Stable Baselines3 Documentation

Release 1.5.0

Stable Baselines3 Contributors

Mar 12, 2023

USER GUIDE

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	✓				✓
QR-DQN		✓			✓
TQC	✓				✓
TRPO	✓	✓	✓	✓	✓

Note: Non-array spaces such as `Dict` or `Tuple` are not currently supported by any algorithm.

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")
```

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

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

4.6 ARS Policies

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

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):

```
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()


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
```

(continues on next page)

(continued from previous page)

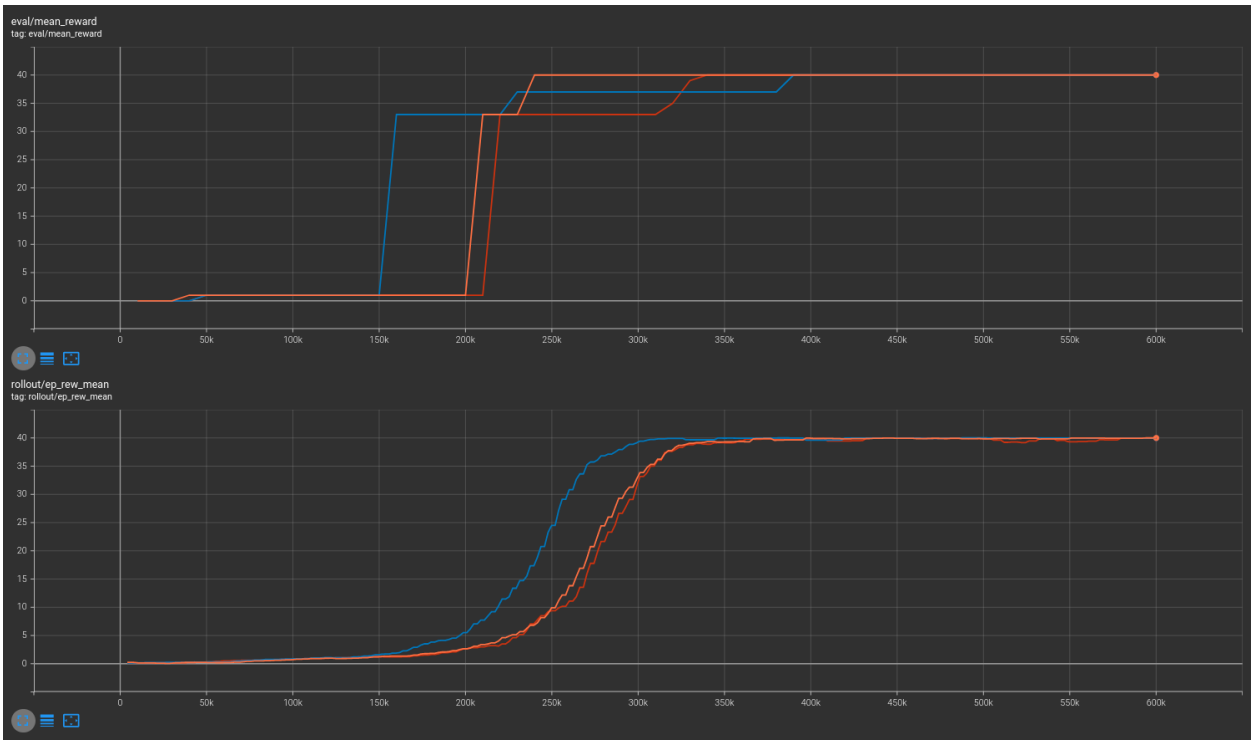
```
# retrieved and used when learning. Note that MaskablePPO does not accept
# a new action_mask_fn kwarg, as it did in an earlier draft.
model = MaskablePPO(MaskableActorCriticPolicy, env, verbose=1)
model.learn()

# 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

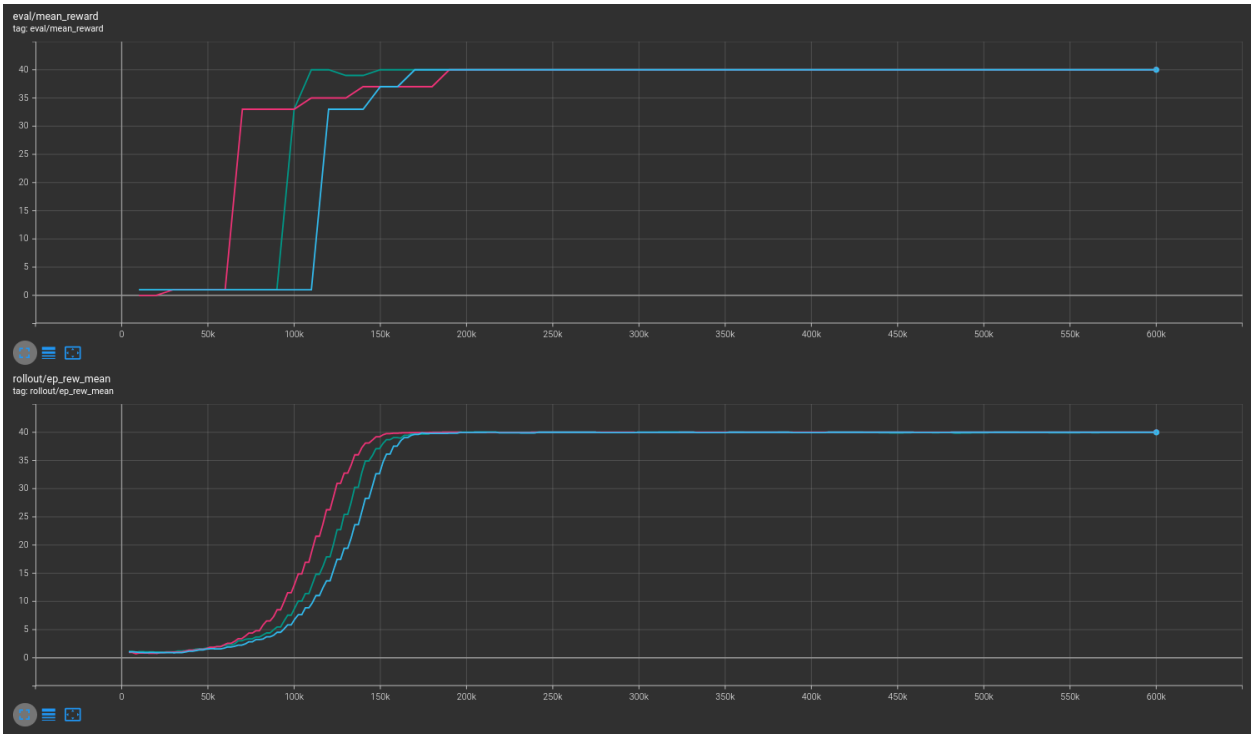
Results are shown for two MicroRTS benchmarks: MicrortsMining4x4F9-v0 (600K steps) and MicrortsMining10x10F9-v0 (1.5M steps). For each, models were trained with and without masking, using 3 seeds.

5.4.1 4x4

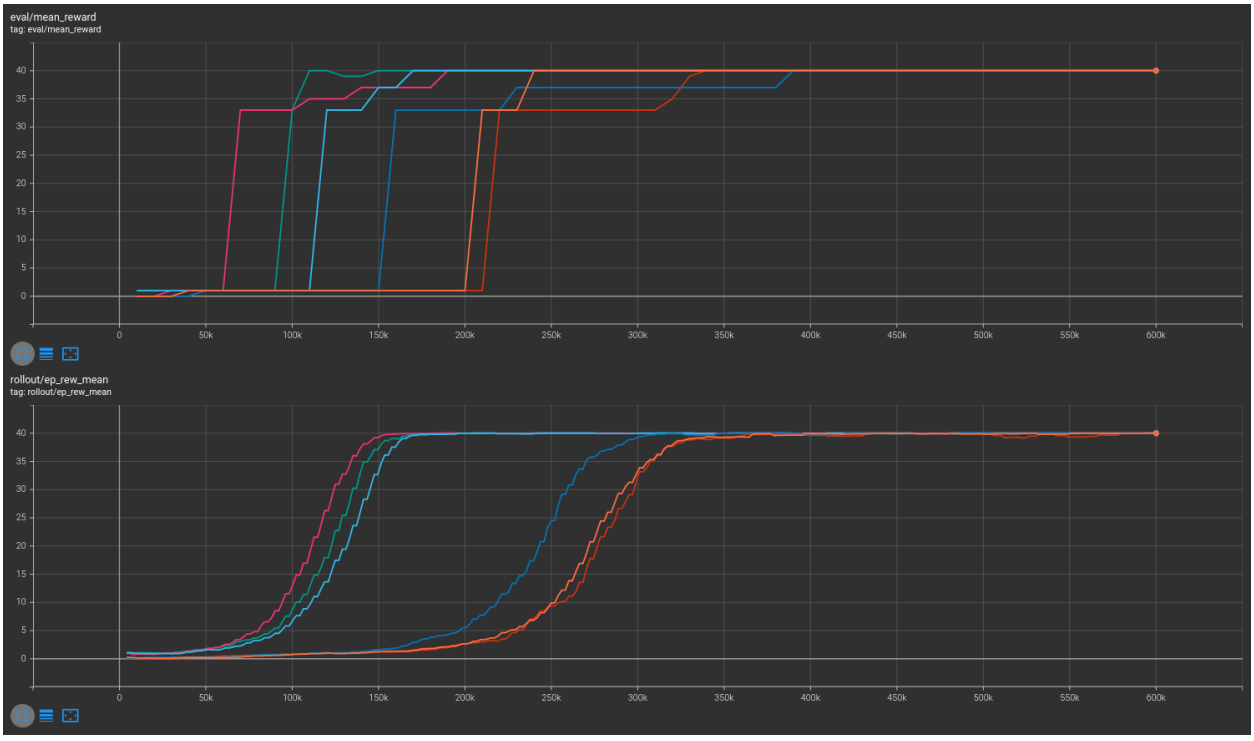
No masking



With masking

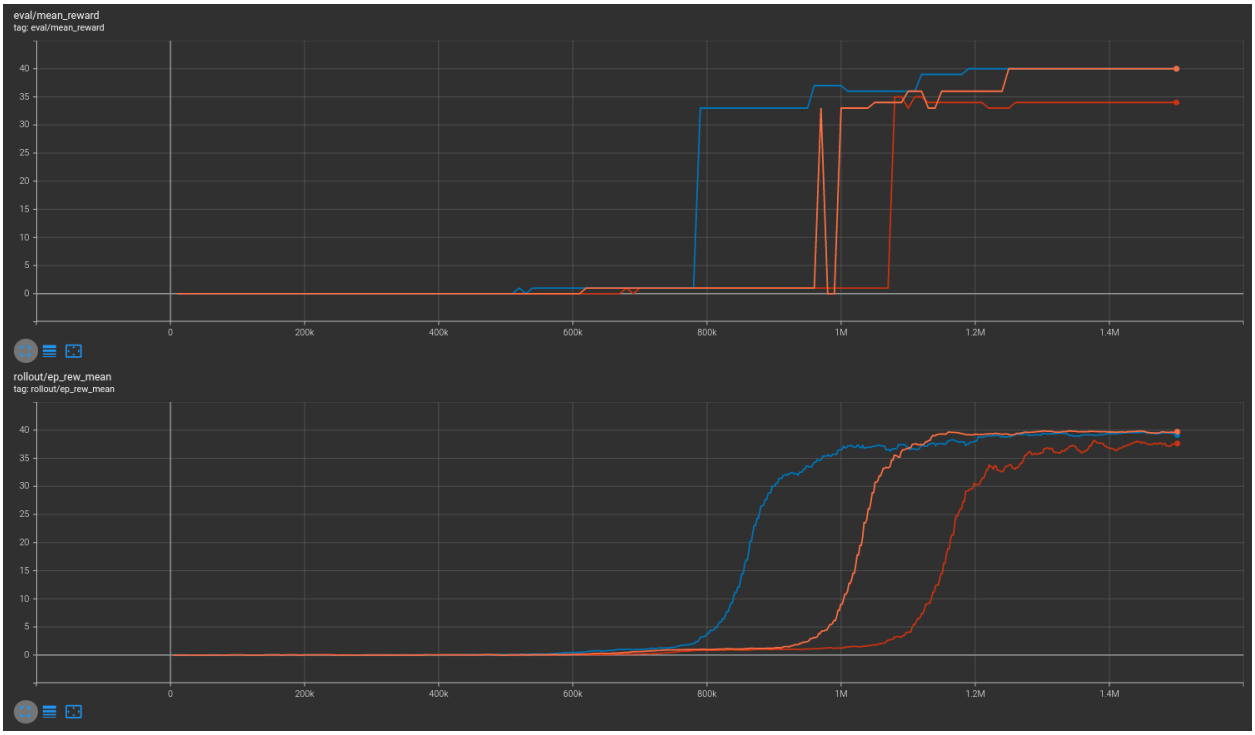


Combined

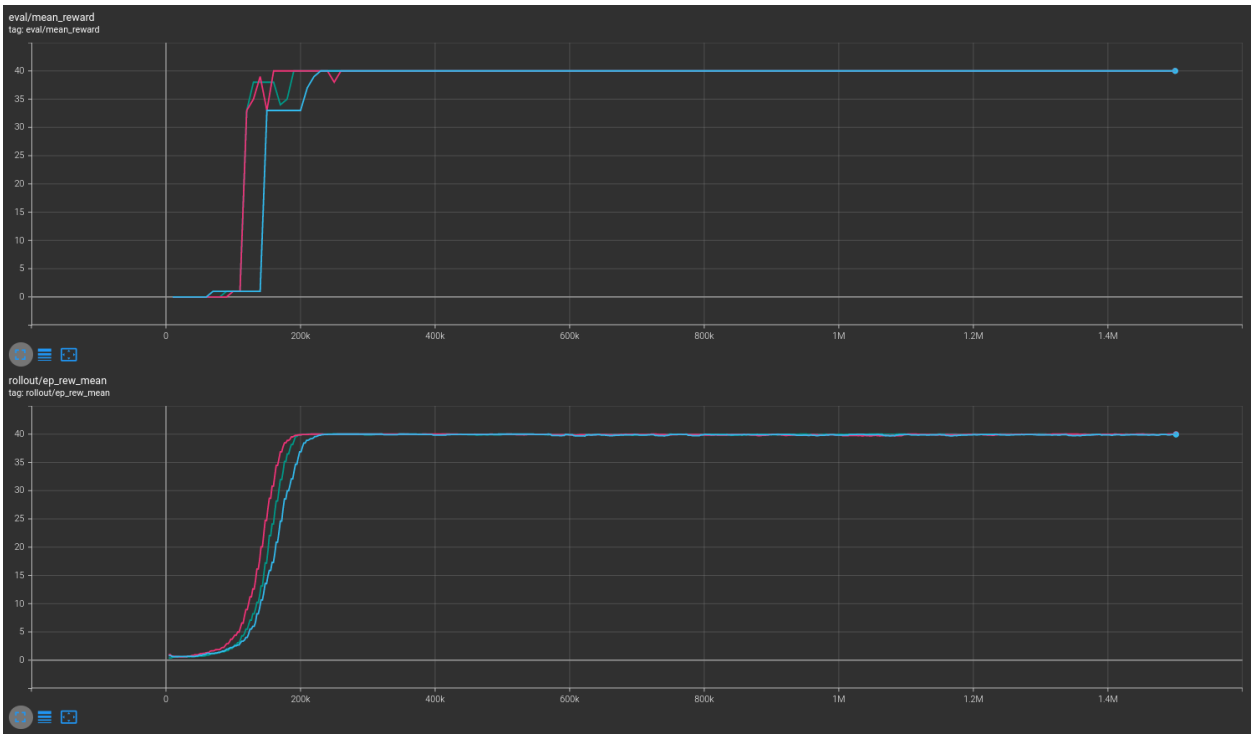


5.4.2 10x10

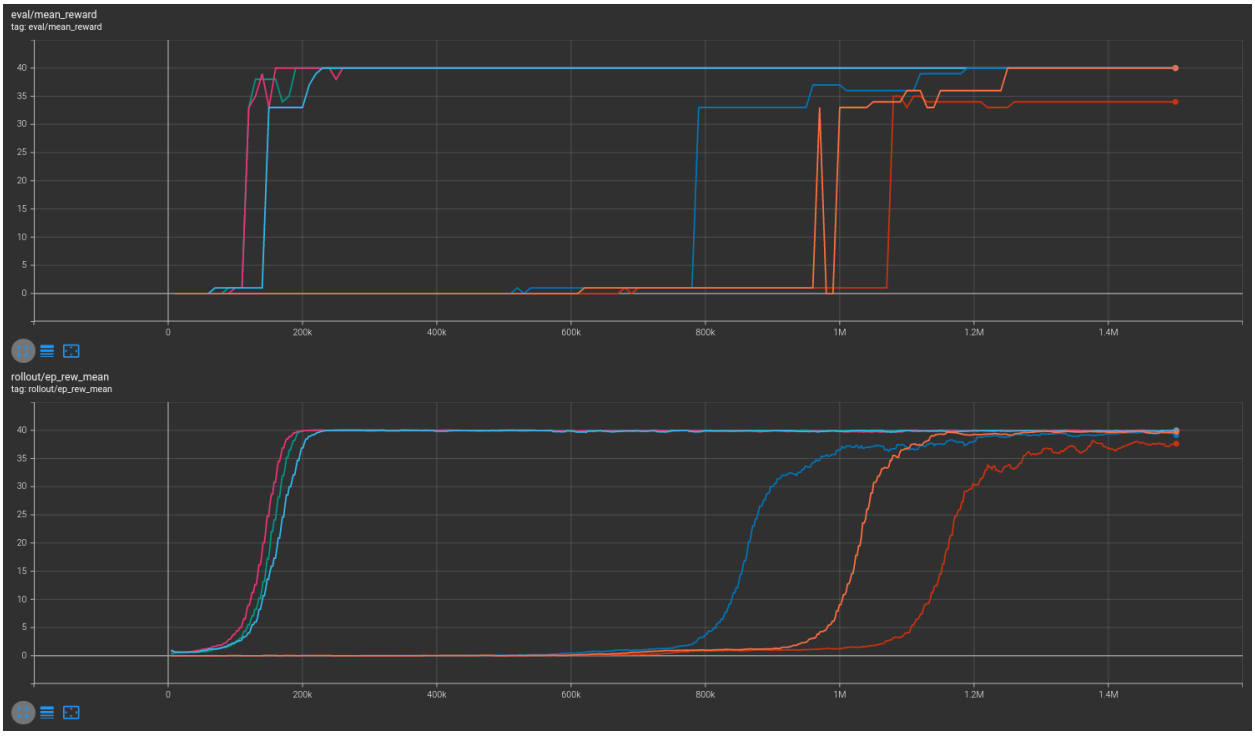
No masking



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

5.6 MaskablePPO Policies

CHAPTER

SIX

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

6.1 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

6.2 Can I use?

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

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

6.3 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()
```

6.4 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

6.4.1 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
```

6.5 Parameters

6.6 QR-DQN Policies

CHAPTER

SEVEN

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

7.1 Notes

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

7.2 Can I use?

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

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

7.3 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()
```

7.4 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

7.4.1 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
```

7.5 Comments

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

7.6 Parameters

7.7 TQC Policies

CHAPTER

EIGHT

TRPO

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

Available Policies

8.1 Notes

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

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 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()
```

8.4 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

8.4.1 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
```

8.5 Parameters

8.6 TRPO Policies

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]], List[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

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]], List[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

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]], List[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

NINE

UTILS

CHAPTER

TEN

GYM WRAPPERS

Additional Gym Wrappers to enhance Gym environments.

10.1 TimeFeatureWrapper

CHAPTER

ELEVEN

CHANGELOG

11.1 Release 1.5.0 (2022-03-25)

11.1.1 Breaking Changes:

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

11.1.2 New Features:

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

11.1.3 Bug Fixes:

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

11.1.4 Deprecations:

11.1.5 Others:

11.2 Release 1.4.0 (2022-01-19)

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

11.2.1 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)

11.2.2 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)

11.2.3 Bug Fixes:

11.2.4 Deprecations:

11.2.5 Others:

- Improve test coverage for `MaskablePPO`

11.2.6 Documentation:

11.3 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.

11.3.1 Breaking Changes:

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

11.3.2 New Features:

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

11.3.3 Bug Fixes:

11.3.4 Deprecations:

11.3.5 Others:

11.3.6 Documentation:

11.4 Release 1.2.0 (2021-09-08)

Train/Eval mode support

11.4.1 Breaking Changes:

- Upgraded to Stable-Baselines3 >= 1.2.0

11.4.2 Bug Fixes:

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

11.4.3 Deprecations:

11.4.4 Others:

- Fixed type annotation
- Added python 3.9 to CI

11.4.5 Documentation:

11.5 Release 1.1.0 (2021-07-01)

Dictionary observation support and timeout handling

11.5.1 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)

11.5.2 Bug Fixes:

- Removed unused code in `TQC`

11.5.3 Deprecations:

11.5.4 Others:

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

11.5.5 Documentation:

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

11.6 Release 1.0 (2021-03-17)

11.6.1 Breaking Changes:

- Upgraded to Stable-Baselines3 >= 1.0

11.6.2 Bug Fixes:

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

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

11.7.1 Bug Fixes:

- Upgraded to Stable-Baselines3 >= 0.11.1

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

11.8.1 Breaking Changes:

- Upgraded to Stable-Baselines3 >= 0.11.0

11.8.2 New Features:

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

11.8.3 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)

11.8.4 Deprecations:

11.8.5 Others:

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

11.8.6 Documentation:

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

Truncated Quantiles Critic (TQC)

11.9.1 Breaking Changes:

11.9.2 New Features:

- Added `TQC` algorithm (@araffin)

11.9.3 Bug Fixes:

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

11.9.4 Deprecations:

11.9.5 Others:

11.9.6 Documentation:

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

11.10 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).

11.11 Contributors:

@ku2482 @guyk1971 @minhlong94 @ayeright @kronion @glmcdona @cyprienc @sgillen

CHAPTER

TWELVE

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

THIRTEEN

CONTRIBUTING

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

CHAPTER

FOURTEEN

INDICES AND TABLES

- genindex
- search
- modindex