Stable Baselines3 Documentation

Release 2.9.0

Stable Baselines3 Contributors

Jun 15, 2026

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.

1.2 Stable Release

To install Stable Baselines3 contrib with pip, execute:

```
pip install sb3-contrib
```

1.3 Bleeding-edge version

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

1.4 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		✓	✓	✓	✓
CrossQ	✓				✓
QR-DQN		✓			✓
RecurrentPPO	✓	✓	✓	✓	✓
TQC	✓				✓
TRPO	✓	✓	✓	✓	✓

Note

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

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.

Warning

You must use `MaskableEvalCallback` from `sb3_contrib.common.maskable.callbacks` instead of the base `EvalCallback` to properly evaluate a model with action masks. Similarly, you must use `evaluate_policy` from `sb3_contrib.common.maskable.evaluation` instead of the SB3 one.

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

(continues on next page)

(continued from previous page)

```
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)

vec_env = model.get_env()
obs = vec_env.reset()
# Cell and hidden state of the LSTM
lstm_states = None
```

(continues on next page)

(continued from previous page)

```
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)
    # Note: vectorized environment resets automatically
    obs, rewards, dones, info = vec_env.step(action)
    episode_starts = dones
    vec_env.render("human")
```

3.7 CrossQ

Train a CrossQ agent on the Pendulum environment.

```
from sb3_contrib import CrossQ

model = CrossQ(
    "MlpPolicy",
    "Pendulum-v1",
    verbose=1,
    policy_kwargs=dict(
        net_arch=dict(
            pi=[256, 256],
            qf=[1024, 1024],
        )
    ),
)
model.learn(total_timesteps=5_000, log_interval=4)
model.save("crossq_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 MuJoCo locomotion tasks.

SB3's 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** (*BasePolicy*) – The policy to train, can be an instance of `ARSPolicy`, or a string from ["LinearPolicy", "MlpPolicy"]
- **env** (*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** (*int | None*) – How many of the top delta to use in each update step. Default is n_delta
- **learning_rate** (*float | Callable[[float], float]*) – Float or schedule for the step size
- **delta_std** (*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** (*dict[str, Any] | None*) – Keyword arguments to pass to the policy on creation. See *ARS Policies*
- **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** (*str | None*) – String with the directory to put tensorboard logs:
- **seed** (*int | None*) – Random seed for the training
- **verbose** (*int*) – Verbosity level: 0 no output, 1 info, 2 debug
- **device** (*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

dump_logs()

Dump information to the logger.

Return type
None

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** (*AsyncEval | None*) – The object for asynchronous evaluation of candidates.

Returns
The episodic return for each candidate.

Return type
Tensor

`get_env()`

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

Returns
The current environment

Return type
VecEnv | None

`get_parameters()`

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

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

Return type
dict[str, dict]

`get_vec_normalize_env()`

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

Returns
The `VecNormalize` env.

Return type
VecNormalize | None

`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** (*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** (*AsyncEval | None*) – The object for asynchronous evaluation of candidates.
- **progress_bar** (*bool*) – Display a progress bar using tqdm and rich.
- **self** (*SelfARS*)

Returns
the trained model

Return type
SelfARS

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** (*str | Path | BufferedIOBase*) – path to the file (or a file-like) where to load the agent from
- **env** (*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** (*device | str*) – Device on which the code should run.
- **custom_objects** (*dict[str, Any] | None*) – 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

Returns
new model instance with loaded parameters

Return type
SelfBaseAlgorithm

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** (*ndarray | dict[str, ndarray]*) – the input observation
- **state** (*tuple[ndarray, ...] | None*) – The last hidden states (can be None, used in recurrent policies)
- **episode_start** (*ndarray | None*) – 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.

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

Return type
tuple[*ndarray*, tuple[*ndarray*, ...] | None]

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

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

Parameters

- **path** (*str | Path | BufferedIOBase*) – path to the file where the rl agent should be saved
- **exclude** (*Iterable[str] | None*) – name of parameters that should be excluded in addition to the default ones
- **include** (*Iterable[str] | None*) – 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** (*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.

Warning

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

Parameters
logger (*Logger*)

Return type
None

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** (*device | str*) – Device on which the code should run.

- **load_path_or_dict** (*str | dict[str, dict]*)

Return type
None

set_random_seed(*seed=None*)

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

Parameters
seed (*int | None*)

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** (*list[int] | None*) – 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*)

Define the computation performed at every call.

Should be overridden by all subclasses.

> **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.

Parameters
obs (*Tensor | dict[str, Tensor]*)

Return type
Tensor

sb3_contrib.ars.**LinearPolicy**

alias of `ARSLinearPolicy`

sb3_contrib.ars.**MlpPolicy**

alias of ARSPolicy

CHAPTER

FIVE

CROSSQ

Implementation of CrossQ proposed in:

`Bhatt A.* & Palenicek D.* et al. Batch Normalization in Deep Reinforcement Learning for Greater Sample Efficiency and Simplicity. ICLR 2024.`

CrossQ is an algorithm that uses batch normalization to improve the sample efficiency of off-policy deep reinforcement learning algorithms. It is based on the idea of carefully introducing batch normalization layers in the critic network and dropping target networks. This results in a simpler and more sample-efficient algorithm without requiring high update-to-data ratios.

Available Policies

MlpPolicy	alias of `CrossQPolicy`

Note

Compared to the original implementation, the default network architecture for the q-value function is `[1024, 1024]` instead of `[2048, 2048]` as it provides a good compromise between speed and performance.

Note

There is currently no `CnnPolicy` for using CrossQ with images. We welcome help from contributors to add this feature.

5.1 Notes

- Original paper: https://openreview.net/pdf?id=PczQtTsTIX
- Original Implementation: https://github.com/adityab/CrossQ
- SBX (SB3 Jax) Implementation: https://github.com/araffin/sbx

5.2 Can I use?

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

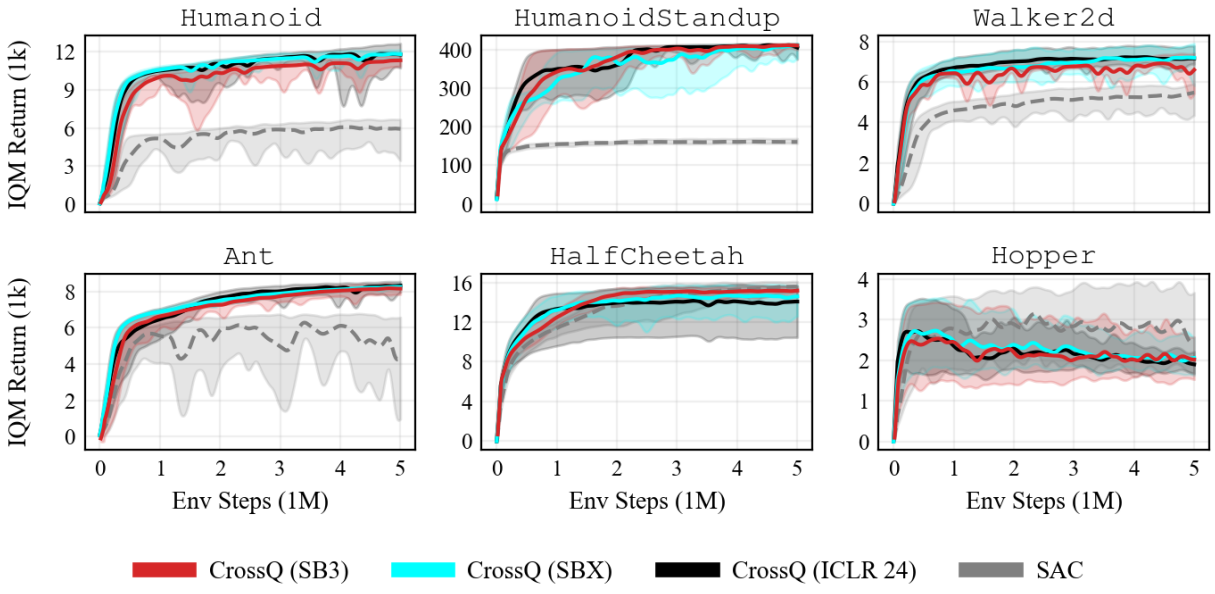
Space	Action	Observation
Discrete		✓
Box	✓	✓
MultiDiscrete		✓
MultiBinary		✓
Dict		

5.3 Example

```
from sb3_contrib import CrossQ

model = CrossQ("MlpPolicy", "Walker2d-v4")
model.learn(total_timesteps=1_000_000)
model.save("crossq_walker")
```

5.4 Results

Performance evaluation of CrossQ on six MuJoCo environments, see PR #243. Compared to results from the original paper as well as a version from SBX.



Open RL benchmark report: https://wandb.ai/openrlbenchmark/sb3-contrib/reports/SB3-Contrib-CrossQ--Vmlldzo4NTE2MTEx

5.4.1 How to replicate the results?

Clone RL-Zoo:

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

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

```
python train.py --algo crossq --env $ENV_ID --n-eval-envs 5 --eval-episodes 20 --eval-
↪freq 25000
```

Plot the results:

```
python scripts/all_plots.py -a crossq -e HalfCheetah Ant Hopper Walker2D -f logs/ -o␣
↪logs/crossq_results
python scripts/plot_from_file.py -i logs/crossq_results.pkl -latex -l CrossQ
```

5.5 Comments

This implementation is based on SB3 SAC implementation.

5.6 Parameters

class sb3_contrib.crossq.**CrossQ**(*policy*, *env*, *learning_rate=0.001*, *buffer_size=1000000*, *learning_starts=100*, *batch_size=256*, *gamma=0.99*, *train_freq=1*, *gradient_steps=1*, *action_noise=None*, *replay_buffer_class=None*, *replay_buffer_kwargs=None*, *optimize_memory_usage=False*, *n_steps=1*, *ent_coef='auto'*, *target_entropy='auto'*, *policy_delay=3*, *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*)

Implementation of Batch Normalization in Deep Reinforcement Learning (CrossQ). Paper: https://openreview.net/forum?id=PczQtTsTIX Reference implementation: https://github.com/araffin/sbx

Parameters

- **policy** (*CrossQPolicy*) – The policy model to use (MlpPolicy)
- **env** (*Env | VecEnv | str*) – The environment to learn from (if registered in Gym, can be str)
- **learning_rate** (*float | Callable[[float], float]*) – 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
- **gamma** (*float*) – the discount factor
- **train_freq** (*int | tuple[int, str]*) – 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`) Set to `-1` means to do as many gradient steps as steps done in the environment during the rollout.
- **action_noise** (*ActionNoise | None*) – 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** (*type[ReplayBuffer] | None*) – Replay buffer class to use (for instance `HerReplayBuffer`). If `None`, it will be automatically selected.
- **replay_buffer_kwargs** (*dict[str, Any] | None*) – 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
- **n_steps** (*int*) – When n_step > 1, uses n-step return (with the NStepReplayBuffer) when updating the Q-value network.
- **ent_coef** (*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_entropy** (*str | float*) – target entropy when learning `ent_coef` (`ent_coef = 'auto'`)
- **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** (*str | None*) – the log location for tensorboard (if None, no logging)
- **policy_kwargs** (*dict[str, Any] | None*) – additional arguments to be passed to the policy on creation. See *CrossQ Policies*
- **verbose** (*int*) – Verbosity level: 0 for no output, 1 for info messages (such as device or wrappers used), 2 for debug messages
- **seed** (*int | None*) – Seed for the pseudo random generators
- **device** (*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
- **policy_delay** (*int*)

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** (*ActionNoise | None*) – 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** (*int | None*) – Log data every log_interval episodes

Returns

Return type
RolloutReturn

dump_logs()

Write log data.

Return type
None

get_env()

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

Returns
The current environment

Return type
VecEnv | None

get_parameters()

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

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

Return type
dict[str, dict]

get_vec_normalize_env()

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

Returns
The VecNormalize env.

Return type
VecNormalize | None

learn(*total_timesteps*, *callback=None*, *log_interval=4*, *tb_log_name='CrossQ'*, *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 Note: it is a lower bound, see issue #1150
- **callback** (*None | Callable | list[BaseCallback] | BaseCallback*) – callback(s) called at every step with state of the algorithm.
- **log_interval** (*int*) – for on-policy algos (e.g., PPO, A2C, ...) this is the number of training iterations (i.e., log_interval * n_steps * n_envs timesteps) before logging; for off-policy algos (e.g., TD3, SAC, ...) this is 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.
- **self** (*SelfCrossQ*)

Returns
the trained model

Return type
SelfCrossQ

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** (*str | Path | BufferedIOBase*) – path to the file (or a file-like) where to load the agent from
- **env** (*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** (*device | str*) – Device on which the code should run.
- **custom_objects** (*dict[str, Any] | None*) – 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

Returns
new model instance with loaded parameters

Return type
SelfBaseAlgorithm

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

Load a replay buffer from a pickle file.

Parameters

- **path** (*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** (*ndarray | dict[str, ndarray]*) – the input observation
- **state** (*tuple[ndarray, ...] | None*) – The last hidden states (can be None, used in recurrent policies)
- **episode_start** (*ndarray | None*) – 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.

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

Return type
tuple[*ndarray*, tuple[*ndarray*, ...] | None]

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

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

Parameters

- **path** (*str | Path | BufferedIOBase*) – path to the file where the rl agent should be saved
- **exclude** (*Iterable[str] | None*) – name of parameters that should be excluded in addition to the default ones
- **include** (*Iterable[str] | None*) – 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 (*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** (*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.

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

Parameters
logger (*Logger*)

Return type
None

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** (*device | str*) – Device on which the code should run.
- **load_path_or_dict** (*str | dict[str, Tensor]*)

Return type
None

set_random_seed(*seed=None*)

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

Parameters
seed (*int | None*)

Return type
None

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

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

Parameters

- **gradient_steps** (*int*)
- **batch_size** (*int*)

Return type
None

5.7 CrossQ Policies

sb3_contrib.crossq.**MlpPolicy**

alias of CrossQPolicy

class sb3_contrib.crossq.policies.**CrossQPolicy**(*observation_space*, *action_space*, *lr_schedule*, *net_arch=None*, *activation_fn=<class 'torch.nn.modules.activation.ReLU'>*, *batch_norm=True*, *batch_norm_momentum=0.01*, *batch_norm_eps=0.001*, *renorm_warmup_steps=100000*, *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_critics=2*, *share_features_extractor=False*)

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

Parameters

- **observation_space** (*Space*) – Observation space
- **action_space** (*Box*) – Action space
- **lr_schedule** (*Callable[[float], float]*) – Learning rate schedule (could be constant)
- **net_arch** (*list[int] | dict[str, list[int]] | None*) – The specification of the policy and value networks.
- **activation_fn** (*type[Module]*) – Activation function
- **batch_norm** (*bool*) – Whether to use Batch Renorm layers (default=True)
- **batch_norm_momentum** (*float*) – The rate of convergence for the batch renormalization statistics
- **batch_norm_eps** (*float*) – A small value added to the variance to prevent division by zero

- **renorm_warmup_steps** (*int*) – Number of steps to warm up BatchRenorm statistics before the running statistics are used for normalization.
- **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** (*dict[str, Any] | None*) – 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** (*dict[str, Any] | None*) – Additional keyword arguments, excluding the learning rate, to pass to the optimizer
- **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*)

Define the computation performed at every call.

Should be overridden by all subclasses.

> **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.

Parameters

- **obs** (*Tensor | dict[str, Tensor]*)
- **deterministic** (*bool*)

Return type

Tensor

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.

Parameters
`mode` (*bool*) – if true, set to training mode, else set to evaluation mode

Return type
None

CHAPTER

SIX

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`

6.1 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/

6.2 Can I use?

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

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

Warning

You must use `MaskableEvalCallback` from `sb3_contrib.common.maskable.callbacks` instead of the base `EvalCallback` to properly evaluate a model with action masks. Similarly, you must use `evaluate_policy` from `sb3_contrib.common.maskable.evaluation` instead of the SB3 one.

Warning

In order to use `SubprocVecEnv` with `MaskablePPO`, you must implement the `action_masks` inside the environment (`ActionMasker` cannot be used). You can have a look at the built-in environments with invalid action masks to have a working example.

6.3 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
# This is a drop-in replacement for EvalCallback
from sb3_contrib.common.maskable.callbacks import MaskableEvalCallback


env = InvalidActionEnvDiscrete(dim=80, n_invalid_actions=60)
model = MaskablePPO("MlpPolicy", env, gamma=0.4, seed=32, verbose=1)
model.learn(5_000)

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, reward, terminated, truncated, info = env.step(action)
```

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).

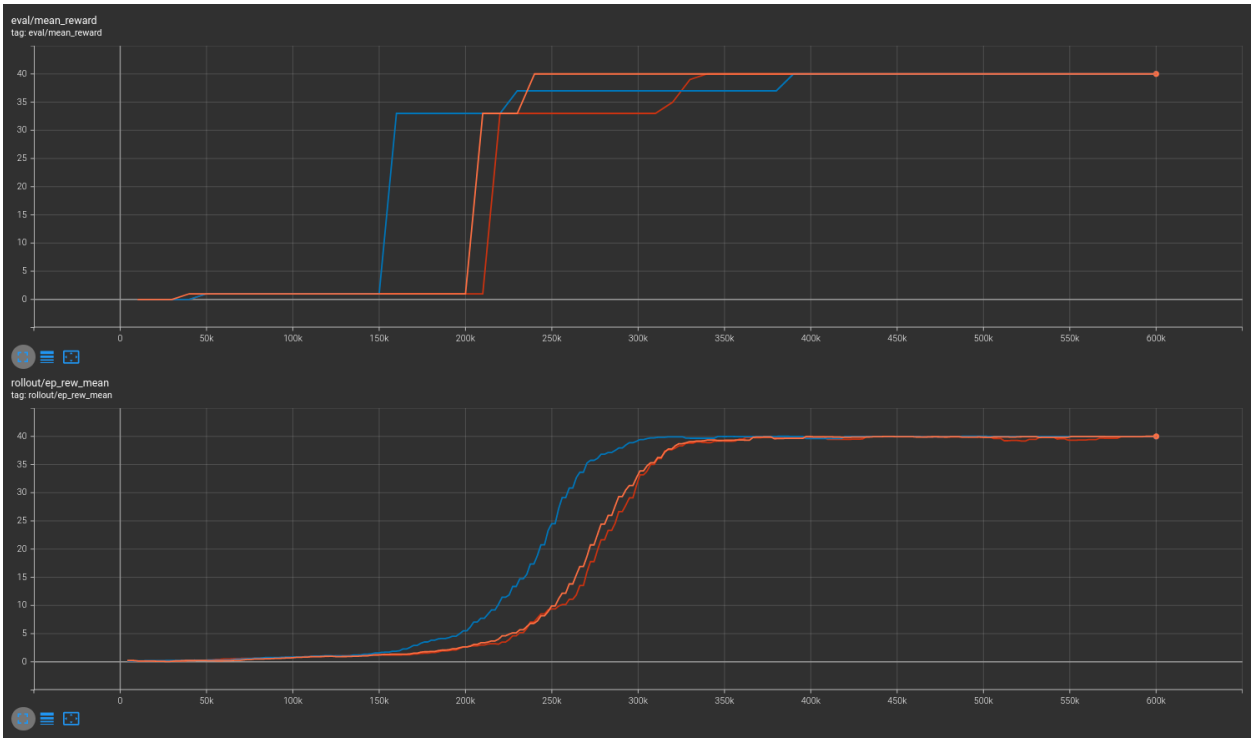
```
import gymnasium as gym
import numpy as np

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
# 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)
```

6.4 Results

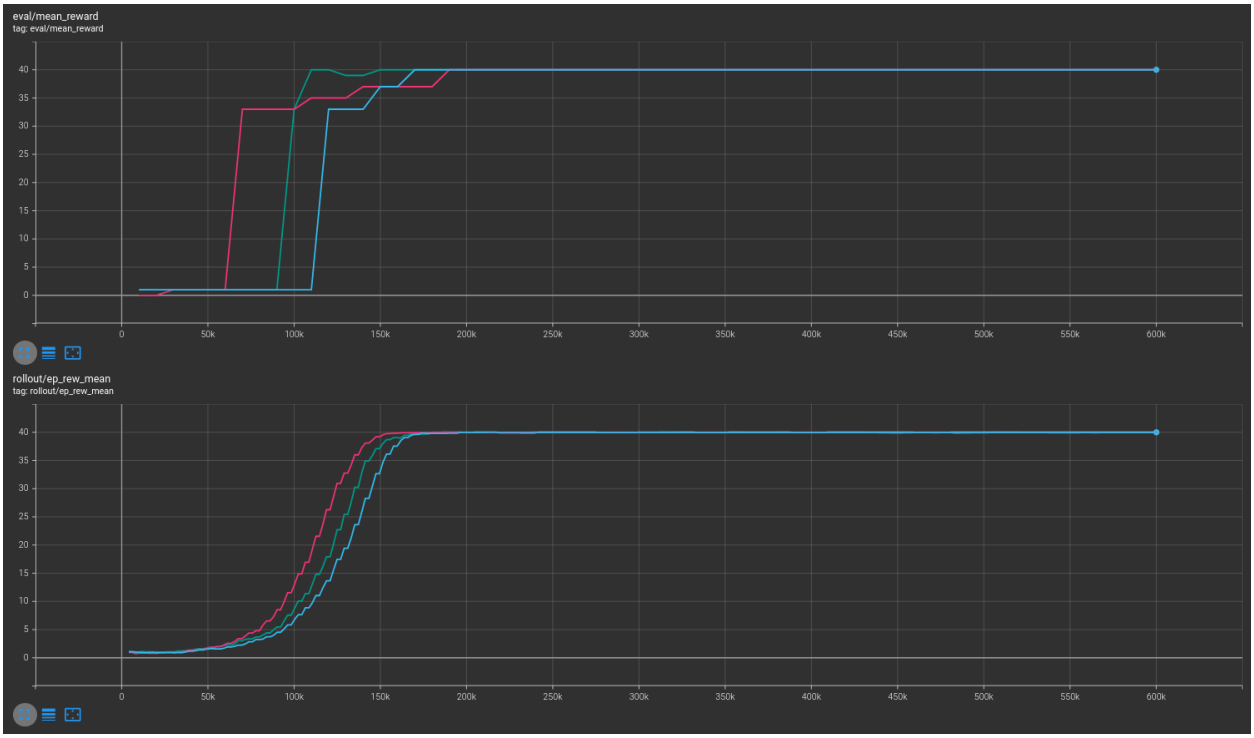
Results are shown for two MicroRTS benchmarks: MicrortsMining4x4F9-v0 (600K steps) and MicrortsMining10x10F9-v0 (1.5M steps). For each, models were trained with and without masking, using 3 seeds.

6.4.1 4x4

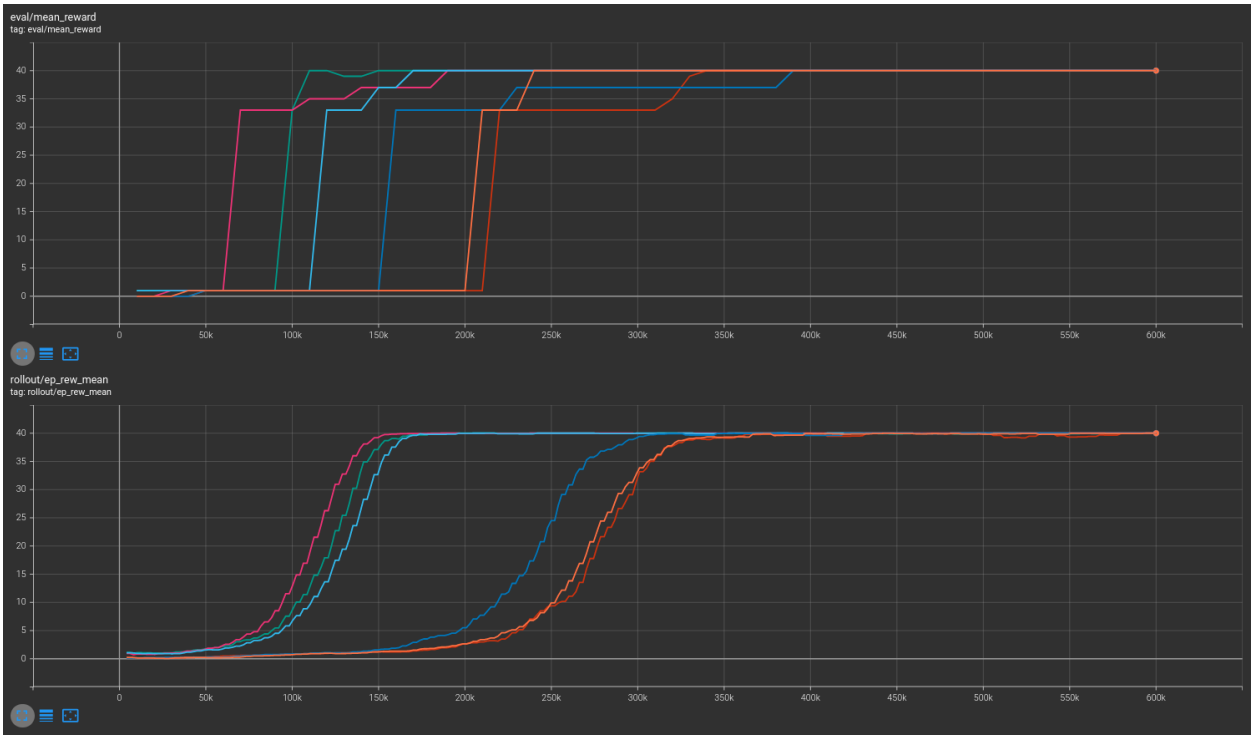
No masking



With masking

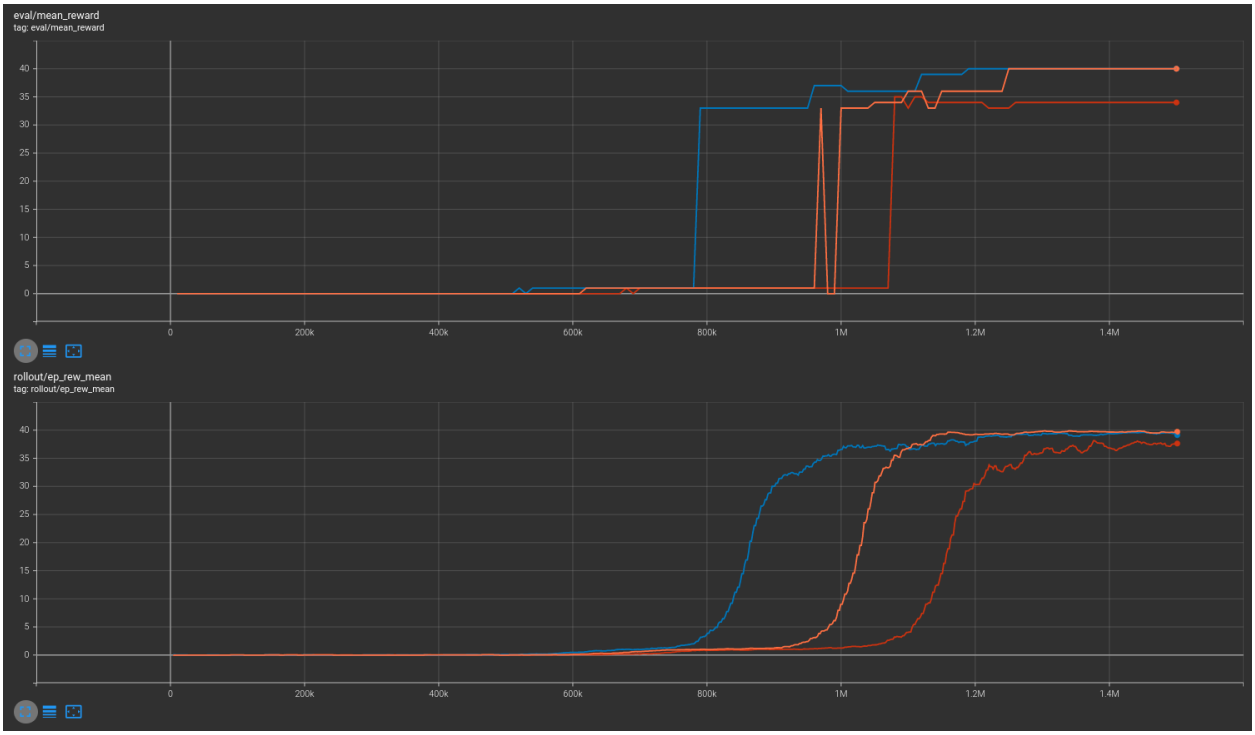


Combined

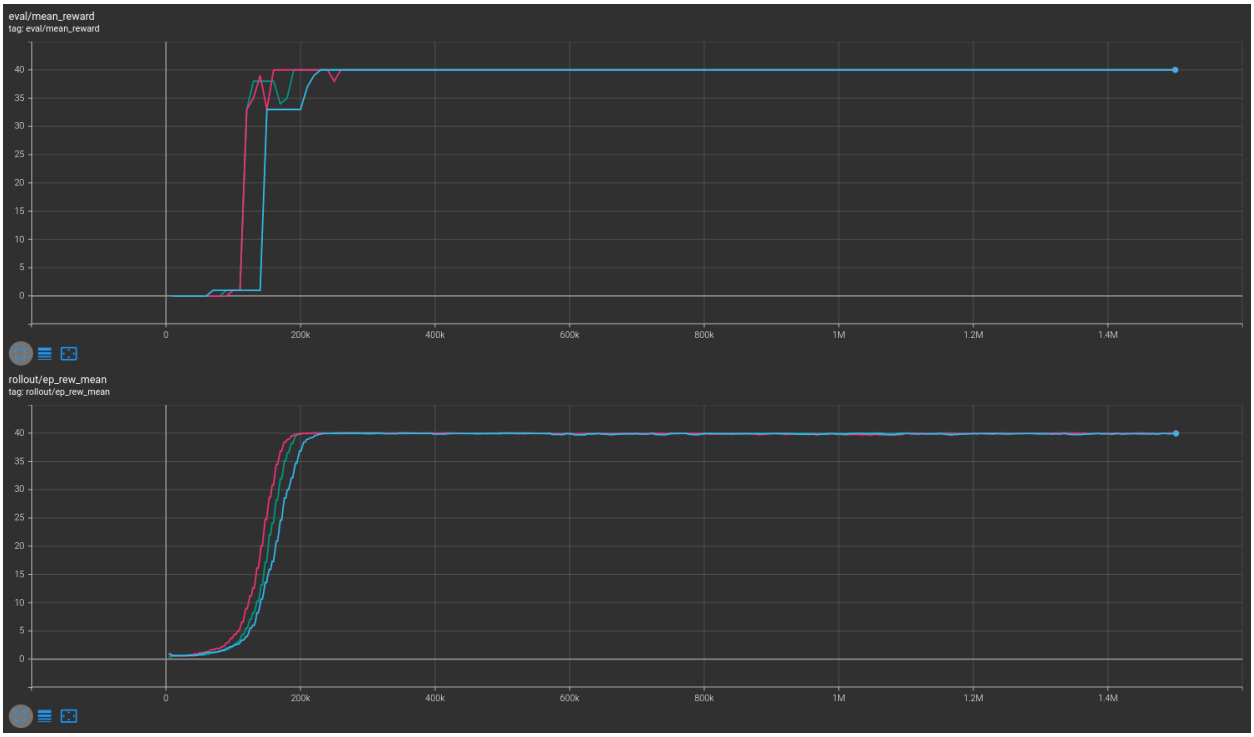


6.4.2 10x10

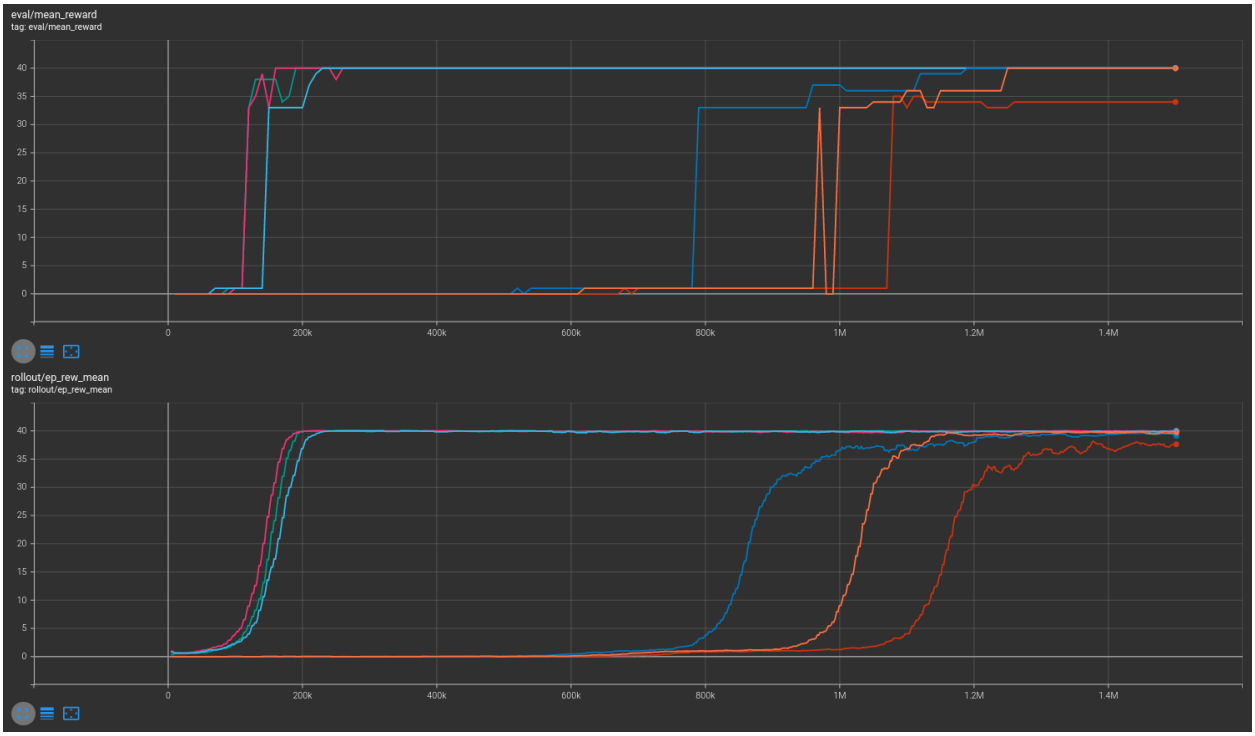
No masking



With masking



Combined



More information may be found in the associated PR.

6.4.3 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
```

(continues on next page)

(continued from previous page)

```
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
```

6.5 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*, *rollout_buffer_class=None*, *rollout_buffer_kwargs=None*, *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** (*MaskableActorCriticPolicy*) – The policy model to use (MlpPolicy, CnnPolicy, ...)
- **env** (*Env | VecEnv | str*) – The environment to learn from (if registered in Gym, can be str)
- **learning_rate** (*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** (*int | None*) – 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** (*float | Callable[[float], float]*) – Clipping parameter, it can be a function of the current progress remaining (from 1 to 0).
- **clip_range_vf** (*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** (*float | None*) – 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** (*str | None*) – the log location for tensorboard (if None, no logging)
- **policy_kwargs** (*dict[str, Any] | None*) – additional arguments to be passed to the policy on creation. See *MaskablePPO Policies*
- **verbose** (*int*) – the verbosity level: 0 no output, 1 info, 2 debug
- **seed** (*int | None*) – Seed for the pseudo random generators
- **device** (*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
- **rollout_buffer_class** (*type[RolloutBuffer] | None*)
- **rollout_buffer_kwargs** (*dict[str, Any] | None*)

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
- **n_rollout_steps** (*int*)

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

Return type
bool

dump_logs(*iteration=0*)

Write log.

Parameters
iteration (*int*) – Current logging iteration

Return type
None

get_env()

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

Returns
The current environment

Return type
VecEnv | None

get_parameters()

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

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

Return type
dict[str, dict]

get_vec_normalize_env()

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

Returns
The `VecNormalize` env.

Return type
VecNormalize | None

learn(*total_timesteps*, *callback=None*, *log_interval=1*, *tb_log_name='MaskablePPO'*, *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 Note: it is a lower bound, see issue #1150

- **callback** (*None | Callable | list[BaseCallback] | BaseCallback*) – callback(s) called at every step with state of the algorithm.
- **log_interval** (*int*) – for on-policy algos (e.g., PPO, A2C, ...) this is the number of training iterations (i.e., log_interval * n_steps * n_envs timesteps) before logging; for off-policy algos (e.g., TD3, SAC, ...) this is 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.
- **self** (*SelfMaskablePPO*)
- **use_masking** (*bool*)

Returns
the trained model

Return type
SelfMaskablePPO

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** (*str | Path | BufferedIOBase*) – path to the file (or a file-like) where to load the agent from
- **env** (*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** (*device | str*) – Device on which the code should run.
- **custom_objects** (*dict[str, Any] | None*) – 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

Returns
new model instance with loaded parameters

Return type
SelfBaseAlgorithm

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 | dict[str, ndarray]*) – the input observation
- **state** (*tuple[ndarray, ...] | None*) – The last hidden states (can be None, used in recurrent policies)
- **episode_start** (*ndarray | None*) – 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.
- **action_masks** (*ndarray | None*)

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

Return type
tuple[*ndarray*, tuple[*ndarray*, ...] | None]

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

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

Parameters

- **path** (*str | Path | BufferedIOBase*) – path to the file where the rl agent should be saved
- **exclude** (*Iterable[str] | None*) – name of parameters that should be excluded in addition to the default ones
- **include** (*Iterable[str] | None*) – 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** (*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.

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

Parameters
logger (*Logger*)

Return type
None

`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** (*device | str*) – Device on which the code should run.
- **load_path_or_dict** (*str | dict[str, Tensor]*)

Return type
None

`set_random_seed`(*seed=None*)

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

Parameters
seed (*int | None*)

Return type
None

`train`()

Update policy using the currently gathered rollout buffer.

Return type
None

6.6 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** (*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** (*dict[str, Any] | None*) – 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** (*dict[str, Any] | None*) – 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
- **action_masks** (*Tensor | None*)

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

Return type
tuple[*Tensor*, *Tensor*, *Tensor* | None]

`extract_features`(*obs*, *features_extractor=None*)

Preprocess the observation if needed and extract features.

Parameters

- **obs** (*Tensor | dict[str, Tensor]*) – Observation
- **features_extractor** (*BaseFeaturesExtractor | None*) – The features extractor to use. If None, then `self.features_extractor` is used.

Returns
The extracted features. If features extractor is not shared, returns a tuple with the features for the actor and the features for the critic.

Return type
Tensor | tuple[*Tensor*, *Tensor*]

`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** (*ndarray | None*) – Action masks to apply to the action distribution

Returns
action, value and log probability of the action

Return type
tuple[*Tensor*, *Tensor*, *Tensor*]

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

Get the current policy distribution given the observations.

Parameters

- **obs** (*Tensor | dict[str, Tensor]*) – Observation
- **action_masks** (*ndarray | None*) – Actions' mask

Returns
the action distribution.

Return type
MaskableDistribution

`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 | dict[str, ndarray]*) – the input observation
- **state** (*tuple[ndarray, ...] | None*) – The last states (can be None, used in recurrent policies)

- **episode_start** (*ndarray | None*) – The last masks (can be None, used in recurrent policies)
- **deterministic** (*bool*) – Whether or not to return deterministic actions.
- **action_masks** (*ndarray | None*) – Action masks to apply to the action distribution

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

Return type
tuple[*ndarray*, tuple[*ndarray*, ...] | None]

predict_values(*obs*)

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

Parameters
obs (*Tensor | dict[str, Tensor]*) – Observation

Returns
the estimated values.

Return type
Tensor

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** (*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** (*dict[str, Any] | None*) – 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** (*dict[str, Any] | None*) – 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** (*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** (*dict[str, Any] | None*) – 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** (*dict[str, Any] | None*) – Additional keyword arguments, excluding the learning rate, to pass to the optimizer

CHAPTER

SEVEN

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`

7.1 Notes

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

7.2 Can I use?

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

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

7.3 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)

vec_env = model.get_env()
mean_reward, std_reward = evaluate_policy(model, vec_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 = vec_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 = vec_env.step(action)
    episode_starts = dones
    vec_env.render("human")
```

7.4 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

7.4.1 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
```

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
```

7.5 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** (*ActorCriticPolicy*) – The policy model to use (MlpPolicy, CnnPolicy, ...)
- **env** (*Env | VecEnv | str*) – The environment to learn from (if registered in Gym, can be str)
- **learning_rate** (*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** (*int | None*) – 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** (*float | Callable[[float], float]*) – Clipping parameter, it can be a function of the current progress remaining (from 1 to 0).
- **clip_range_vf** (*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** (*float | None*) – 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** (*str | None*) – the log location for tensorboard (if None, no logging)
- **policy_kwargs** (*dict[str, Any] | None*) – additional arguments to be passed to the policy on creation. See *RecurrentPPO Policies*
- **verbose** (*int*) – the verbosity level: 0 no output, 1 info, 2 debug
- **seed** (*int | None*) – Seed for the pseudo random generators
- **device** (*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
- **use_sde** (*bool*)
- **sde_sample_freq** (*int*)

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
- **n_rollout_steps** (*int*)

Returns

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

Return type

bool

dump_logs(*iteration=0*)

Write log.

Parameters
iteration (*int*) – Current logging iteration

Return type
None

`get_env()`

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

Returns
The current environment

Return type
VecEnv | None

`get_parameters()`

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

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

Return type
dict[str, dict]

`get_vec_normalize_env()`

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

Returns
The `VecNormalize` env.

Return type
VecNormalize | None

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 Note: it is a lower bound, see issue #1150
- **callback** (*None | Callable | list[BaseCallback] | BaseCallback*) – callback(s) called at every step with state of the algorithm.
- **log_interval** (*int*) – for on-policy algos (e.g., PPO, A2C, ...) this is the number of training iterations (i.e., log_interval * n_steps * n_envs timesteps) before logging; for off-policy algos (e.g., TD3, SAC, ...) this is 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.
- **self** (*SelfRecurrentPPO*)

Returns
the trained model

Return type
SelfRecurrentPPO

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** (*str | Path | BufferedIOBase*) – path to the file (or a file-like) where to load the agent from
- **env** (*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** (*device | str*) – Device on which the code should run.
- **custom_objects** (*dict[str, Any] | None*) – 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

Returns
new model instance with loaded parameters

Return type
SelfBaseAlgorithm

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** (*ndarray | dict[str, ndarray]*) – the input observation
- **state** (*tuple[ndarray, ...] | None*) – The last hidden states (can be None, used in recurrent policies)
- **episode_start** (*ndarray | None*) – 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.

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

Return type
tuple[*ndarray*, tuple[*ndarray*, ...] | None]

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

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

Parameters

- **path** (*str | Path | BufferedIOBase*) – path to the file where the rl agent should be saved
- **exclude** (*Iterable[str] | None*) – name of parameters that should be excluded in addition to the default ones
- **include** (*Iterable[str] | None*) – 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** (*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.

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

Parameters
logger (*Logger*)

Return type
None

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** (*device | str*) – Device on which the code should run.

- **load_path_or_dict** (*str | dict[str, Tensor]*)

Return type
None

set_random_seed(*seed=None*)

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

Parameters
seed (*int | None*)

Return type
None

train()

Update policy using the currently gathered rollout buffer.

Return type
None

7.6 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** (*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** (*dict[str, Any] | None*) – 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** (*dict[str, Any] | None*) – 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** (*dict[str, Any] | None*) – 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).

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

Return type
tuple[*Tensor*, *Tensor*, *Tensor*]

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

Returns
action, value and log probability of the action

Return type
tuple[*Tensor*, *Tensor*, *Tensor*, *RNNStates*]

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).

Returns
the action distribution and new hidden states.

Return type
tuple[*Distribution*, tuple[*Tensor*, ...]]

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** (*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.
- **state** (*tuple[ndarray, ...] | None*)
- **episode_start** (*ndarray | None*)

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

Return type
tuple[*ndarray*, tuple[*ndarray*, ...] | None]

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).

Returns
the estimated values.

Return type
Tensor

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** (*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** (*dict[str, Any] | None*) – 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** (*dict[str, Any] | None*) – 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** (*dict[str, Any] | None*) – 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=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*)

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** (*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** (*dict[str, Any] | None*) – 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** (*dict[str, Any] | None*) – 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** (*dict[str, Any] | None*) – Additional keyword arguments to pass the the LSTM constructor.

CHAPTER

EIGHT

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.

8.1 Notes

- Original paper: https://arxiv.org/abs/1710.10044
- 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

8.2 Can I use?

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

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

8.3 Example

```
import gymnasium as gym

from sb3_contrib import QRDQN

env = gym.make("CartPole-v1", render_mode="human")

policy_kwargs = dict(n_quantiles=50)
model = QRDQN("MlpPolicy", env, policy_kwargs=policy_kwargs, verbose=1)
model.learn(total_timesteps=10_000, 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, terminated, truncated, info = env.step(action)
    env.render()
    if terminated or truncated:
        obs, _ = env.reset()
```

8.4 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

8.4.1 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
```

8.5 Parameters

class sb3_contrib.qrdqn.**QRDQN**(*policy*, *env*, *learning_rate=5e-05*, *buffer_size=1000000*, *learning_starts=100*, *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*, *n_steps=1*, *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 (except for the `learning_starts` parameter).

Parameters

- **policy** (*QRDQNPolicy*) – The policy model to use (MlpPolicy, CnnPolicy, ...)
- **env** (*Env | VecEnv | str*) – The environment to learn from (if registered in Gym, can be str)
- **learning_rate** (*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 | tuple[int, str]*) – 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** (*type[ReplayBuffer] | None*) – Replay buffer class to use (for instance `HerReplayBuffer`). If `None`, it will be automatically selected.
- **replay_buffer_kwargs** (*dict[str, Any] | None*) – 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
- **n_steps** (*int*) – When n_step > 1, uses n-step return (with the NStepReplayBuffer) when updating the Q-value network.
- **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** (*float | None*) – 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** (*str | None*) – the log location for tensorboard (if None, no logging)
- **policy_kwargs** (*dict[str, Any] | None*) – additional arguments to be passed to the policy on creation. See *QR-DQN Policies*
- **verbose** (*int*) – the verbosity level: 0 no output, 1 info, 2 debug
- **seed** (*int | None*) – Seed for the pseudo random generators
- **device** (*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** (*ActionNoise | None*) – 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** (*int | None*) – Log data every `log_interval` episodes

Returns

Return type
RolloutReturn

`dump_logs()`

Write log data.

Return type
None

`get_env()`

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

Returns
The current environment

Return type
VecEnv | None

`get_parameters()`

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

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

Return type
dict[str, dict]

`get_vec_normalize_env()`

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

Returns
The `VecNormalize` env.

Return type
VecNormalize | None

`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 Note: it is a lower bound, see issue #1150
- **callback** (*None | Callable | list[BaseCallback] | BaseCallback*) – callback(s) called at every step with state of the algorithm.
- **log_interval** (*int*) – for on-policy algos (e.g., PPO, A2C, ...) this is the number of training iterations (i.e., log_interval * n_steps * n_envs timesteps) before logging; for off-policy algos (e.g., TD3, SAC, ...) this is 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.
- **self** (*SelfQRDQN*)

Returns
the trained model

Return type
SelfQRDQN

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** (*str | Path | BufferedIOBase*) – path to the file (or a file-like) where to load the agent from
- **env** (*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** (*device | str*) – Device on which the code should run.
- **custom_objects** (*dict[str, Any] | None*) – 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

Returns
new model instance with loaded parameters

Return type
SelfBaseAlgorithm

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

Load a replay buffer from a pickle file.

Parameters

- **path** (*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** (*ndarray | dict[str, ndarray]*) – the input observation
- **state** (*tuple[ndarray, ...] | None*) – The last hidden states (can be None, used in recurrent policies)
- **episode_start** (*ndarray | None*) – 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.

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

Return type
tuple[*ndarray*, tuple[*ndarray*, ...] | None]

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

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

Parameters

- **path** (*str | Path | BufferedIOBase*) – path to the file where the rl agent should be saved
- **exclude** (*Iterable[str] | None*) – name of parameters that should be excluded in addition to the default ones
- **include** (*Iterable[str] | None*) – 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 (*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** (*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.

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

Parameters
logger (*Logger*)

Return type
None

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** (*device | str*) – Device on which the code should run.
- **load_path_or_dict** (*str | dict[str, Tensor]*)

Return type
None

set_random_seed(*seed=None*)

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

Parameters
seed (*int | None*)

Return type
None

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

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

Parameters

- **gradient_steps** (*int*)
- **batch_size** (*int*)

Return type
None

8.6 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** (*Discrete*) – Action space
- **lr_schedule** (*Callable[[float], float]*) – Learning rate schedule (could be constant)
- **n_quantiles** (*int*) – Number of quantiles
- **net_arch** (*list[int] | None*) – The specification of the network architecture.
- **activation_fn** (*type[Module]*) – Activation function
- **features_extractor_class** (*type[BaseFeaturesExtractor]*) – Features extractor to use.
- **features_extractor_kwargs** (*dict[str, Any] | None*) – 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** (*dict[str, Any] | None*) – Additional keyword arguments, excluding the learning rate, to pass to the optimizer

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

Define the computation performed at every call.

Should be overridden by all subclasses.

> **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.

Parameters

- **obs** (*Tensor | dict[str, Tensor]*)

- **deterministic** (*bool*)

Return type
Tensor

set_training_mode(*mode*)

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

Parameters
mode (*bool*)

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** (*Discrete*) – Action space
- **lr_schedule** (*Callable[[float], float]*) – Learning rate schedule (could be constant)
- **n_quantiles** (*int*) – Number of quantiles
- **net_arch** (*list[int] | None*) – 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** (*dict[str, Any] | None*) – Additional keyword arguments, excluding the learning rate, to pass to the optimizer
- **features_extractor_kwargs** (*dict[str, Any] | None*)

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** (*Discrete*) – Action space
- **lr_schedule** (*Callable[[float], float]*) – Learning rate schedule (could be constant)
- **n_quantiles** (*int*) – Number of quantiles
- **net_arch** (*list[int] | None*) – 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** (*dict[str, Any] | None*) – Additional keyword arguments, excluding the learning rate, to pass to the optimizer
- **features_extractor_kwargs** (*dict[str, Any] | None*)

CHAPTER

NINE

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.

9.1 Notes

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

9.2 Can I use?

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

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

9.3 Example

```
import gymnasium as gym
import numpy as np

from sb3_contrib import TQC

env = gym.make("Pendulum-v1", render_mode="human")

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=10_000, 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, terminated, truncated, info = env.step(action)
    env.render()
    if terminated or truncated:
        obs, _ = env.reset()
```

9.4 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

9.4.1 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
```

9.5 Comments

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

9.6 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*, *n_steps=1*, *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** (*TQCPolicy*) – The policy model to use (MlpPolicy, CnnPolicy, ...)
- **env** (*Env | VecEnv | str*) – The environment to learn from (if registered in Gym, can be str)
- **learning_rate** (*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 | tuple[int, str]*) – 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** (*ActionNoise | None*) – 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** (*type[ReplayBuffer] | None*) – Replay buffer class to use (for instance `HerReplayBuffer`). If `None`, it will be automatically selected.
- **replay_buffer_kwargs** (*dict[str, Any] | None*) – 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
- **n_steps** (*int*) – When n_step > 1, uses n-step return (with the NStepReplayBuffer) when updating the Q-value network.
- **ent_coef** (*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** (*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** (*str | None*) – the log location for tensorboard (if None, no logging)
- **policy_kwargs** (*dict[str, Any] | None*) – additional arguments to be passed to the policy on creation. See *TQC Policies*
- **verbose** (*int*) – the verbosity level: 0 no output, 1 info, 2 debug
- **seed** (*int | None*) – Seed for the pseudo random generators
- **device** (*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** (*ActionNoise | None*) – 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** (*int | None*) – Log data every `log_interval` episodes

Returns

Return type
RolloutReturn

dump_logs()

Write log data.

Return type
None

get_env()

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

Returns
The current environment

Return type
VecEnv | None

`get_parameters()`

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

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

Return type
dict[str, dict]

`get_vec_normalize_env()`

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

Returns
The `VecNormalize` env.

Return type
VecNormalize | None

`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 Note: it is a lower bound, see issue #1150
- **callback** (*None | Callable | list[BaseCallback] | BaseCallback*) – callback(s) called at every step with state of the algorithm.
- **log_interval** (*int*) – for on-policy algos (e.g., PPO, A2C, ...) this is the number of training iterations (i.e., log_interval * n_steps * n_envs timesteps) before logging; for off-policy algos (e.g., TD3, SAC, ...) this is 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.
- **self** (*SelfTQC*)

Returns
the trained model

Return type
SelfTQC

`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** (*str | Path | BufferedIOBase*) – path to the file (or a file-like) where to load the agent from
- **env** (*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** (*device | str*) – Device on which the code should run.
- **custom_objects** (*dict[str, Any] | None*) – 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

Returns
new model instance with loaded parameters

Return type
SelfBaseAlgorithm

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

Load a replay buffer from a pickle file.

Parameters

- **path** (*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** (*ndarray | dict[str, ndarray]*) – the input observation
- **state** (*tuple[ndarray, ...] | None*) – The last hidden states (can be None, used in recurrent policies)
- **episode_start** (*ndarray | None*) – 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.

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

Return type
tuple[*ndarray*, tuple[*ndarray*, ...] | None]

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

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

Parameters

- **path** (*str | Path | BufferedIOBase*) – path to the file where the rl agent should be saved
- **exclude** (*Iterable[str] | None*) – name of parameters that should be excluded in addition to the default ones
- **include** (*Iterable[str] | None*) – 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 (*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** (*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.

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

Parameters
logger (*Logger*)

Return type
None

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** (*device | str*) – Device on which the code should run.
- **load_path_or_dict** (*str | dict[str, Tensor]*)

Return type
None

set_random_seed(*seed=None*)

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

Parameters
seed (*int | None*)

Return type
None

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

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

Parameters

- **gradient_steps** (*int*)
- **batch_size** (*int*)

Return type
None

9.7 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** (*Box*) – Action space
- **lr_schedule** (*Callable[[float], float]*) – Learning rate schedule (could be constant)

- **net_arch** (*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** (*dict[str, Any] | None*) – 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** (*dict[str, Any] | None*) – 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*)

Define the computation performed at every call.

Should be overridden by all subclasses.

> **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.

Parameters

- **obs** (*Tensor | dict[str, Tensor]*)
- **deterministic** (*bool*)

Return type

Tensor

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. :param mode: if true, set to training mode, else set to evaluation mode

Parameters
mode (*bool*)

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** (*Box*) – Action space
- **lr_schedule** (*Callable[[float], float]*) – Learning rate schedule (could be constant)
- **net_arch** (*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** (*dict[str, Any] | None*) – 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)
- **features_extractor_kwargs** (*dict[str, Any] | None*)

CHAPTER

TEN

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`

10.1 Notes

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

10.2 Can I use?

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

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

10.3 Example

```
import gymnasium as gym
import numpy as np

from sb3_contrib import TRPO

env = gym.make("Pendulum-v1", render_mode="human")

model = TRPO("MlpPolicy", env, verbose=1)
model.learn(total_timesteps=10_000, 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, terminated, truncated, info = env.step(action)
    env.render()
    if terminated or truncated:
        obs, _ = env.reset()
```

10.4 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

10.4.1 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 trpo --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
```

10.5 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*, *rollout_buffer_class=None*, *rollout_buffer_kwargs=None*, *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** (*ActorCriticPolicy*) – The policy model to use (MlpPolicy, CnnPolicy, ...)
- **env** (*Env | VecEnv | str*) – The environment to learn from (if registered in Gym, can be str)
- **learning_rate** (*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)
- **rollout_buffer_class** (*type[RolloutBuffer] | None*) – Rollout buffer class to use. If None, it will be automatically selected.
- **rollout_buffer_kwargs** (*dict[str, Any] | None*) – Keyword arguments to pass to the rollout buffer on creation
- **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** (*str | None*) – the log location for tensorboard (if None, no logging)
- **policy_kwargs** (*dict[str, Any] | None*) – additional arguments to be passed to the policy on creation. See *TRPO Policies*
- **verbose** (*int*) – the verbosity level: 0 no output, 1 info, 2 debug
- **seed** (*int | None*) – Seed for the pseudo random generators
- **device** (*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

Returns

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

Return type

bool

dump_logs(*iteration=0*)

Write log.

Parameters

iteration (*int*) – Current logging iteration

Return type
None

`get_env()`

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

Returns
The current environment

Return type
VecEnv | None

`get_parameters()`

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

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

Return type
dict[str, dict]

`get_vec_normalize_env()`

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

Returns
The `VecNormalize` env.

Return type
VecNormalize | None

`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

Returns
Hessian-vector dot product (with damping)

Return type
Tensor

`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 Note: it is a lower bound, see issue #1150
- **callback** (*None | Callable | list[BaseCallback] | BaseCallback*) – callback(s) called at every step with state of the algorithm.

- **log_interval** (*int*) – for on-policy algos (e.g., PPO, A2C, ...) this is the number of training iterations (i.e., log_interval * n_steps * n_envs timesteps) before logging; for off-policy algos (e.g., TD3, SAC, ...) this is 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.
- **self** (*SelfTRPO*)

Returns
the trained model

Return type
SelfTRPO

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** (*str | Path | BufferedIOBase*) – path to the file (or a file-like) where to load the agent from
- **env** (*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** (*device | str*) – Device on which the code should run.
- **custom_objects** (*dict[str, Any] | None*) – 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

Returns
new model instance with loaded parameters

Return type
SelfBaseAlgorithm

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** (*ndarray | dict[str, ndarray]*) – the input observation

- **state** (*tuple[ndarray, ...] | None*) – The last hidden states (can be None, used in recurrent policies)
- **episode_start** (*ndarray | None*) – 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.

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

Return type
tuple[*ndarray*, tuple[*ndarray*, ...] | None]

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

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

Parameters

- **path** (*str | Path | BufferedIOBase*) – path to the file where the rl agent should be saved
- **exclude** (*Iterable[str] | None*) – name of parameters that should be excluded in addition to the default ones
- **include** (*Iterable[str] | None*) – 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** (*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.

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

Parameters
logger (*Logger*)

Return type
None

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** (*device | str*) – Device on which the code should run.
- **load_path_or_dict** (*str | dict[str, Tensor]*)

Return type
None

set_random_seed(*seed=None*)

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

Parameters
seed (*int | None*)

Return type
None

train()

Update policy using the currently gathered rollout buffer.

Return type
None

10.6 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** (*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** (*dict[str, Any] | None*) – 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** (*dict[str, Any] | None*) – 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 | dict[str, Tensor]*) – Observation
- **actions** (*Tensor*) – Actions

Returns

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

Return type

tuple[*Tensor*, *Tensor*, *Tensor* | None]

extract_features(*obs*, *features_extractor=None*)

Preprocess the observation if needed and extract features.

Parameters

- **obs** (*Tensor | dict[str, Tensor]*) – Observation
- **features_extractor** (*BaseFeaturesExtractor | None*) – The features extractor to use. If None, then `self.features_extractor` is used.

Returns
The extracted features. If features extractor is not shared, returns a tuple with the features for the actor and the features for the critic.

Return type
Tensor | tuple[*Tensor*, *Tensor*]

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

Returns
action, value and log probability of the action

Return type
tuple[*Tensor*, *Tensor*, *Tensor*]

get_distribution(*obs*)

Get the current policy distribution given the observations.

Parameters
obs (*Tensor | dict[str, Tensor]*)

Returns
the action distribution.

Return type
Distribution

predict_values(*obs*)

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

Parameters
obs (*Tensor | dict[str, Tensor]*) – Observation

Returns
the estimated values.

Return type
Tensor

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** (*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** (*dict[str, Any] | None*) – 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** (*dict[str, Any] | None*) – 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** (*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** (*dict[str, Any] | None*) – 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** (*dict[str, Any] | None*) – Additional keyword arguments, excluding the learning rate, to pass to the optimizer

CHAPTER

ELEVEN

TORCH LAYERS

class sb3_contrib.common.torch_layers.**BatchRenorm**(*num_features*, *eps=0.001*, *momentum=0.01*, *affine=True*, *warmup_steps=100000*)

BatchRenorm Module (https://arxiv.org/abs/1702.03275). Adapted to Pytorch from https://github.com/araffin/sbx/blob/master/sbx/common/jax_layers.py

BatchRenorm is an improved version of vanilla BatchNorm. Contrary to BatchNorm, BatchRenorm uses the running statistics for normalizing the batches after a warmup phase. This makes it less prone to suffer from "outlier" batches that can happen during very long training runs and, therefore, is more robust during long training runs.

During the warmup phase, it behaves exactly like a BatchNorm layer. After the warmup phase, the running statistics are used for normalization. The running statistics are updated during training mode. During evaluation mode, the running statistics are used for normalization but not updated.

Parameters

- **num_features** (*int*) – Number of features in the input tensor.
- **eps** (*float*) – A value added to the variance for numerical stability.
- **momentum** (*float*) – The value used for the ra_mean and ra_var (running average) computation. It controls the rate of convergence for the batch renormalization statistics.
- **affine** (*bool*) – A boolean value that when set to True, this module has learnable affine parameters. Default: True
- **warmup_steps** (*int*) – Number of warum steps that are performed before the running statistics are used for normalization. During the warump phase, the batch statistics are used.

extra_repr()

Return the extra representation of the module.

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

Return type
str

forward(*x*)

Normalize the input tensor.

Parameters
x (*Tensor*) – Input tensor

Returns
Normalized tensor.

Return type
Tensor

class sb3_contrib.common.torch_layers.**BatchRenorm1d**(*num_features*, *eps=0.001*, *momentum=0.01*, *affine=True*, *warmup_steps=100000*)

Parameters

- **num_features** (*int*)
- **eps** (*float*)
- **momentum** (*float*)
- **affine** (*bool*)
- **warmup_steps** (*int*)

CHAPTER

TWELVE

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`.

Returns
Tensor containing the flattened gradients

Return type
Tensor

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** (*Tensor | None*) – 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

Returns
the loss

Return type
Tensor

CHAPTER

THIRTEEN

GYM WRAPPERS

Additional Gymnasium Wrappers to enhance Gymnasium environments.

13.1 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(***kwargs*)

Uses the *reset()* of the env that can be overwritten to change the returned data.

Return type
tuple[*ndarray* | dict[str, *ndarray*], dict[str, *Any*]]

step(*action*)

Uses the *step()* of the env that can be overwritten to change the returned data.

Parameters
action (*ActType*)

Return type
tuple[*ndarray* | dict[str, *ndarray*], *SupportsFloat*, bool, bool, dict[str, *Any*]]

CHAPTER

FOURTEEN

CHANGELOG

14.1 Release 2.9.0 (2026-06-15)

14.1.1 Breaking Changes:

- Upgraded to Stable-Baselines3 >= 2.9.0

14.1.2 New Features:

14.1.3 Bug Fixes:

14.1.4 Deprecations:

14.1.5 Others:

- Optimized tests (faster to run)

14.1.6 Documentation:

- Fixed dead link for `RecurrentPPO`.

14.2 Release 2.8.0 (2026-04-01)

14.2.1 Breaking Changes:

- Removed support for Python 3.9, please upgrade to Python >= 3.10
- Upgraded to Stable-Baselines3 >= 2.8.0
- Set `strict=True` for every call to `zip(...)`

14.2.2 New Features:

- Added official support for Python 3.13

14.2.3 Bug Fixes:

- Fixed `MaskablePPO` and `RecurrentPPO` inaccurate `n_updates` counting when `target_kl` early exits the training loop
- Fixed `RecurrentPPO` and `MaskablePPO` `forward` and `predict` not reshaping the action before clipping it (@immortal-boy)
- Do not call `forward()` method directly in `RecurrentPPO` (@immortal-boy)
- Fixed `MaskableCategorical.apply_masking()` crashing with `ValueError: Simplex` when cached `probs` deviate from sum=1 in float32 with large action spaces (torch 2.9+) (@kirann-05)

14.2.4 Deprecations:

14.2.5 Others:

14.2.6 Documentation:

- Switched to markdown documentation (using MyST parser)

14.3 Release 2.7.1 (2025-12-05)

> **Warning**
>
> Stable-Baselines3 (SB3) v2.7.1 will be the last one supporting Python 3.9 (end of life in October 2025). We highly recommended you to upgrade to Python >= 3.10.

14.3.1 Breaking Changes:

14.3.2 New Features:

14.3.3 Bug Fixes:

- Fix tensorboard log name for `MaskablePPO`

14.3.4 Deprecations:

14.3.5 Others:

14.3.6 Documentation:

14.4 Release 2.7.0 (2025-07-25)

14.4.1 Breaking Changes:

- Upgraded to Stable-Baselines3 >= 2.7.0

14.4.2 New Features:

- Added support for n-step returns for off-policy algorithms via the `n_steps` parameter

14.4.3 Bug Fixes:

- Use the `FloatSchedule` and `LinearSchedule` classes instead of lambdas in the ARS, PPO, and QRDQN implementations to improve model portability across different operating systems

14.4.4 Deprecations:

14.4.5 Others:

14.4.6 Documentation:

14.5 Release 2.6.0 (2025-03-24)

14.5.1 Breaking Changes:

- Upgraded to Stable-Baselines3 >= 2.6.0
- Renamed `_dump_logs()` to `dump_logs()`

14.5.2 New Features:

- Added support for Gymnasium v1.1.0

14.5.3 Bug Fixes:

- Fixed issues with `SubprocVecEnv` and `MaskablePPO` by using `vec_env.has_attr()` (pickling issues, mask function not present)

14.6 Release 2.5.0 (2025-01-27)

14.6.1 Breaking Changes:

- Upgraded to PyTorch 2.3.0
- Dropped Python 3.8 support
- Upgraded to Stable-Baselines3 >= 2.5.0

14.6.2 New Features:

- Added Python 3.12 support
- Added Numpy v2.0 support

14.7 Release 2.4.0 (2024-11-18)

New algorithm: added CrossQ, Gymnasium v1.0 support

14.7.1 Breaking Changes:

- Upgraded to Stable-Baselines3 >= 2.4.0

14.7.2 New Features:

- Added `CrossQ` algorithm, from "Batch Normalization in Deep Reinforcement Learning" paper (@danielpalen)
- Added `BatchRenorm` PyTorch layer used in `CrossQ` (@danielpalen)
- Added support for Gymnasium v1.0

14.7.3 Bug Fixes:

- Updated QR-DQN optimizer input to only include quantile_net parameters (@corentinlger)
- Updated QR-DQN paper link in docs (@corentinlger)
- Fixed a warning with PyTorch 2.4 when loading a `RecurrentPPO` model (You are using torch.load with weights_only=False)
- Fixed loading QRDQN changes `target_update_interval` (@jak3122)

14.7.4 Others:

- Updated PyTorch version on CI to 2.3.1
- Remove unnecessary SDE noise resampling in PPO/TRPO update
- Switched to uv to download packages on GitHub CI

14.8 Release 2.3.0 (2024-03-31)

New defaults hyperparameters for QR-DQN

14.8.1 Breaking Changes:

- Upgraded to Stable-Baselines3 >= 2.3.0
- The default `learning_starts` parameter of `QRDQN` have been changed to be consistent with the other offpolicy algorithms

```
# SB3 < 2.3.0 default hyperparameters, 50_000 corresponded to Atari defaults␣
↪hyperparameters
# model = QRDQN("MlpPolicy", env, learning_starts=50_000)
# SB3 >= 2.3.0:
model = QRDQN("MlpPolicy", env, learning_starts=100)
```

14.8.2 New Features:

- Added `rollout_buffer_class` and `rollout_buffer_kwargs` arguments to MaskablePPO
- Log success rate `rollout/success_rate` when available for on policy algorithms

14.8.3 Others:

- Fixed `train_freq` type annotation for tqc and qrdqn (@Armandpl)
- Fixed `sb3_contrib/common/maskable/*.py` type annotations
- Fixed `sb3_contrib/ppo_mask/ppo_mask.py` type annotations
- Fixed `sb3_contrib/common/vec_env/async_eval.py` type annotations

14.8.4 Documentation:

- Add some additional notes about `MaskablePPO` (evaluation and multi-process) (@icheered)

14.9 Release 2.2.1 (2023-11-17)

14.9.1 Breaking Changes:

- Upgraded to Stable-Baselines3 >= 2.2.1
- Switched to `ruff` for sorting imports (isort is no longer needed), black and ruff version now require a minimum version
- Dropped `x is False` in favor of `not x`, which means that callbacks that wrongly returned None (instead of a boolean) will cause the training to stop (@iwishiwasaneagle)

14.9.2 New Features:

- Added `set_options` for `AsyncEval`
- Added `rollout_buffer_class` and `rollout_buffer_kwargs` arguments to TRPO

14.9.3 Others:

- Fixed `ActorCriticPolicy.extract_features()` signature by adding an optional `features_extractor` argument
- Update dependencies (accept newer Shimmy/Sphinx version and remove `sphinx_autodoc_typehints`)

14.10 Release 2.1.0 (2023-08-17)

14.10.1 Breaking Changes:

- Removed Python 3.7 support
- SB3 now requires PyTorch > 1.13
- Upgraded to Stable-Baselines3 >= 2.1.0

14.10.2 New Features:

- Added Python 3.11 support

14.10.3 Bug Fixes:

- Fixed MaskablePPO ignoring `stats_window_size` argument

14.11 Release 2.0.0 (2023-06-22)

Gymnasium support

> **Warning**
>
> Stable-Baselines3 (SB3) v2.0 will be the last one supporting python 3.7 (end of life in June 2023). We highly recommended you to upgrade to Python >= 3.8.

14.11.1 Breaking Changes:

- Switched to Gymnasium as primary backend, Gym 0.21 and 0.26 are still supported via the `shimmy` package (@carlosluis, @arjun-kg, @tlpss)
- Upgraded to Stable-Baselines3 >= 2.0.0

14.11.2 Bug Fixes:

- Fixed QRDQN update interval for multi envs

14.11.3 Others:

- Fixed `sb3_contrib/tqc/*.py` type hints
- Fixed `sb3_contrib/trpo/*.py` type hints
- Fixed `sb3_contrib/common/envs/invalid_actions_env.py` type hints

14.11.4 Documentation:

- Update documentation, switch from Gym to Gymnasium

14.12 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.

14.12.1 Breaking Changes:

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

14.12.2 New Features:

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

14.12.3 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`

14.12.4 Documentation:

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

14.13 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.

14.13.1 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

14.13.2 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`

14.13.3 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`

14.13.4 Deprecations:

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

14.13.5 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`

14.14 Release 1.6.2 (2022-10-10)

Progress bar and upgrade to latest SB3 version

14.14.1 Breaking Changes:

- Upgraded to Stable-Baselines3 >= 1.6.2

14.14.2 New Features:

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

14.14.3 Deprecations:

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

14.14.4 Others:

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

14.15 Release 1.6.1 (2022-09-29)

Bug fix release

14.15.1 Breaking Changes:

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

14.15.2 New Features:

14.15.3 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)

14.15.4 Deprecations:

14.15.5 Others:

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

14.16 Release 1.6.0 (2022-07-11)

Add RecurrentPPO (aka PPO LSTM)

14.16.1 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)

14.16.2 New Features:

- Added `RecurrentPPO` (aka PPO LSTM)

14.16.3 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

14.16.4 Deprecations:

14.17 Release 1.5.0 (2022-03-25)

14.17.1 Breaking Changes:

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

14.17.2 New Features:

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

14.17.3 Bug Fixes:

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

14.17.4 Deprecations:

14.17.5 Others:

14.17.6 Documentation:

14.18 Release 1.4.0 (2022-01-19)

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

14.18.1 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)

14.18.2 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)

14.18.3 Bug Fixes:

14.18.4 Deprecations:

14.18.5 Others:

- Improve test coverage for `MaskablePPO`

14.18.6 Documentation:

14.19 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.

14.19.1 Breaking Changes:

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

14.19.2 New Features:

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

14.19.3 Bug Fixes:

14.19.4 Deprecations:

14.19.5 Others:

14.19.6 Documentation:

14.20 Release 1.2.0 (2021-09-08)

Train/Eval mode support

14.20.1 Breaking Changes:

- Upgraded to Stable-Baselines3 >= 1.2.0

14.20.2 Bug Fixes:

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

14.20.3 Deprecations:

14.20.4 Others:

- Fixed type annotation
- Added python 3.9 to CI

14.20.5 Documentation:

14.21 Release 1.1.0 (2021-07-01)

Dictionary observation support and timeout handling

14.21.1 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)

14.21.2 Bug Fixes:

- Removed unused code in `TQC`

14.21.3 Deprecations:

14.21.4 Others:

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

14.21.5 Documentation:

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

14.22 Release 1.0 (2021-03-17)

14.22.1 Breaking Changes:

- Upgraded to Stable-Baselines3 >= 1.0

14.22.2 Bug Fixes:

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

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

14.23.1 Bug Fixes:

- Upgraded to Stable-Baselines3 >= 0.11.1

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

14.24.1 Breaking Changes:

- Upgraded to Stable-Baselines3 >= 0.11.0

14.24.2 New Features:

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

14.24.3 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)

14.24.4 Deprecations:

14.24.5 Others:

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

14.24.6 Documentation:

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

Truncated Quantiles Critic (TQC)

14.25.1 Breaking Changes:

14.25.2 New Features:

- Added `TQC` algorithm (@araffin)

14.25.3 Bug Fixes:

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

14.25.4 Deprecations:

14.25.5 Others:

14.25.6 Documentation:

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

14.26 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).

14.27 Contributors:

@ku2482 @guyk1971 @minhlong94 @ayeright @kronion @glmcdona @cyprienc @sgillen @Gregwar @rnederstigt @qgallouedec @mlodel @CppMaster @burakdmb @honglu2875 @ZikangXiong @AlexPasqua @jonasreiher @icheered @Armandpl @danielpalen @corentinlger @immortal-boy

CHAPTER

FIFTEEN

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

SIXTEEN

CONTRIBUTING

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

CHAPTER

SEVENTEEN

INDICES AND TABLES

- genindex
- search
- modindex

PYTHON MODULE INDEX

S

INDEX

H

L

M

P

Q

R

S

T