calls to *reset()*. In other words, each call of *reset()* should yield an environment suitable for a new episode, independent of previous episodes.

Return type
Union[Tuple, Dict[str, Any], ndarray, int]

Returns:
observation (object): the initial observation.

`step`(*action*)

Run one timestep of the environment's dynamics. When end of episode is reached, you are responsible for calling *reset()* to reset this environment's state.

Accepts an action and returns a tuple (observation, reward, done, info).

Return type
`Tuple[Union[Tuple, Dict[str, Any], ndarray, int], float, bool, Dict]`

Args:
action (object): an action provided by the agent

Returns:
observation (object): agent's observation of the current environment reward (float) : amount of reward returned after previous action done (bool): whether the episode has ended, in which case further step() calls will return undefined results info (dict): contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

CHAPTER

TWELVE

CHANGELOG

12.1 Release 1.8.0 (2023-04-07)

Warning: Stable-Baselines3 (SB3) v1.8.0 will be the last one to use Gym as a backend. Starting with v2.0.0, Gymnasium will be the default backend (though SB3 will have compatibility layers for Gym envs). You can find a migration guide here: https://gymnasium.farama.org/content/migration-guide/. If you want to try the SB3 v2.0 alpha version, you can take a look at PR #1327.

12.1.1 Breaking Changes:

- Removed shared layers in `mlp_extractor` (@AlexPasqua)
- Upgraded to Stable-Baselines3 >= 1.8.0

12.1.2 New Features:

- Added `stats_window_size` argument to control smoothing in rollout logging (@jonasreiher)

12.1.3 Bug Fixes:

12.1.4 Deprecations:

12.1.5 Others:

- Moved to pyproject.toml
- Added github issue forms
- Fixed Atari Roms download in CI
- Fixed `sb3_contrib/qrdqn/*.py` type hints
- Switched from `flake8` to `ruff`

12.1.6 Documentation:

- Added warning about potential crashes caused by `check_env` in the `MaskablePPO` docs (@AlexPasqua)

12.2 Release 1.7.0 (2023-01-10)

Warning: Shared layers in MLP policy (`mlp_extractor`) are now deprecated for PPO, A2C and TRPO. This feature will be removed in SB3 v1.8.0 and the behavior of `net_arch=[64, 64]` will create **separate** networks with the same architecture, to be consistent with the off-policy algorithms.

12.2.1 Breaking Changes:

- Removed deprecated `create_eval_env`, `eval_env`, `eval_log_path`, `n_eval_episodes` and `eval_freq` parameters, please use an `EvalCallback` instead
- Removed deprecated `sde_net_arch` parameter
- Upgraded to Stable-Baselines3 >= 1.7.0

12.2.2 New Features:

- Introduced mypy type checking
- Added support for Python 3.10
- Added `with_bias` parameter to `ARSPolicy`
- Added option to have non-shared features extractor between actor and critic in on-policy algorithms (@AlexPasqua)
- Features extractors now properly support unnormalized image-like observations (3D tensor) when passing `normalize_images=False`

12.2.3 Bug Fixes:

- Fixed a bug in `RecurrentPPO` where the lstm states where incorrectly reshaped for `n_lstm_layers > 1` (thanks @kolbytn)
- Fixed `RuntimeError: rnn: hx is not contiguous` while predicting terminal values for `RecurrentPPO` when `n_lstm_layers > 1`

12.2.4 Deprecations:

- You should now explicitely pass a `features_extractor` parameter when calling `extract_features()`
- Deprecated shared layers in `MlpExtractor` (@AlexPasqua)

12.2.5 Others:

- Fixed flake8 config
- Fixed `sb3_contrib/common/utils.py` type hint
- Fixed `sb3_contrib/common/recurrent/type_aliases.py` type hint
- Fixed `sb3_contrib/ars/policies.py` type hint
- Exposed modules in *__init__.py* with *__all__* attribute (@ZikangXiong)
- Removed ignores on Flake8 F401 (@ZikangXiong)
- Upgraded GitHub CI/setup-python to v4 and checkout to v3
- Set tensors construction directly on the device
- Standardized the use of `from gym import spaces`

12.3 Release 1.6.2 (2022-10-10)

Progress bar and upgrade to latest SB3 version

12.3.1 Breaking Changes:

- Upgraded to Stable-Baselines3 >= 1.6.2

12.3.2 New Features:

- Added `progress_bar` argument in the `learn()` method, displayed using TQDM and rich packages

12.3.3 Bug Fixes:

12.3.4 Deprecations:

- Deprecate parameters `eval_env`, `eval_freq` and `create_eval_env`

12.3.5 Others:

- Fixed the return type of `.load()` methods so that they now use `TypeVar`

12.4 Release 1.6.1 (2022-09-29)

Bug fix release

12.4.1 Breaking Changes:

- Fixed the issue that `predict` does not always return action as `np.ndarray` (@qgallouedec)
- Upgraded to Stable-Baselines3 >= 1.6.1

12.4.2 New Features:

12.4.3 Bug Fixes:

- Fixed the issue of wrongly passing policy arguments when using CnnLstmPolicy or MultiInputLstmPolicy with `RecurrentPPO` (@mlodel)
- Fixed division by zero error when computing FPS when a small number of time has elapsed in operating systems with low-precision timers.
- Fixed calling child callbacks in MaskableEvalCallback (@CppMaster)
- Fixed missing verbose parameter passing in the `MaskableEvalCallback` constructor (@burakdmb)
- Fixed the issue that when updating the target network in QRDQN, TQC, the `running_mean` and `running_var` properties of batch norm layers are not updated (@honglu2875)

12.4.4 Deprecations:

12.4.5 Others:

- Changed the default buffer device from `"cpu"` to `"auto"`

12.5 Release 1.6.0 (2022-07-11)

Add RecurrentPPO (aka PPO LSTM)

12.5.1 Breaking Changes:

- Upgraded to Stable-Baselines3 >= 1.6.0
- Changed the way policy "aliases" are handled ("MlpPolicy", "CnnPolicy", ...), removing the former `register_policy` helper, `policy_base` parameter and using `policy_aliases` static attributes instead (@Gregwar)
- Renamed `rollout/exploration rate` key to `rollout/exploration_rate` for QRDQN (to be consistent with SB3 DQN)
- Upgraded to python 3.7+ syntax using `pyupgrade`
- SB3 now requires PyTorch >= 1.11
- Changed the default network architecture when using `CnnPolicy` or `MultiInputPolicy` with TQC, `share_features_extractor` is now set to False by default and the `net_arch=[256, 256]` (instead of `net_arch=[]` that was before)

12.5.2 New Features:

- Added `RecurrentPPO` (aka PPO LSTM)

12.5.3 Bug Fixes:

- Fixed a bug in `RecurrentPPO` when calculating the masked loss functions (@rnederstigt)
- Fixed a bug in `TRPO` where kl divergence was not implemented for `MultiDiscrete` space

12.5.4 Deprecations:

12.6 Release 1.5.0 (2022-03-25)

12.6.1 Breaking Changes:

- Switched minimum Gym version to 0.21.0.
- Upgraded to Stable-Baselines3 >= 1.5.0

12.6.2 New Features:

- Allow PPO to turn of advantage normalization (see PR #61) (@vwxyzjn)

12.6.3 Bug Fixes:

- Removed explict calls to `forward()` method as per pytorch guidelines

12.6.4 Deprecations:

12.6.5 Others:

12.6.6 Documentation:

12.7 Release 1.4.0 (2022-01-19)

Add Trust Region Policy Optimization (TRPO) and Augmented Random Search (ARS) algorithms

12.7.1 Breaking Changes:

- Dropped python 3.6 support
- Upgraded to Stable-Baselines3 >= 1.4.0
- `MaskablePPO` was updated to match latest SB3 `PPO` version (timeout handling and new method for the policy object)

12.7.2 New Features:

- Added `TRPO` (@cyprienc)
- Added experimental support to train off-policy algorithms with multiple envs (note: `HerReplayBuffer` currently not supported)
- Added Augmented Random Search (ARS) (@sgillen)

12.7.3 Bug Fixes:

12.7.4 Deprecations:

12.7.5 Others:

- Improve test coverage for `MaskablePPO`

12.7.6 Documentation:

12.8 Release 1.3.0 (2021-10-23)

Add Invalid action masking for PPO

> **Warning:** This version will be the last one supporting Python 3.6 (end of life in Dec 2021). We highly recommended you to upgrade to Python >= 3.7.

12.8.1 Breaking Changes:

- Removed `sde_net_arch`
- Upgraded to Stable-Baselines3 >= 1.3.0

12.8.2 New Features:

- Added `MaskablePPO` algorithm (@kronion)
- `MaskablePPO` Dictionary Observation support (@glmcdona)

12.8.3 Bug Fixes:

12.8.4 Deprecations:

12.8.5 Others:

12.8.6 Documentation:

12.9 Release 1.2.0 (2021-09-08)

Train/Eval mode support

12.9.1 Breaking Changes:

- Upgraded to Stable-Baselines3 >= 1.2.0

12.9.2 Bug Fixes:

- QR-DQN and TQC updated so that their policies are switched between train and eval mode at the correct time (@ayeright)

12.9.3 Deprecations:

12.9.4 Others:

- Fixed type annotation
- Added python 3.9 to CI

12.9.5 Documentation:

12.10 Release 1.1.0 (2021-07-01)

Dictionary observation support and timeout handling

12.10.1 Breaking Changes:

- Added support for Dictionary observation spaces (cf. SB3 doc)
- Upgraded to Stable-Baselines3 >= 1.1.0
- Added proper handling of timeouts for off-policy algorithms (cf. SB3 doc)
- Updated usage of logger (cf. SB3 doc)

12.10.2 Bug Fixes:

- Removed unused code in `TQC`

12.10.3 Deprecations:

12.10.4 Others:

- SB3 docs and tests dependencies are no longer required for installing SB3 contrib

12.10.5 Documentation:

- updated QR-DQN docs checkmark typo (@minhlong94)

12.11 Release 1.0 (2021-03-17)

12.11.1 Breaking Changes:

- Upgraded to Stable-Baselines3 >= 1.0

12.11.2 Bug Fixes:

- Fixed a bug with `QR-DQN` predict method when using `deterministic=False` with image space

12.12 Pre-Release 0.11.1 (2021-02-27)

12.12.1 Bug Fixes:

- Upgraded to Stable-Baselines3 >= 0.11.1

12.13 Pre-Release 0.11.0 (2021-02-27)

12.13.1 Breaking Changes:

- Upgraded to Stable-Baselines3 >= 0.11.0

12.13.2 New Features:

- Added `TimeFeatureWrapper` to the wrappers
- Added `QR-DQN` algorithm (@ku2482)

12.13.3 Bug Fixes:

- Fixed bug in `TQC` when saving/loading the policy only with non-default number of quantiles
- Fixed bug in `QR-DQN` when calculating the target quantiles (@ku2482, @guyk1971)

12.13.4 Deprecations:

12.13.5 Others:

- Updated `TQC` to match new SB3 version
- Updated SB3 min version
- Moved `quantile_huber_loss` to `common/utils.py` (@ku2482)

12.13.6 Documentation:

12.14 Pre-Release 0.10.0 (2020-10-28)

Truncated Quantiles Critic (TQC)

12.14.1 Breaking Changes:

12.14.2 New Features:

- Added `TQC` algorithm (@araffin)

12.14.3 Bug Fixes:

- Fixed features extractor issue (`TQC` with `CnnPolicy`)

12.14.4 Deprecations:

12.14.5 Others:

12.14.6 Documentation:

- Added initial documentation
- Added contribution guide and related PR templates

12.15 Maintainers

Stable-Baselines3 is currently maintained by Antonin Raffin (aka @araffin), Ashley Hill (aka @hill-a), Maximilian Ernestus (aka @ernestum), Adam Gleave (@AdamGleave) and Anssi Kanervisto (aka @Miffyli).

12.16 Contributors:

@ku2482 @guyk1971 @minhlong94 @ayeright @kronion @glmcdona @cyprienc @sgillen @Gregwar @rnederstigt @qgallouedec @mlodel @CppMaster @burakdmb @honglu2875 @ZikangXiong @AlexPasqua @jonasreiher

CHAPTER

THIRTEEN

CITING STABLE BASELINES3

To cite this project in publications:

```
@misc{stable-baselines3,
  author = {Raffin, Antonin and Hill, Ashley and Ernestus, Maximilian and Gleave, Adam␣
↪and Kanervisto, Anssi and Dormann, Noah},
  title = {Stable Baselines3},
  year = {2019},
  publisher = {GitHub},
  journal = {GitHub repository},
  howpublished = {\url{https://github.com/DLR-RM/stable-baselines3}},
}
```

CHAPTER

FOURTEEN

CONTRIBUTING

If you want to contribute, please read CONTRIBUTING.md first.

CHAPTER

FIFTEEN

INDICES AND TABLES

- genindex
- search
- modindex

PYTHON MODULE INDEX

S

sb3_contrib.ars, 8
sb3_contrib.common.utils, 83
sb3_contrib.common.wrappers, 85
sb3_contrib.ppo_mask, 16
sb3_contrib.ppo_recurrent, 33
sb3_contrib.qrdqn, 48
sb3_contrib.tqc, 58
sb3_contrib.trpo, 69

INDEX

A

ARS (*class in sb3_contrib.ars*), 11

C

CnnLstmPolicy (*in module sb3_contrib.ppo_recurrent*), 45
CnnPolicy (*class in sb3_contrib.qrdqn*), 57
CnnPolicy (*class in sb3_contrib.tqc*), 68
CnnPolicy (*in module sb3_contrib.ppo_mask*), 31
CnnPolicy (*in module sb3_contrib.trpo*), 80
collect_rollouts() (*sb3_contrib.ppo_mask.MaskablePPO method*), 25
collect_rollouts() (*sb3_contrib.ppo_recurrent.RecurrentPPO method*), 38
collect_rollouts() (*sb3_contrib.qrdqn.QRDQN method*), 52
collect_rollouts() (*sb3_contrib.tqc.TQC method*), 63
collect_rollouts() (*sb3_contrib.trpo.TRPO method*), 74
conjugate_gradient_solver() (*in module sb3_contrib.common.utils*), 83

E

evaluate_candidates() (*sb3_contrib.ars.ARS method*), 12

F

flat_grad() (*in module sb3_contrib.common.utils*), 83

G

get_env() (*sb3_contrib.ars.ARS method*), 12
get_env() (*sb3_contrib.ppo_mask.MaskablePPO method*), 25
get_env() (*sb3_contrib.ppo_recurrent.RecurrentPPO method*), 38
get_env() (*sb3_contrib.qrdqn.QRDQN method*), 53
get_env() (*sb3_contrib.tqc.TQC method*), 63
get_env() (*sb3_contrib.trpo.TRPO method*), 74
get_parameters() (*sb3_contrib.ars.ARS method*), 12
get_parameters() (*sb3_contrib.ppo_mask.MaskablePPO method*), 25
get_parameters() (*sb3_contrib.ppo_recurrent.RecurrentPPO method*), 38
get_parameters() (*sb3_contrib.qrdqn.QRDQN method*), 53
get_parameters() (*sb3_contrib.tqc.TQC method*), 63
get_parameters() (*sb3_contrib.trpo.TRPO method*), 74
get_vec_normalize_env() (*sb3_contrib.ars.ARS method*), 13
get_vec_normalize_env() (*sb3_contrib.ppo_mask.MaskablePPO method*), 25
get_vec_normalize_env() (*sb3_contrib.ppo_recurrent.RecurrentPPO method*), 39
get_vec_normalize_env() (*sb3_contrib.qrdqn.QRDQN method*), 53
get_vec_normalize_env() (*sb3_contrib.tqc.TQC method*), 63
get_vec_normalize_env() (*sb3_contrib.trpo.TRPO method*), 75

H

hessian_vector_product() (*sb3_contrib.trpo.TRPO method*), 75

L

learn() (*sb3_contrib.ars.ARS method*), 13
learn() (*sb3_contrib.ppo_mask.MaskablePPO method*), 26
learn() (*sb3_contrib.ppo_recurrent.RecurrentPPO method*), 39
learn() (*sb3_contrib.qrdqn.QRDQN method*), 53
learn() (*sb3_contrib.tqc.TQC method*), 64
learn() (*sb3_contrib.trpo.TRPO method*), 75
LinearPolicy (*in module sb3_contrib.ars*), 16
load() (*sb3_contrib.ars.ARS class method*), 13
load() (*sb3_contrib.ppo_mask.MaskablePPO class method*), 26
load() (*sb3_contrib.ppo_recurrent.RecurrentPPO class method*), 39
load() (*sb3_contrib.qrdqn.QRDQN class method*), 53

load() (*sb3_contrib.tqc.TQC class method*), 64
load() (*sb3_contrib.trpo.TRPO class method*), 75
load_replay_buffer() (*sb3_contrib.qrdqn.QRDQN method*), 54
load_replay_buffer() (*sb3_contrib.tqc.TQC method*), 65
logger (*sb3_contrib.ars.ARS property*), 14
logger (*sb3_contrib.ppo_mask.MaskablePPO property*), 27
logger (*sb3_contrib.ppo_recurrent.RecurrentPPO property*), 40
logger (*sb3_contrib.qrdqn.QRDQN property*), 54
logger (*sb3_contrib.tqc.TQC property*), 65
logger (*sb3_contrib.trpo.TRPO property*), 76

M

MaskablePPO (*class in sb3_contrib.ppo_mask*), 24
MlpLstmPolicy (*in module sb3_contrib.ppo_recurrent*), 42
MlpPolicy (*in module sb3_contrib.ars*), 16
MlpPolicy (*in module sb3_contrib.ppo_mask*), 29
MlpPolicy (*in module sb3_contrib.qrdqn*), 56
MlpPolicy (*in module sb3_contrib.tqc*), 67
MlpPolicy (*in module sb3_contrib.trpo*), 78
module
    sb3_contrib.ars, 8
    sb3_contrib.common.utils, 83
    sb3_contrib.common.wrappers, 85
    sb3_contrib.ppo_mask, 16
    sb3_contrib.ppo_recurrent, 33
    sb3_contrib.qrdqn, 48
    sb3_contrib.tqc, 58
    sb3_contrib.trpo, 69
MultiInputLstmPolicy (*in module sb3_contrib.ppo_recurrent*), 46
MultiInputPolicy (*class in sb3_contrib.qrdqn*), 58
MultiInputPolicy (*in module sb3_contrib.ppo_mask*), 32
MultiInputPolicy (*in module sb3_contrib.trpo*), 81

P

predict() (*sb3_contrib.ars.ARS method*), 14
predict() (*sb3_contrib.ppo_mask.MaskablePPO method*), 27
predict() (*sb3_contrib.ppo_recurrent.RecurrentPPO method*), 40
predict() (*sb3_contrib.qrdqn.QRDQN method*), 54
predict() (*sb3_contrib.tqc.TQC method*), 65
predict() (*sb3_contrib.trpo.TRPO method*), 76

Q

QRDQN (*class in sb3_contrib.qrdqn*), 51
quantile_huber_loss() (*in module sb3_contrib.common.utils*), 84

R

RecurrentPPO (*class in sb3_contrib.ppo_recurrent*), 37
reset() (*sb3_contrib.common.wrappers.TimeFeatureWrapper method*), 85

S

save() (*sb3_contrib.ars.ARS method*), 14
save() (*sb3_contrib.ppo_mask.MaskablePPO method*), 27
save() (*sb3_contrib.ppo_recurrent.RecurrentPPO method*), 40
save() (*sb3_contrib.qrdqn.QRDQN method*), 55
save() (*sb3_contrib.tqc.TQC method*), 65
save() (*sb3_contrib.trpo.TRPO method*), 76
save_replay_buffer() (*sb3_contrib.qrdqn.QRDQN method*), 55
save_replay_buffer() (*sb3_contrib.tqc.TQC method*), 65
sb3_contrib.ars
    module, 8
sb3_contrib.common.utils
    module, 83
sb3_contrib.common.wrappers
    module, 85
sb3_contrib.ppo_mask
    module, 16
sb3_contrib.ppo_recurrent
    module, 33
sb3_contrib.qrdqn
    module, 48
sb3_contrib.tqc
    module, 58
sb3_contrib.trpo
    module, 69
set_env() (*sb3_contrib.ars.ARS method*), 14
set_env() (*sb3_contrib.ppo_mask.MaskablePPO method*), 27
set_env() (*sb3_contrib.ppo_recurrent.RecurrentPPO method*), 40
set_env() (*sb3_contrib.qrdqn.QRDQN method*), 55
set_env() (*sb3_contrib.tqc.TQC method*), 66
set_env() (*sb3_contrib.trpo.TRPO method*), 77
set_logger() (*sb3_contrib.ars.ARS method*), 14
set_logger() (*sb3_contrib.ppo_mask.MaskablePPO method*), 27
set_logger() (*sb3_contrib.ppo_recurrent.RecurrentPPO method*), 41
set_logger() (*sb3_contrib.qrdqn.QRDQN method*), 55
set_logger() (*sb3_contrib.tqc.TQC method*), 66
set_logger() (*sb3_contrib.trpo.TRPO method*), 77
set_parameters() (*sb3_contrib.ars.ARS method*), 15
set_parameters() (*sb3_contrib.ppo_mask.MaskablePPO method*), 28

set_parameters() (*sb3_contrib.ppo_recurrent.RecurrentPPO method*), 41
set_parameters() (*sb3_contrib.qrdqn.QRDQN method*), 55
set_parameters() (*sb3_contrib.tqc.TQC method*), 66
set_parameters() (*sb3_contrib.trpo.TRPO method*), 77
set_random_seed() (*sb3_contrib.ars.ARS method*), 15
set_random_seed() (*sb3_contrib.ppo_mask.MaskablePPO method*), 28
set_random_seed() (*sb3_contrib.ppo_recurrent.RecurrentPPO method*), 41
set_random_seed() (*sb3_contrib.qrdqn.QRDQN method*), 56
set_random_seed() (*sb3_contrib.tqc.TQC method*), 66
set_random_seed() (*sb3_contrib.trpo.TRPO method*), 77
step() (*sb3_contrib.common.wrappers.TimeFeatureWrapper method*), 85

T

TimeFeatureWrapper (*class in sb3_contrib.common.wrappers*), 85
TQC (*class in sb3_contrib.tqc*), 61
train() (*sb3_contrib.ppo_mask.MaskablePPO method*), 28
train() (*sb3_contrib.ppo_recurrent.RecurrentPPO method*), 41
train() (*sb3_contrib.qrdqn.QRDQN method*), 56
train() (*sb3_contrib.tqc.TQC method*), 66
train() (*sb3_contrib.trpo.TRPO method*), 77
TRPO (*class in sb3_contrib.trpo*), 73