Stable Baselines3 Documentation

Release 2.6.1a1

Stable Baselines3 Contributors

May 15, 2025

USER GUIDE

Stable Baselines3 (SB3) is a set of reliable implementations of reinforcement learning algorithms in PyTorch. It is the next major version of Stable Baselines.

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

Paper: https://jmlr.org/papers/volume22/20-1364/20-1364.pdf

RL Baselines3 Zoo (training framework for SB3): https://github.com/DLR-RM/rl-baselines3-zoo

RL Baselines3 Zoo provides a collection of pre-trained agents, scripts for training, evaluating agents, tuning hyperparameters, plotting results and recording videos.

SB3 Contrib (experimental RL code, latest algorithms): https://github.com/Stable-Baselines-Team/stable-baselines3-contrib

SBX (SB3 + Jax): https://github.com/araffin/sbx

CHAPTER

ONE

MAIN FEATURES

- Unified structure for all algorithms
- PEP8 compliant (unified code style)
- Documented functions and classes
- Tests, high code coverage and type hints
- Clean code
- Tensorboard support
- **The performance of each algorithm was tested** (see *Results* section in their respective page)

1.1 Installation

1.1.1 Prerequisites

Stable-Baselines3 requires python 3.9+ and PyTorch >= 2.3

Windows

We recommend using Anaconda for Windows users for easier installation of Python packages and required libraries. You need an environment with Python version 3.8 or above.

For a quick start you can move straight to installing Stable-Baselines3 in the next step.

> **Note**
>
> Trying to create Atari environments may result in vague errors related to missing DLL files and modules. This is an issue with atari-py package. See this discussion for more information.

Stable Release

To install Stable Baselines3 with pip, execute:

```
pip install stable-baselines3[extra]
```

> **Note**
>
> Some shells such as Zsh require quotation marks around brackets, i.e. `pip install 'stable-baselines3[extra]'` More information.

This includes optional dependencies like Tensorboard, OpenCV or `ale-py` to train on Atari games. If you do not need those, you can use:

```
pip install stable-baselines3
```

Note

If you need to work with OpenCV on a machine without a X-server (for instance inside a docker image), you will need to install `opencv-python-headless`, see issue #298.

1.1.2 Bleeding-edge version

```
pip install git+https://github.com/DLR-RM/stable-baselines3
```

with extras:

```
pip install "stable_baselines3[extra,tests,docs] @ git+https://github.com/DLR-RM/stable-
↪baselines3"
```

1.1.3 Development version

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

```
git clone https://github.com/DLR-RM/stable-baselines3 && cd stable-baselines3
pip install -e .[docs,tests,extra]
```

1.1.4 Using Docker Images

If you are looking for docker images with stable-baselines already installed in it, we recommend using images from RL Baselines3 Zoo.

Otherwise, the following images contained all the dependencies for stable-baselines3 but not the stable-baselines3 package itself. They are made for development.

Use Built Images

GPU image (requires nvidia-docker):

```
docker pull stablebaselines/stable-baselines3
```

CPU only:

```
docker pull stablebaselines/stable-baselines3-cpu
```

Build the Docker Images

Build GPU image (with nvidia-docker):

```
make docker-gpu
```

Build CPU image:

```
make docker-cpu
```

Note: if you are using a proxy, you need to pass extra params during build and do some tweaks:

```
--network=host --build-arg HTTP_PROXY=http://your.proxy.fr:8080/ --build-arg http_
↪proxy=http://your.proxy.fr:8080/ --build-arg HTTPS_PROXY=https://your.proxy.fr:8080/ --
↪build-arg https_proxy=https://your.proxy.fr:8080/
```

Run the images (CPU/GPU)

Run the nvidia-docker GPU image

```
docker run -it --runtime=nvidia --rm --network host --ipc=host --name test --mount src="
↪$(pwd)",target=/home/mamba/stable-baselines3,type=bind stablebaselines/stable-
↪baselines3 bash -c 'cd /home/mamba/stable-baselines3/ && pytest tests/'
```

Or, with the shell file:

```
./scripts/run_docker_gpu.sh pytest tests/
```

Run the docker CPU image

```
docker run -it --rm --network host --ipc=host --name test --mount src="$(pwd)",target=/
↪home/mamba/stable-baselines3,type=bind stablebaselines/stable-baselines3-cpu bash -c
↪'cd /home/mamba/stable-baselines3/ && pytest tests/'
```

Or, with the shell file:

```
./scripts/run_docker_cpu.sh pytest tests/
```

Explanation of the docker command:

- `docker run -it` create an instance of an image (=container), and run it interactively (so ctrl+c will work)
- `--rm` option means to remove the container once it exits/stops (otherwise, you will have to use `docker rm`)
- `--network host` don't use network isolation, this allows to use tensorboard/visdom on host machine
- `--ipc=host` Use the host system's IPC namespace. IPC (POSIX/SysV IPC) namespace provides separation of named shared memory segments, semaphores and message queues.
- `--name test` give explicitly the name `test` to the container, otherwise it will be assigned a random name
- `--mount src=...` give access of the local directory (`pwd` command) to the container (it will be map to `/home/mamba/stable-baselines`), so all the logs created in the container in this folder will be kept
- `bash -c '...'` Run command inside the docker image, here run the tests (`pytest tests/`)

1.2 Getting Started

Note

Stable-Baselines3 (SB3) uses *vectorized environments (VecEnv)* internally. Please read the associated section to learn more about its features and differences compared to a single Gym environment.

Most of the library tries to follow a sklearn-like syntax for the Reinforcement Learning algorithms.

Here is a quick example of how to train and run A2C on a CartPole environment:

```
import gymnasium as gym

from stable_baselines3 import A2C

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

model = A2C("MlpPolicy", env, verbose=1)
model.learn(total_timesteps=10_000)

vec_env = model.get_env()
obs = vec_env.reset()
for i in range(1000):
    action, _state = model.predict(obs, deterministic=True)
    obs, reward, done, info = vec_env.step(action)
    vec_env.render("human")
    # VecEnv resets automatically
    # if done:
    #   obs = vec_env.reset()
```

Note

You can find explanations about the logger output and names in the *Logger* section.

Or just train a model with a one line if the environment is registered in Gymnasium and if the policy is registered:

```
from stable_baselines3 import A2C

model = A2C("MlpPolicy", "CartPole-v1").learn(10_000)
```

1.3 Reinforcement Learning Tips and Tricks

The aim of this section is to help you run reinforcement learning experiments. It covers general advice about RL (where to start, which algorithm to choose, how to evaluate an algorithm, . . .), as well as tips and tricks when using a custom environment or implementing an RL algorithm.

Note

We have a video on YouTube that covers this section in more details. You can also find the slides here.

Note

We also have a video on Designing and Running Real-World RL Experiments, slides can be found online.

1.3.1 General advice when using Reinforcement Learning

TL;DR

1. Read about RL and Stable-Baselines3 (SB3)
2. Do quantitative experiments and hyperparameter tuning if needed
3. Evaluate the performance using a separate test environment (remember to check wrappers!)
4. For better performance, increase the training budget

Like any other subject, if you want to work with RL, you should first read about it (we have a dedicated resource page to get you started) to understand what you are using. We also recommend that you read the Stable Baselines3 (SB3) documentation and do the tutorial. It covers basic usage and guides you towards more advanced concepts of the library (e.g. callbacks and wrappers).

Reinforcement Learning differs from other machine learning methods in several ways. The data used to train the agent is collected through interactions with the environment by the agent itself (as opposed to, for example, supervised learning where you have a fixed dataset). This dependency can lead to a vicious circle: if the agent collects poor quality data (e.g. trajectories with no rewards), it will not improve and will continue to collect bad trajectories.

This factor, among others, explains that results in RL may vary from one run to another (i.e., when only the seed of the pseudo-random generator changes). For this reason, you should always do several runs to obtain quantitative results.

Good results in RL generally depend on finding appropriate hyperparameters. Recent algorithms (PPO, SAC, TD3, DroQ) normally require little hyperparameter tuning, however, *don't expect the default ones to work* in every environment.

Therefore, we *highly recommend you* to take a look at the RL zoo (or the original papers) for tuned hyperparameters. A best practice when you apply RL to a new problem is to do automatic hyperparameter optimization. Again, this is included in the RL zoo.

When applying RL to a custom problem, you should always normalize the input to the agent (e.g. using `VecNormalize` for PPO/A2C) and look at common preprocessing done on other environments (e.g. for Atari, frame-stack, …). Please refer to *Tips and Tricks when creating a custom environment* paragraph below for more advice related to custom environments.

Current Limitations of RL

You have to be aware of the current limitations of reinforcement learning.

Model-free RL algorithms (i.e. all the algorithms implemented in SB3) are usually *sample inefficient*. They require a lot of samples (sometimes millions of interactions) to learn anything useful. That's why most of the successes in RL were achieved on games or in simulation only. For instance, in this work by ETH Zurich, the ANYmal robot was trained in simulation only, and then tested in the real world.

As a general advice, to obtain better performances, you should augment the budget of the agent (number of training timesteps).

In order to achieve the desired behavior, expert knowledge is often required to design an adequate reward function. This *reward engineering* (or *RewArt* as coined by Freek Stulp), necessitates several iterations. As a good example of reward shaping, you can take a look at Deep Mimic paper which combines imitation learning and reinforcement learning to do acrobatic moves.

A final limitation of RL is the instability of training. That is, you can observe a huge drop in performance during training. This behavior is particularly present in `DDPG`, that's why its extension `TD3` tries to tackle that issue. Other methods, such as `TRPO` or `PPO` use a *trust region* to minimize this problem by avoiding too large updates.

How to evaluate an RL algorithm?

> **Note**
>
> Pay attention to environment wrappers when evaluating your agent and comparing results to others' results. Modifications to episode rewards or lengths may also affect evaluation results which may not be desirable. Check `evaluate_policy` helper function in *Evaluation Helper* section.

Because most algorithms use exploration noise during training, you need a separate test environment to evaluate the performance of your agent at a given time. It is recommended to periodically evaluate your agent for `n` test episodes (`n` is usually between 5 and 20) and average the reward per episode to have a good estimate.

> **Note**
>
> We provide an `EvalCallback` for doing such evaluation. You can read more about it in the *Callbacks* section.

As some policies are stochastic by default (e.g. A2C or PPO), you should also try to set *deterministic=True* when calling the *.predict()* method, this frequently leads to better performance. Looking at the training curve (episode reward function of the timesteps) is a good proxy but underestimates the agent true performance.

We highly recommend reading Empirical Design in Reinforcement Learning, as it provides valuable insights for best practices when running RL experiments.

We also suggest reading Deep Reinforcement Learning that Matters for a good discussion about RL evaluation, and Rliable: Better Evaluation for Reinforcement Learning for comparing results.

You can also take a look at this blog post and this issue by Cédric Colas.

1.3.2 Which algorithm should I use?

There is no silver bullet in RL, you can choose one or the other depending on your needs and problems. The first distinction comes from your action space, i.e., do you have discrete (e.g. LEFT, RIGHT, …) or continuous actions (ex: go to a certain speed)?

Some algorithms are only tailored for one or the other domain: `DQN` supports only discrete actions, while `SAC` is restricted to continuous actions.

The second difference that will help you decide is whether you can parallelize your training or not. If what matters is the wall clock training time, then you should lean towards `A2C` and its derivatives (PPO, …). Take a look at the Vectorized Environments to learn more about training with multiple workers.

To accelerate training, you can also take a look at SBX, which is SB3 + Jax, it has less features than SB3 but can be up to 20x faster than SB3 PyTorch thanks to JIT compilation of the gradient update.

In sparse reward settings, we either recommend using either dedicated methods like HER (see below) or population-based algorithms like ARS (available in our *contrib repo*).

To sum it up:

Discrete Actions

> **Note**
>
> This covers `Discrete`, `MultiDiscrete`, `Binary` and `MultiBinary` spaces

Discrete Actions - Single Process

DQN with extensions (double DQN, prioritized replay, ...) are the recommended algorithms. We notably provide QR-DQN in our *contrib repo*. DQN is usually slower to train (regarding wall clock time) but is the most sample efficient (because of its replay buffer).

Discrete Actions - Multiprocessed

You should give a try to PPO or A2C.

Continuous Actions

Continuous Actions - Single Process

Current State Of The Art (SOTA) algorithms are SAC, TD3, CrossQ and TQC (available in our *contrib repo* and *SBX (SB3 + Jax) repo*). Please use the hyperparameters in the RL zoo for best results.

If you want an extremely sample-efficient algorithm, we recommend using the DroQ configuration in SBX (it does many gradient steps per step in the environment).

Continuous Actions - Multiprocessed

Take a look at PPO, TRPO (available in our *contrib repo*) or A2C. Again, don't forget to take the hyperparameters from the RL zoo for continuous actions problems (cf *Bullet* envs).

> **Note**
>
> Normalization is critical for those algorithms

Goal Environment

If your environment follows the GoalEnv interface (cf *HER*), then you should use HER + (SAC/TD3/DDPG/DQN/QR-DQN/TQC) depending on the action space.

> **Note**
>
> The batch_size is an important hyperparameter for experiments with *HER*

1.3.3 Tips and Tricks when creating a custom environment

If you want to learn about how to create a custom environment, we recommend you read this page. We also provide a colab notebook for a concrete example of creating a custom gym environment.

Some basic advice:

- always normalize your observation space if you can, i.e. if you know the boundaries
- normalize your action space and make it symmetric if it is continuous (see potential problem below) A good practice is to rescale your actions so that they lie in [-1, 1]. This does not limit you, as you can easily rescale the action within the environment.
- start with a shaped reward (i.e. informative reward) and a simplified version of your problem
- debug with random actions to check if your environment works and follows the gym interface (with check_env, see below)

Two important things to keep in mind when creating a custom environment are avoiding breaking the Markov assumption and properly handle termination due to a timeout (maximum number of steps in an episode). For example, if there is a time delay between action and observation (e.g. due to wifi communication), you should provide a history of observations as input.

Termination due to timeout (max number of steps per episode) needs to be handled separately. You should return `truncated = True`. If you are using the gym `TimeLimit` wrapper, this will be done automatically. You can read Time Limit in RL, take a look at the Designing and Running Real-World RL Experiments video or RL Tips and Tricks video for more details.

We provide a helper to check that your environment runs without error:

```python
from stable_baselines3.common.env_checker import check_env

env = CustomEnv(arg1, ...)
# It will check your custom environment and output additional warnings if needed
check_env(env)
```

If you want to quickly try a random agent on your environment, you can also do:

```python
env = YourEnv()
obs, info = env.reset()
n_steps = 10
for _ in range(n_steps):
    # Random action
    action = env.action_space.sample()
    obs, reward, terminated, truncated, info = env.step(action)
    if done:
        obs, info = env.reset()
```

Why should I normalize the action space?

Most reinforcement learning algorithms rely on a Gaussian distribution (initially centered at 0 with std 1) for continuous actions. So, if you forget to normalize the action space when using a custom environment, this can harm learning and can be difficult to debug (cf attached image and issue #473).

Another consequence of using a Gaussian distribution is that the action range is not bounded. That's why clipping is usually used as a bandage to stay in a valid interval. A better solution would be to use a squashing function (cf `SAC`) or a Beta distribution (cf issue #112).

Note

This statement is not true for `DDPG` or `TD3` because they don't rely on any probability distribution.

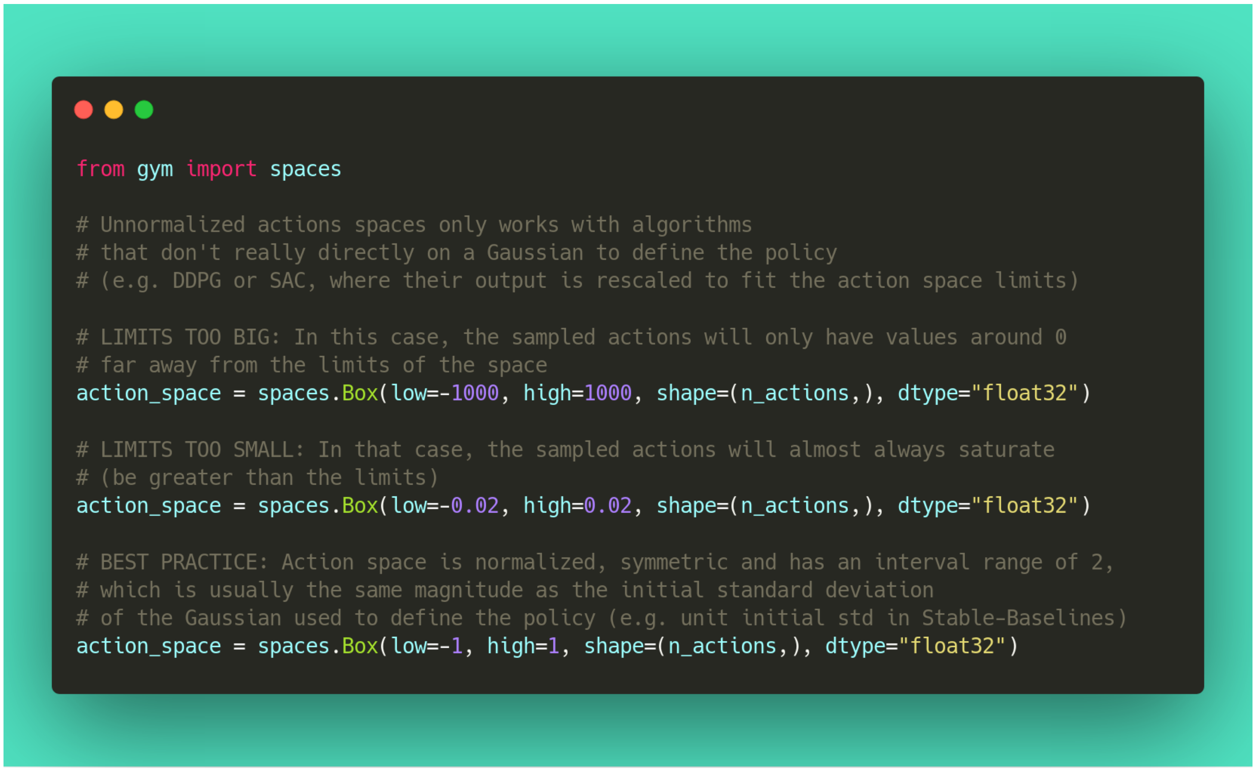
1.3.4 Tips and Tricks when implementing an RL algorithm

Note

We have a video on YouTube about reliable RL that covers this section in more details. You can also find the slides online.

When you try to reproduce a RL paper by implementing the algorithm, the nuts and bolts of RL research by John Schulman are quite useful (video).

We *recommend following those steps to have a working RL algorithm*:



```python
from gym import spaces

# Unnormalized actions spaces only works with algorithms
# that don't really directly on a Gaussian to define the policy
# (e.g. DDPG or SAC, where their output is rescaled to fit the action space limits)

# LIMITS TOO BIG: In this case, the sampled actions will only have values around 0
# far away from the limits of the space
action_space = spaces.Box(low=-1000, high=1000, shape=(n_actions,), dtype="float32")

# LIMITS TOO SMALL: In that case, the sampled actions will almost always saturate
# (be greater than the limits)
action_space = spaces.Box(low=-0.02, high=0.02, shape=(n_actions,), dtype="float32")

# BEST PRACTICE: Action space is normalized, symmetric and has an interval range of 2,
# which is usually the same magnitude as the initial standard deviation
# of the Gaussian used to define the policy (e.g. unit initial std in Stable-Baselines)
action_space = spaces.Box(low=-1, high=1, shape=(n_actions,), dtype="float32")
```

1. Read the original paper several times
2. Read existing implementations (if available)
3. Try to have some "sign of life" on toy problems
4. Validate the implementation by making it run on harder and harder envs (you can compare results against the RL zoo). You usually need to run hyperparameter optimization for that step.

You need to be particularly careful on the shape of the different objects you are manipulating (a broadcast mistake will fail silently cf. issue #75) and when to stop the gradient propagation.

Don't forget to handle termination due to timeout separately (see remark in the custom environment section above), you can also take a look at Issue #284 and Issue #633.

A personal pick (by @araffin) for environments with gradual difficulty in RL with continuous actions:

1. Pendulum (easy to solve)
2. HalfCheetahBullet (medium difficulty with local minima and shaped reward)
3. BipedalWalkerHardcore (if it works on that one, then you can have a cookie)

in RL with discrete actions:

1. CartPole-v1 (easy to be better than random agent, harder to achieve maximal performance)
2. LunarLander
3. Pong (one of the easiest Atari game)
4. other Atari games (e.g. Breakout)

1.4 Reinforcement Learning Resources

Stable-Baselines3 assumes that you already understand the basic concepts of Reinforcement Learning (RL).

However, if you want to learn about RL, there are several good resources to get started:

- OpenAI Spinning Up
- The Deep Reinforcement Learning Course
- David Silver's course
- Lilian Weng's blog
- Berkeley's Deep RL Bootcamp
- Berkeley's Deep Reinforcement Learning course
- DQN tutorial
- Decisions & Dragons - FAQ for RL foundations
- More resources

1.5 RL Algorithms

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

Name	`Box`	`Discrete`	`MultiDiscrete`	`MultiBinary`	Multi Processing
ARS[1]	✓	✓			✓
A2C	✓	✓	✓	✓	✓
CrossQ[1]	✓				✓
DDPG	✓				✓
DQN		✓			✓
HER	✓	✓			✓
PPO	✓	✓	✓	✓	✓
QR-DQN[1]		✓			✓
RecurrentPPO[1]	✓	✓	✓	✓	✓
SAC	✓				✓
TD3	✓				✓
TQC[1]	✓				✓
TRPO[1]	✓	✓	✓	✓	✓
Maskable PPO[1]		✓	✓	✓	✓

Note

`Tuple` observation spaces are not supported by any environment, however, single-level `Dict` spaces are (cf. *Examples*).

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.

[1] Implemented in SB3 Contrib

- 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.

Note

More algorithms (like QR-DQN or TQC) are implemented in our *contrib repo* and in our *SBX (SB3 + Jax) repo* (DroQ, CrossQ, SimBa, . . .).

Note

Some logging values (like ep_rew_mean, ep_len_mean) are only available when using a Monitor wrapper See Issue #339 for more info.

Note

When using off-policy algorithms, Time Limits (aka timeouts) are handled properly (cf. issue #284). You can revert to SB3 < 2.1.0 behavior by passing handle_timeout_termination=False via the replay_buffer_kwargs argument.

1.5.1 Reproducibility

Completely reproducible results are not guaranteed across PyTorch releases or different platforms. Furthermore, results need not be reproducible between CPU and GPU executions, even when using identical seeds.

In order to make computations deterministics, on your specific problem on one specific platform, you need to pass a seed argument at the creation of a model. If you pass an environment to the model using set_env(), then you also need to seed the environment first.

Credit: part of the *Reproducibility* section comes from PyTorch Documentation

1.5.2 Training exceeds total_timesteps

When you train an agent using SB3, you pass a total_timesteps parameter to the learn() method which defines the training budget for the agent (how many interactions with the environment are allowed). For example:

```python
from stable_baselines3 import PPO

model = PPO("MlpPolicy", "CartPole-v1").learn(total_timesteps=1_000)
```

Because of the way the algorithms work, total_timesteps is a lower bound (see issue #1150). In the example above, PPO will effectively collect n_steps * n_envs = 2048 * 1 steps despite total_timesteps=1_000 In more details:

- PPO/A2C and derivates collect n_steps * n_envs of experience before performing an update, so if you want to have exactly total_timesteps, you will need to adjust those values
- SAC/DQN/TD3 and other off-policy algorithms collect train_freq * n_envs steps before doing an update (when train_freq is in steps and not episodes), so if you want to have exactly total_timesteps you have to adjust these values (train_freq=4 by default for DQN)
- ARS and other population-based algorithms evaluate the policy for n_episodes with n_envs, so unless the number of steps per episode is fixed, it is not possible to exactly achieve total_timesteps

- when using multiple envs, each call to `env.step()` corresponds to `n_envs` timesteps, so it is no longer possible to use the `EvaluationCallback` at an exact timestep

1.6 Examples

Note

These examples are only to demonstrate the use of the library and its functions, and the trained agents may not solve the environments. Optimized hyperparameters can be found in the RL Zoo repository.

1.6.1 Try it online with Colab Notebooks!

All the following examples can be executed online using Google colab notebooks:

- Full Tutorial
- All Notebooks
- Getting Started
- Training, Saving, Loading
- Multiprocessing
- Monitor Training and Plotting
- Atari Games
- RL Baselines zoo
- PyBullet
- Hindsight Experience Replay
- Advanced Saving and Loading

1.6.2 Basic Usage: Training, Saving, Loading

In the following example, we will train, save and load a DQN model on the Lunar Lander environment.

Fig. 1: Lunar Lander Environment

Note

LunarLander requires the python package `box2d`. You can install it using `apt install swig` and then `pip install box2d box2d-kengz`

Warning

`load` method re-creates the model from scratch and should be called on the Algorithm without instantiating it first, e.g. `model = DQN.load("dqn_lunar", env=env)` instead of `model = DQN(env=env)` followed by `model.load("dqn_lunar")`. The latter **will not work** as `load` is not an in-place operation. If you want to load parameters without re-creating the model, e.g. to evaluate the same model with multiple different sets of parameters, consider using `set_parameters` instead.

```python
import gymnasium as gym

from stable_baselines3 import DQN
from stable_baselines3.common.evaluation import evaluate_policy


# Create environment
env = gym.make("LunarLander-v2", render_mode="rgb_array")

# Instantiate the agent
model = DQN("MlpPolicy", env, verbose=1)
# Train the agent and display a progress bar
model.learn(total_timesteps=int(2e5), progress_bar=True)
# Save the agent
model.save("dqn_lunar")
del model  # delete trained model to demonstrate loading

# Load the trained agent
# NOTE: if you have loading issue, you can pass `print_system_info=True`
# to compare the system on which the model was trained vs the current one
# model = DQN.load("dqn_lunar", env=env, print_system_info=True)
model = DQN.load("dqn_lunar", env=env)

# Evaluate the agent
# NOTE: If you use wrappers with your environment that modify rewards,
#       this will be reflected here. To evaluate with original rewards,
#       wrap environment in a "Monitor" wrapper before other wrappers.
mean_reward, std_reward = evaluate_policy(model, model.get_env(), n_eval_episodes=10)

# Enjoy trained agent
vec_env = model.get_env()
obs = vec_env.reset()
for i in range(1000):
    action, _states = model.predict(obs, deterministic=True)
    obs, rewards, dones, info = vec_env.step(action)
    vec_env.render("human")
```

1.6.3 Multiprocessing: Unleashing the Power of Vectorized Environments

Fig. 2: CartPole Environment

```python
import gymnasium as gym

from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import DummyVecEnv, SubprocVecEnv
from stable_baselines3.common.env_util import make_vec_env
```

(continues on next page)

(continued from previous page)

```
from stable_baselines3.common.utils import set_random_seed

def make_env(env_id: str, rank: int, seed: int = 0):
    """
    Utility function for multiprocessed env.

    :param env_id: the environment ID
    :param num_env: the number of environments you wish to have in subprocesses
    :param seed: the initial seed for RNG
    :param rank: index of the subprocess
    """
    def _init():
        env = gym.make(env_id, render_mode="human")
        env.reset(seed=seed + rank)
        return env
    set_random_seed(seed)
    return _init

if __name__ == "__main__":
    env_id = "CartPole-v1"
    num_cpu = 4  # Number of processes to use
    # Create the vectorized environment
    vec_env = SubprocVecEnv([make_env(env_id, i) for i in range(num_cpu)])

    # Stable Baselines provides you with make_vec_env() helper
    # which does exactly the previous steps for you.
    # You can choose between `DummyVecEnv` (usually faster) and `SubprocVecEnv`
    # env = make_vec_env(env_id, n_envs=num_cpu, seed=0, vec_env_cls=SubprocVecEnv)

    model = PPO("MlpPolicy", vec_env, verbose=1)
    model.learn(total_timesteps=25_000)

    obs = vec_env.reset()
    for _ in range(1000):
        action, _states = model.predict(obs)
        obs, rewards, dones, info = vec_env.step(action)
        vec_env.render()
```

1.6.4 Multiprocessing with off-policy algorithms

⚠ **Warning**

When using multiple environments with off-policy algorithms, you should update the `gradient_steps` parameter too. Set it to `gradient_steps=-1` to perform as many gradient steps as transitions collected. There is usually a compromise between wall-clock time and sample efficiency, see this example in PR #439

```
import gymnasium as gym

from stable_baselines3 import SAC
from stable_baselines3.common.env_util import make_vec_env
```

(continues on next page)

(continued from previous page)

```
vec_env = make_vec_env("Pendulum-v0", n_envs=4, seed=0)

# We collect 4 transitions per call to `env.step()`
# and performs 2 gradient steps per call to `env.step()`
# if gradient_steps=-1, then we would do 4 gradients steps per call to `env.step()`
model = SAC("MlpPolicy", vec_env, train_freq=1, gradient_steps=2, verbose=1)
model.learn(total_timesteps=10_000)
```

1.6.5 Dict Observations

You can use environments with dictionary observation spaces. This is useful in the case where one can't directly concatenate observations such as an image from a camera combined with a vector of servo sensor data (e.g., rotation angles). Stable Baselines3 provides SimpleMultiObsEnv as an example of this kind of setting. The environment is a simple grid world, but the observations for each cell come in the form of dictionaries. These dictionaries are randomly initialized on the creation of the environment and contain a vector observation and an image observation.

```
from stable_baselines3 import PPO
from stable_baselines3.common.envs import SimpleMultiObsEnv


# Stable Baselines provides SimpleMultiObsEnv as an example environment with Dict observations
env = SimpleMultiObsEnv(random_start=False)

model = PPO("MultiInputPolicy", env, verbose=1)
model.learn(total_timesteps=100_000)
```

1.6.6 Callbacks: Monitoring Training

> **Note**
>
> We recommend reading the Callback section

You can define a custom callback function that will be called inside the agent. This could be useful when you want to monitor training, for instance display live learning curves in Tensorboard or save the best agent. If your callback returns False, training is aborted early.

```
import os

import gymnasium as gym
import numpy as np
import matplotlib.pyplot as plt

from stable_baselines3 import TD3
from stable_baselines3.common import results_plotter
from stable_baselines3.common.monitor import Monitor
from stable_baselines3.common.results_plotter import load_results, ts2xy, plot_results
from stable_baselines3.common.noise import NormalActionNoise
```

(continues on next page)

(continued from previous page)

```
from stable_baselines3.common.callbacks import BaseCallback


class SaveOnBestTrainingRewardCallback(BaseCallback):
    """
    Callback for saving a model (the check is done every ``check_freq`` steps)
    based on the training reward (in practice, we recommend using ``EvalCallback``).

    :param check_freq:
    :param log_dir: Path to the folder where the model will be saved.
      It must contain the file created by the ``Monitor`` wrapper.
    :param verbose: Verbosity level: 0 for no output, 1 for info messages, 2 for debug␣
↪messages
    """
    def __init__(self, check_freq: int, log_dir: str, verbose: int = 1):
        super().__init__(verbose)
        self.check_freq = check_freq
        self.log_dir = log_dir
        self.save_path = os.path.join(log_dir, "best_model")
        self.best_mean_reward = -np.inf

    def _init_callback(self) -> None:
        # Create folder if needed
        if self.save_path is not None:
            os.makedirs(self.save_path, exist_ok=True)

    def _on_step(self) -> bool:
        if self.n_calls % self.check_freq == 0:

          # Retrieve training reward
          x, y = ts2xy(load_results(self.log_dir), "timesteps")
          if len(x) > 0:
              # Mean training reward over the last 100 episodes
              mean_reward = np.mean(y[-100:])
              if self.verbose >= 1:
                print(f"Num timesteps: {self.num_timesteps}")
                print(f"Best mean reward: {self.best_mean_reward:.2f} - Last mean reward␣
↪per episode: {mean_reward:.2f}")

              # New best model, you could save the agent here
              if mean_reward > self.best_mean_reward:
                  self.best_mean_reward = mean_reward
                  # Example for saving best model
                  if self.verbose >= 1:
                    print(f"Saving new best model to {self.save_path}")
                  self.model.save(self.save_path)

        return True

# Create log dir
log_dir = "tmp/"
os.makedirs(log_dir, exist_ok=True)
```

(continues on next page)

(continued from previous page)

```
# Create and wrap the environment
env = gym.make("LunarLanderContinuous-v2")
env = Monitor(env, log_dir)

# Add some action noise for exploration
n_actions = env.action_space.shape[-1]
action_noise = NormalActionNoise(mean=np.zeros(n_actions), sigma=0.1 * np.ones(n_
↪actions))
# Because we use parameter noise, we should use a MlpPolicy with layer normalization
model = TD3("MlpPolicy", env, action_noise=action_noise, verbose=0)
# Create the callback: check every 1000 steps
callback = SaveOnBestTrainingRewardCallback(check_freq=1000, log_dir=log_dir)
# Train the agent
timesteps = 1e5
model.learn(total_timesteps=int(timesteps), callback=callback)

plot_results([log_dir], timesteps, results_plotter.X_TIMESTEPS, "TD3 LunarLander")
plt.show()
```

1.6.7 Callbacks: Evaluate Agent Performance

To periodically evaluate an agent's performance on a separate test environment, use `EvalCallback`. You can control the evaluation frequency with `eval_freq` to monitor your agent's progress during training.

```
import os
import gymnasium as gym

from stable_baselines3 import SAC
from stable_baselines3.common.callbacks import EvalCallback
from stable_baselines3.common.env_util import make_vec_env

env_id = "Pendulum-v1"
n_training_envs = 1
n_eval_envs = 5

# Create log dir where evaluation results will be saved
eval_log_dir = "./eval_logs/"
os.makedirs(eval_log_dir, exist_ok=True)

# Initialize a vectorized training environment with default parameters
train_env = make_vec_env(env_id, n_envs=n_training_envs, seed=0)

# Separate evaluation env, with different parameters passed via env_kwargs
# Eval environments can be vectorized to speed up evaluation.
eval_env = make_vec_env(env_id, n_envs=n_eval_envs, seed=0,
                        env_kwargs={'g':0.7})

# Create callback that evaluates agent for 5 episodes every 500 training environment␣
↪steps.
# When using multiple training environments, agent will be evaluated every
# eval_freq calls to train_env.step(), thus it will be evaluated every
```

(continues on next page)

(continued from previous page)

```
# (eval_freq * n_envs) training steps. See EvalCallback doc for more information.
eval_callback = EvalCallback(eval_env, best_model_save_path=eval_log_dir,
                              log_path=eval_log_dir, eval_freq=max(500 // n_training_
↪envs, 1),
                              n_eval_episodes=5, deterministic=True,
                              render=False)

model = SAC("MlpPolicy", train_env)
model.learn(5000, callback=eval_callback)
```

1.6.8 Atari Games

Fig. 3: Trained A2C agent on Breakout

Fig. 4: Pong Environment

Training a RL agent on Atari games is straightforward thanks to make_atari_env helper function. It will do all the preprocessing and multiprocessing for you. To install the Atari environments, run the command pip install gymnasium[atari,accept-rom-license] to install the Atari environments and ROMs, or install Stable Baselines3 with pip install stable-baselines3[extra] to install this and other optional dependencies.

```
from stable_baselines3.common.env_util import make_atari_env
from stable_baselines3.common.vec_env import VecFrameStack
from stable_baselines3 import A2C

import ale_py

# There already exists an environment generator
# that will make and wrap atari environments correctly.
# Here we are also multi-worker training (n_envs=4 => 4 environments)
vec_env = make_atari_env("PongNoFrameskip-v4", n_envs=4, seed=0)
# Frame-stacking with 4 frames
vec_env = VecFrameStack(vec_env, n_stack=4)

model = A2C("CnnPolicy", vec_env, verbose=1)
model.learn(total_timesteps=25_000)

obs = vec_env.reset()
while True:
    action, _states = model.predict(obs, deterministic=False)
    obs, rewards, dones, info = vec_env.step(action)
    vec_env.render("human")
```

1.6.9 PyBullet: Normalizing input features

Normalizing input features may be essential to successful training of an RL agent (by default, images are scaled, but other types of input are not), for instance when training on PyBullet environments. For this, there is a wrapper VecNormalize that will compute a running average and standard deviation of the input features (it can do the same for rewards).

Note

you need to install pybullet envs with `pip install pybullet_envs_gymnasium`

```python
from pathlib import Path

import pybullet_envs_gymnasium

from stable_baselines3.common.vec_env import VecNormalize
from stable_baselines3.common.env_util import make_vec_env
from stable_baselines3 import PPO

# Alternatively, you can use the MuJoCo equivalent "HalfCheetah-v4"
vec_env = make_vec_env("HalfCheetahBulletEnv-v0", n_envs=1)
# Automatically normalize the input features and reward
vec_env = VecNormalize(vec_env, norm_obs=True, norm_reward=True, clip_obs=10.0)

model = PPO("MlpPolicy", vec_env)
model.learn(total_timesteps=2000)

# Don't forget to save the VecNormalize statistics when saving the agent
log_dir = Path("/tmp/")
model.save(log_dir / "ppo_halfcheetah")
stats_path = log_dir / "vec_normalize.pkl"
vec_env.save(stats_path)

# To demonstrate loading
del model, vec_env

# Load the saved statistics
vec_env = make_vec_env("HalfCheetahBulletEnv-v0", n_envs=1)
vec_env = VecNormalize.load(stats_path, vec_env)
#  do not update them at test time
vec_env.training = False
# reward normalization is not needed at test time
vec_env.norm_reward = False

# Load the agent
model = PPO.load(log_dir / "ppo_halfcheetah", env=vec_env)
```

1.6.10 Hindsight Experience Replay (HER)

For this example, we are using Highway-Env by @eleurent.

Fig. 5: The highway-parking-v0 environment.

The parking env is a goal-conditioned continuous control task, in which the vehicle must park in a given space with the appropriate heading.

Note

The hyperparameters in the following example were optimized for that environment.

```python
import gymnasium as gym
import highway_env
import numpy as np

from stable_baselines3 import HerReplayBuffer, SAC, DDPG, TD3
from stable_baselines3.common.noise import NormalActionNoise

env = gym.make("parking-v0")

# Create 4 artificial transitions per real transition
n_sampled_goal = 4

# SAC hyperparams:
model = SAC(
    "MultiInputPolicy",
    env,
    replay_buffer_class=HerReplayBuffer,
    replay_buffer_kwargs=dict(
      n_sampled_goal=n_sampled_goal,
      goal_selection_strategy="future",
    ),
    verbose=1,
    buffer_size=int(1e6),
    learning_rate=1e-3,
    gamma=0.95,
    batch_size=256,
    policy_kwargs=dict(net_arch=[256, 256, 256]),
)

model.learn(int(2e5))
model.save("her_sac_highway")

# Load saved model
# Because it needs access to `env.compute_reward()`
# HER must be loaded with the env
env = gym.make("parking-v0", render_mode="human") # Change the render mode
model = SAC.load("her_sac_highway", env=env)

obs, info = env.reset()

# Evaluate the agent
episode_reward = 0
for _ in range(100):
    action, _ = model.predict(obs, deterministic=True)
    obs, reward, terminated, truncated, info = env.step(action)
    episode_reward += reward
    if terminated or truncated or info.get("is_success", False):
        print("Reward:", episode_reward, "Success?", info.get("is_success", False))
```

(continues on next page)

(continued from previous page)

```
        episode_reward = 0.0
        obs, info = env.reset()
```

1.6.11 Learning Rate Schedule

All algorithms allow you to pass a learning rate schedule that takes as input the current progress remaining (from 1 to 0). PPO's clip_range` parameter also accepts such schedule.

The RL Zoo already includes linear and constant schedules.

```
from typing import Callable

from stable_baselines3 import PPO


def linear_schedule(initial_value: float) -> Callable[[float], float]:
    """
    Linear learning rate schedule.

    :param initial_value: Initial learning rate.
    :return: schedule that computes
      current learning rate depending on remaining progress
    """
    def func(progress_remaining: float) -> float:
        """
        Progress will decrease from 1 (beginning) to 0.

        :param progress_remaining:
        :return: current learning rate
        """
        return progress_remaining * initial_value

    return func

# Initial learning rate of 0.001
model = PPO("MlpPolicy", "CartPole-v1", learning_rate=linear_schedule(0.001), verbose=1)
model.learn(total_timesteps=20_000)
# By default, `reset_num_timesteps` is True, in which case the learning rate schedule␣
↪resets.
# progress_remaining = 1.0 - (num_timesteps / total_timesteps)
model.learn(total_timesteps=10_000, reset_num_timesteps=True)
```

1.6.12 Advanced Saving and Loading

In this example, we show how to use a policy independently from a model (and how to save it, load it) and save/load a replay buffer.

By default, the replay buffer is not saved when calling model.save(), in order to save space on the disk (a replay buffer can be up to several GB when using images). However, SB3 provides a save_replay_buffer() and load_replay_buffer() method to save it separately.

Note

For training model after loading it, we recommend loading the replay buffer to ensure stable learning (for off-policy algorithms). You also need to pass `reset_num_timesteps=True` to `learn` function which initializes the environment and agent for training if a new environment was created since saving the model.

```
from stable_baselines3 import SAC
from stable_baselines3.common.evaluation import evaluate_policy
from stable_baselines3.sac.policies import MlpPolicy

# Create the model and the training environment
model = SAC("MlpPolicy", "Pendulum-v1", verbose=1,
            learning_rate=1e-3)

# train the model
model.learn(total_timesteps=6000)

# save the model
model.save("sac_pendulum")

# the saved model does not contain the replay buffer
loaded_model = SAC.load("sac_pendulum")
print(f"The loaded_model has {loaded_model.replay_buffer.size()} transitions in its␣
↪buffer")

# now save the replay buffer too
model.save_replay_buffer("sac_replay_buffer")

# load it into the loaded_model
loaded_model.load_replay_buffer("sac_replay_buffer")

# now the loaded replay is not empty anymore
print(f"The loaded_model has {loaded_model.replay_buffer.size()} transitions in its␣
↪buffer")

# Save the policy independently from the model
# Note: if you don't save the complete model with `model.save()`
# you cannot continue training afterward
policy = model.policy
policy.save("sac_policy_pendulum")

# Retrieve the environment
env = model.get_env()

# Evaluate the policy
mean_reward, std_reward = evaluate_policy(policy, env, n_eval_episodes=10,␣
↪deterministic=True)

print(f"mean_reward={mean_reward:.2f} +/- {std_reward}")
```

(continues on next page)

(continued from previous page)

```
# Load the policy independently from the model
saved_policy = MlpPolicy.load("sac_policy_pendulum")

# Evaluate the loaded policy
mean_reward, std_reward = evaluate_policy(saved_policy, env, n_eval_episodes=10,␣
↪deterministic=True)

print(f"mean_reward={mean_reward:.2f} +/- {std_reward}")
```

1.6.13 Accessing and modifying model parameters

You can access model's parameters via `set_parameters` and `get_parameters` functions, or via `model.policy.state_dict()` (and `load_state_dict()`), which use dictionaries that map variable names to PyTorch tensors.

These functions are useful when you need to e.g. evaluate large set of models with same network structure, visualize different layers of the network or modify parameters manually.

Policies also offers a simple way to save/load weights as a NumPy vector, using `parameters_to_vector()` and `load_from_vector()` method.

Following example demonstrates reading parameters, modifying some of them and loading them to model by implementing evolution strategy (es) for solving the `CartPole-v1` environment. The initial guess for parameters is obtained by running A2C policy gradient updates on the model.

```
from typing import Dict

import gymnasium as gym
import numpy as np
import torch as th

from stable_baselines3 import A2C
from stable_baselines3.common.evaluation import evaluate_policy


def mutate(params: Dict[str, th.Tensor]) -> Dict[str, th.Tensor]:
    """Mutate parameters by adding normal noise to them"""
    return dict((name, param + th.randn_like(param)) for name, param in params.items())


# Create policy with a small network
model = A2C(
    "MlpPolicy",
    "CartPole-v1",
    ent_coef=0.0,
    policy_kwargs={"net_arch": [32]},
    seed=0,
    learning_rate=0.05,
)

# Use traditional actor-critic policy gradient updates to
# find good initial parameters
model.learn(total_timesteps=10_000)
```

(continues on next page)

(continued from previous page)

```python
# Include only variables with "policy", "action" (policy) or "shared_net" (shared layers)
# in their name: only these ones affect the action.
# NOTE: you can retrieve those parameters using model.get_parameters() too
mean_params = dict(
    (key, value)
    for key, value in model.policy.state_dict().items()
    if ("policy" in key or "shared_net" in key or "action" in key)
)

# population size of 50 individuals
pop_size = 50
# Keep top 10%
n_elite = pop_size // 10
# Retrieve the environment
vec_env = model.get_env()

for iteration in range(10):
    # Create population of candidates and evaluate them
    population = []
    for population_i in range(pop_size):
        candidate = mutate(mean_params)
        # Load new policy parameters to agent.
        # Tell function that it should only update parameters
        # we give it (policy parameters)
        model.policy.load_state_dict(candidate, strict=False)
        # Evaluate the candidate
        fitness, _ = evaluate_policy(model, vec_env)
        population.append((candidate, fitness))
    # Take top 10% and use average over their parameters as next mean parameter
    top_candidates = sorted(population, key=lambda x: x[1], reverse=True)[:n_elite]
    mean_params = dict(
        (
            name,
            th.stack([candidate[0][name] for candidate in top_candidates]).mean(dim=0),
        )
        for name in mean_params.keys()
    )
    mean_fitness = sum(top_candidate[1] for top_candidate in top_candidates) / n_elite
    print(f"Iteration {iteration + 1:<3} Mean top fitness: {mean_fitness:.2f}")
    print(f"Best fitness: {top_candidates[0][1]:.2f}")
```

1.6.14 SB3 with Isaac Lab, Brax, Procgen, EnvPool

Some massively parallel simulations such as EnvPool, Isaac Lab, Brax or ProcGen already produce a vectorized environment to speed up data collection (see discussion in issue #314).

To use SB3 with these tools, you need to wrap the env with tool-specific `VecEnvWrapper` that pre-processes the data for SB3, you can find links to some of these wrappers in issue #772.

- Isaac Lab wrapper: link
- Brax: link
- EnvPool: link

1.6.15 SB3 with DeepMind Control (dm_control)

If you want to use SB3 with dm_control, you need to use two wrappers (one from shimmy, one pre-built one) to convert it to a Gymnasium compatible environment:

```
import shimmy
import stable_baselines3 as sb3
from dm_control import suite
from gymnasium.wrappers import FlattenObservation

# Available envs:
# suite._DOMAINS and suite.dog.SUITE

env = suite.load(domain_name="dog", task_name="run")
gym_env = FlattenObservation(shimmy.DmControlCompatibilityV0(env))

model = sb3.PPO("MlpPolicy", gym_env, verbose=1)
model.learn(10_000, progress_bar=True)
```

1.6.16 Record a Video

Record a mp4 video (here using a random agent).

Note

It requires `ffmpeg` or `avconv` to be installed on the machine.

```
import gymnasium as gym
from stable_baselines3.common.vec_env import VecVideoRecorder, DummyVecEnv

env_id = "CartPole-v1"
video_folder = "logs/videos/"
video_length = 100

vec_env = DummyVecEnv([lambda: gym.make(env_id, render_mode="rgb_array")])

obs = vec_env.reset()

# Record the video starting at the first step
vec_env = VecVideoRecorder(vec_env, video_folder,
                       record_video_trigger=lambda x: x == 0, video_length=video_length,
                       name_prefix=f"random-agent-{env_id}")

vec_env.reset()
for _ in range(video_length + 1):
  action = [vec_env.action_space.sample()]
  obs, _, _, _ = vec_env.step(action)
# Save the video
vec_env.close()
```

1.6.17 Bonus: Make a GIF of a Trained Agent

```python
import imageio
import numpy as np

from stable_baselines3 import A2C

model = A2C("MlpPolicy", "LunarLander-v2").learn(100_000)

images = []
obs = model.env.reset()
img = model.env.render(mode="rgb_array")
for i in range(350):
    images.append(img)
    action, _ = model.predict(obs)
    obs, _, _ ,_ = model.env.step(action)
    img = model.env.render(mode="rgb_array")

imageio.mimsave("lander_a2c.gif", [np.array(img) for i, img in enumerate(images) if i%2 == 0], fps=29)
```

1.7 Vectorized Environments

Vectorized Environments are a method for stacking multiple independent environments into a single environment. Instead of training an RL agent on 1 environment per step, it allows us to train it on `n` environments per step. Because of this, `actions` passed to the environment are now a vector (of dimension `n`). It is the same for `observations`, `rewards` and end of episode signals (`dones`). In the case of non-array observation spaces such as `Dict` or `Tuple`, where different sub-spaces may have different shapes, the sub-observations are vectors (of dimension `n`).

Name	`Box`	`Discrete`	`Dict`	`Tuple`	Multi Processing
DummyVecEnv	✓	✓	✓	✓	
SubprocVecEnv	✓	✓	✓	✓	✓

Note

Vectorized environments are required when using wrappers for frame-stacking or normalization.

Note

When using vectorized environments, the environments are automatically reset at the end of each episode. Thus, the observation returned for the i-th environment when `done[i]` is true will in fact be the first observation of the next episode, not the last observation of the episode that has just terminated. You can access the "real" final observation of the terminated episode—that is, the one that accompanied the `done` event provided by the underlying environment—using the `terminal_observation` keys in the info dicts returned by the `VecEnv`.

Warning

When defining a custom `VecEnv` (for instance, using gym3 `ProcgenEnv`), you should provide `terminal_observation` keys in the info dicts returned by the `VecEnv` (cf. note above).

Warning

When using `SubprocVecEnv`, users must wrap the code in an `if __name__ == "__main__":` if using the `forkserver` or `spawn` start method (default on Windows). On Linux, the default start method is `fork` which is not thread safe and can create deadlocks.

For more information, see Python's multiprocessing guidelines.

1.7.1 VecEnv API vs Gym API

For consistency across Stable-Baselines3 (SB3) versions and because of its special requirements and features, SB3 VecEnv API is not the same as Gym API. SB3 VecEnv API is actually close to Gym 0.21 API but differs to Gym 0.26+ API:

- the `reset()` method only returns the observation (`obs = vec_env.reset()`) and not a tuple, the info at reset are stored in `vec_env.reset_infos`.
- only the initial call to `vec_env.reset()` is required, environments are reset automatically afterward (and `reset_infos` is updated automatically).
- the `vec_env.step(actions)` method expects an array as input (with a batch size corresponding to the number of environments) and returns a 4-tuple (and not a 5-tuple): `obs, rewards, dones, infos` instead of `obs, reward, terminated, truncated, info` where `dones = terminated or truncated` (for each env). `obs, rewards, dones` are NumPy arrays with shape `(n_envs, shape_for_single_env)` (so with a batch dimension). Additional information is passed via the `infos` value which is a list of dictionaries.
- at the end of an episode, `infos[env_idx]["TimeLimit.truncated"] = truncated and not terminated` tells the user if an episode was truncated or not: you should bootstrap if `infos[env_idx]["TimeLimit.truncated"] is True` (episode over due to a timeout/truncation) or `dones[env_idx] is False` (episode not finished). Note: compared to Gym 0.26+ `infos[env_idx]["TimeLimit.truncated"]` and `terminated` are mutually exclusive. The conversion from SB3 to Gym API is

```python
# done is True at the end of an episode
# dones[env_idx] = terminated[env_idx] or truncated[env_idx]
# In SB3, truncated and terminated are mutually exclusive
# infos[env_idx]["TimeLimit.truncated"] = truncated and not terminated
# terminated[env_idx] tells you whether you should bootstrap or not:
# when the episode has not ended or when the termination was a timeout/truncation
terminated[env_idx] = dones[env_idx] and not infos[env_idx]["TimeLimit.truncated"]
should_bootstrap[env_idx] = not terminated[env_idx]
```

- at the end of an episode, because the environment resets automatically, we provide `infos[env_idx]["terminal_observation"]` which contains the last observation of an episode (and can be used when bootstrapping, see note in the previous section)
- to overcome the current Gymnasium limitation (only one render mode allowed per env instance, see issue #100), we recommend using `render_mode="rgb_array"` since we can both have the image as a numpy array and display it with OpenCV. if no mode is passed or `mode="rgb_array"` is passed when calling `vec_env.render` then we use the default mode, otherwise, we use the OpenCV display. Note that if `render_mode != "rgb_array"`, you can only call `vec_env.render()` (without argument or with `mode=env.render_mode`).

- the `reset()` method doesn't take any parameter. If you want to seed the pseudo-random generator or pass options, you should call `vec_env.seed(seed=seed)/vec_env.set_options(options)` and `obs = vec_env.reset()` afterward (seed and options are discarded after each call to `reset()`).
- methods and attributes of the underlying Gym envs can be accessed, called and set using `vec_env.get_attr("attribute_name")`, `vec_env.env_method("method_name", args1, args2, kwargs1=kwargs1)` and `vec_env.set_attr("attribute_name", new_value)`.

1.7.2 Modifying Vectorized Environments Attributes

If you plan to modify the attributes of an environment while it is used (e.g., modifying an attribute specifying the task carried out for a portion of training when doing multi-task learning, or a parameter of the environment dynamics), you must expose a setter method. In fact, directly accessing the environment attribute in the callback can lead to unexpected behavior because environments can be wrapped (using gym or VecEnv wrappers, the `Monitor` wrapper being one example).

Consider the following example for a custom env:

```
import gymnasium as gym
from gymnasium import spaces

from stable_baselines3.common.env_util import make_vec_env


class MyMultiTaskEnv(gym.Env):

  def __init__(self):
      super().__init__()
      """
      A state and action space for robotic locomotion.
      The multi-task twist is that the policy would need to adapt to different terrains,␣
↪each with its own
      friction coefficient, mu.
      The friction coefficient is the only parameter that changes between tasks.
      mu is a scalar between 0 and 1, and during training a callback is used to update␣
↪mu.
      """
      ...

  def step(self, action):
    # Do something, depending on the action and current value of mu the next state is␣
↪computed
    return self._get_obs(), reward, done, truncated, info

  def set_mu(self, new_mu: float) -> None:
      # Note: this value should be used only at the next reset
      self.mu = new_mu

# Example of wrapped env
# env is of type <TimeLimit<OrderEnforcing<PassiveEnvChecker<CartPoleEnv<CartPole-v1>>>>>
env = gym.make("CartPole-v1")
# To access the base env, without wrapper, you should use `.unwrapped`
# or env.get_wrapper_attr("gravity") to include wrappers
env.unwrapped.gravity
# SB3 uses VecEnv for training, where `env.unwrapped.x = new_value` cannot be used to set␣
```

(continues on next page)

(continued from previous page)

```
↪an attribute
# therefore, you should expose a setter like `set_mu` to properly set an attribute
vec_env = make_vec_env(MyMultiTaskEnv)
# Print current mu value
# Note: you should use vec_env.env_method("get_wrapper_attr", "mu") in Gymnasium v1.0
print(vec_env.env_method("get_wrapper_attr", "mu"))
# Change `mu` attribute via the setter
vec_env.env_method("set_mu", "mu", 0.1)
# If the variable exists, you can also use `set_wrapper_attr` to set it
assert vec_env.has_attr("mu")
vec_env.env_method("set_wrapper_attr", "mu", 0.1)
```

In this example `env.mu` cannot be accessed/changed directly because it is wrapped in a `VecEnv` and because it could be wrapped with other wrappers (see GH#1573 for a longer explanation). Instead, the callback should use the `set_mu` method via the `env_method` method for Vectorized Environments.

```
from itertools import cycle

class ChangeMuCallback(BaseCallback):
  """
  This callback changes the value of mu during training looping
  through a list of values until training is aborted.
  The environment is implemented so that the impact of changing
  the value of mu mid-episode is visible only after the episode is over
  and the reset method has been called.
  """"
  def __init__(self):
    super().__init__()
    # An iterator that contains the different of the friction coefficient
    self.mus = cycle([0.1, 0.2, 0.5, 0.13, 0.9])

  def _on_step(self):
    # Note: in practice, you should not change this value at every step
    # but rather depending on some events/metrics like agent performance/episode␣
↪termination
    # both accessible via the `self.logger` or `self.locals` variables
    self.training_env.env_method("set_mu", next(self.mus))
```

This callback can then be used to safely modify environment attributes during training since it calls the environment setter method.

1.7.3 Vectorized Environments Wrappers

If you want to alter or augment a `VecEnv` without redefining it completely (e.g. stack multiple frames, monitor the `VecEnv`, normalize the observation, . . .), you can use `VecEnvWrapper` for that. They are the vectorized equivalents (i.e., they act on multiple environments at the same time) of `gym.Wrapper`.

You can find below an example for extracting one key from the observation:

```
import numpy as np

from stable_baselines3.common.vec_env.base_vec_env import VecEnv, VecEnvStepReturn,␣
↪VecEnvWrapper
```

(continues on next page)

(continued from previous page)

```
class VecExtractDictObs(VecEnvWrapper):
    """
    A vectorized wrapper for filtering a specific key from dictionary observations.
    Similar to Gym's FilterObservation wrapper:
        https://github.com/openai/gym/blob/master/gym/wrappers/filter_observation.py

    :param venv: The vectorized environment
    :param key: The key of the dictionary observation
    """

    def __init__(self, venv: VecEnv, key: str):
        self.key = key
        super().__init__(venv=venv, observation_space=venv.observation_space.spaces[self.
↪key])

    def reset(self) -> np.ndarray:
        obs = self.venv.reset()
        return obs[self.key]

    def step_async(self, actions: np.ndarray) -> None:
        self.venv.step_async(actions)

    def step_wait(self) -> VecEnvStepReturn:
        obs, reward, done, info = self.venv.step_wait()
        return obs[self.key], reward, done, info

env = DummyVecEnv([lambda: gym.make("FetchReach-v1")])
# Wrap the VecEnv
env = VecExtractDictObs(env, key="observation")
```

Note

When creating a vectorized environment, you can also specify ordinary gymnasium wrappers to wrap each of the sub-environments. See the *make_vec_env* documentation for details. Example:

```
from gymnasium.wrappers import RescaleAction
from stable_baselines3.common.env_util import make_vec_env

# Use gym wrapper for each sub-env of the VecEnv
wrapper_kwargs = dict(min_action=-1.0, max_action=1.0)
vec_env = make_vec_env(
    "Pendulum-v1", n_envs=2, wrapper_class=RescaleAction, wrapper_kwargs=wrapper_
↪kwargs
)
```

1.7.4 VecEnv

`class stable_baselines3.common.vec_env.`**`VecEnv`**(*num_envs*, *observation_space*, *action_space*)

An abstract asynchronous, vectorized environment.

Parameters

- **num_envs** (*int*) – Number of environments
- **observation_space** (*Space*) – Observation space
- **action_space** (*Space*) – Action space

`abstractmethod` **`close`**()

Clean up the environment's resources.

Return type

None

`abstractmethod` **`env_is_wrapped`**(*wrapper_class*, *indices=None*)

Check if environments are wrapped with a given wrapper.

Parameters

- **method_name** – The name of the environment method to invoke.
- **indices** (*None | int | Iterable[int]*) – Indices of envs whose method to call
- **method_args** – Any positional arguments to provide in the call
- **method_kwargs** – Any keyword arguments to provide in the call
- **wrapper_class** (*type[Wrapper]*)

Returns

True if the env is wrapped, False otherwise, for each env queried.

Return type

list[bool]

`abstractmethod` **`env_method`**(*method_name*, **method_args*, *indices=None*, ***method_kwargs*)

Call instance methods of vectorized environments.

Parameters

- **method_name** (*str*) – The name of the environment method to invoke.
- **indices** (*None | int | Iterable[int]*) – Indices of envs whose method to call
- **method_args** – Any positional arguments to provide in the call
- **method_kwargs** – Any keyword arguments to provide in the call

Returns

List of items returned by the environment's method call

Return type

list[*Any*]

`abstractmethod` **`get_attr`**(*attr_name*, *indices=None*)

Return attribute from vectorized environment.

Parameters

- **attr_name** (*str*) – The name of the attribute whose value to return
- **indices** (*None | int | Iterable[int]*) – Indices of envs to get attribute from

Returns
List of values of 'attr_name' in all environments

Return type
list[*Any*]

`get_images()`

Return RGB images from each environment when available

Return type
Sequence[*ndarray* | None]

`getattr_depth_check(`*name*, *already_found*`)`

Check if an attribute reference is being hidden in a recursive call to __getattr__

Parameters

- `name` (*str*) – name of attribute to check for
- `already_found` (*bool*) – whether this attribute has already been found in a wrapper

Returns
name of module whose attribute is being shadowed, if any.

Return type
str | None

`has_attr(`*attr_name*`)`

Check if an attribute exists for a vectorized environment.

Parameters
`attr_name` (*str*) – The name of the attribute to check

Returns
True if 'attr_name' exists in all environments

Return type
bool

`render(`*mode=None*`)`

Gym environment rendering

Parameters
`mode` (*str | None*) – the rendering type

Return type
ndarray | None

`abstractmethod reset()`

Reset all the environments and return an array of observations, or a tuple of observation arrays.

If step_async is still doing work, that work will be cancelled and step_wait() should not be called until step_async() is invoked again.

Returns
observation

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

`seed`(*seed=None*)

Sets the random seeds for all environments, based on a given seed. Each individual environment will still get its own seed, by incrementing the given seed. WARNING: since gym 0.26, those seeds will only be passed to the environment at the next reset.

Parameters
`seed` (`int | None`) – The random seed. May be None for completely random seeding.

Returns
Returns a list containing the seeds for each individual env. Note that all list elements may be None, if the env does not return anything when being seeded.

Return type
Sequence[None | int]

`abstractmethod set_attr`(*attr_name*, *value*, *indices=None*)

Set attribute inside vectorized environments.

Parameters

- `attr_name` (`str`) – The name of attribute to assign new value
- `value` (`Any`) – Value to assign to *attr_name*
- `indices` (`None | int | Iterable[int]`) – Indices of envs to assign value

Returns

Return type
None

`set_options`(*options=None*)

Set environment options for all environments. If a dict is passed instead of a list, the same options will be used for all environments. WARNING: Those options will only be passed to the environment at the next reset.

Parameters
`options` (`list[dict] | dict | None`) – A dictionary of environment options to pass to each environment at the next reset.

Return type
None

`step`(*actions*)

Step the environments with the given action

Parameters
`actions` (`ndarray`) – the action

Returns
observation, reward, done, information

Return type
tuple[*ndarray* | dict[str, *ndarray*] | tuple[*ndarray*, . . .], *ndarray*, *ndarray*, list[dict]]

`abstractmethod step_async`(*actions*)

Tell all the environments to start taking a step with the given actions. Call step_wait() to get the results of the step.

You should not call this if a step_async run is already pending.

Parameters
`actions` (`ndarray`)

Return type
None

`abstractmethod step_wait()`

Wait for the step taken with step_async().

Returns
observation, reward, done, information

Return type
tuple[*ndarray* | dict[str, *ndarray*] | tuple[*ndarray*, . . .], *ndarray*, *ndarray*, list[dict]]

1.7.5 DummyVecEnv

`class stable_baselines3.common.vec_env.DummyVecEnv(`*env_fns*`)`

Creates a simple vectorized wrapper for multiple environments, calling each environment in sequence on the current Python process. This is useful for computationally simple environment such as `Cartpole-v1`, as the overhead of multiprocess or multithread outweighs the environment computation time. This can also be used for RL methods that require a vectorized environment, but that you want a single environments to train with.

Parameters
env_fns (*list[Callable[[], Env]]*) – a list of functions that return environments to vectorize

Raises
ValueError – If the same environment instance is passed as the output of two or more different env_fn.

`close()`

Clean up the environment's resources.

Return type
None

`env_is_wrapped(`*wrapper_class*, *indices=None*`)`

Check if worker environments are wrapped with a given wrapper

Parameters

- **wrapper_class** (*type[Wrapper]*)
- **indices** (*None | int | Iterable[int]*)

Return type
list[bool]

`env_method(`*method_name*, **method_args*, *indices=None*, ***method_kwargs*`)`

Call instance methods of vectorized environments.

Parameters

- **method_name** (*str*)
- **indices** (*None | int | Iterable[int]*)

Return type
list[*Any*]

`get_attr(`*attr_name*, *indices=None*`)`

Return attribute from vectorized environment (see base class).

Parameters

- **attr_name** (*str*)
- **indices** (*None | int | Iterable[int]*)

Return type
list[*Any*]

`get_images()`

Return RGB images from each environment when available

Return type
Sequence[*ndarray* | None]

`render(mode=None)`

Gym environment rendering. If there are multiple environments then they are tiled together in one image via `BaseVecEnv.render()`.

Parameters
mode (*str | None*) – The rendering type.

Return type
ndarray | None

`reset()`

Reset all the environments and return an array of observations, or a tuple of observation arrays.

If step_async is still doing work, that work will be cancelled and step_wait() should not be called until step_async() is invoked again.

Returns
observation

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

`set_attr(attr_name, value, indices=None)`

Set attribute inside vectorized environments (see base class).

Parameters

- **attr_name** (*str*)
- **value** (*Any*)
- **indices** (*None | int | Iterable[int]*)

Return type
None

`step_async(actions)`

Tell all the environments to start taking a step with the given actions. Call step_wait() to get the results of the step.

You should not call this if a step_async run is already pending.

Parameters
actions (*ndarray*)

Return type
None

step_wait()

Wait for the step taken with step_async().

Returns

observation, reward, done, information

Return type

tuple[*ndarray* | dict[str, *ndarray*] | tuple[*ndarray*, . . .], *ndarray*, *ndarray*, list[dict]]

1.7.6 SubprocVecEnv

class stable_baselines3.common.vec_env.**SubprocVecEnv**(*env_fns*, *start_method=None*)

Creates a multiprocess vectorized wrapper for multiple environments, distributing each environment to its own process, allowing significant speed up when the environment is computationally complex.

For performance reasons, if your environment is not IO bound, the number of environments should not exceed the number of logical cores on your CPU.

> ⚠ **Warning**
>
> Only 'forkserver' and 'spawn' start methods are thread-safe, which is important when TensorFlow sessions or other non thread-safe libraries are used in the parent (see issue #217). However, compared to 'fork' they incur a small start-up cost and have restrictions on global variables. With those methods, users must wrap the code in an `if __name__ == "__main__":` block. For more information, see the multiprocessing documentation.

Parameters

- **env_fns** (*list[Callable[[], Env]]*) – Environments to run in subprocesses
- **start_method** (*str | None*) – method used to start the subprocesses. Must be one of the methods returned by multiprocessing.get_all_start_methods(). Defaults to 'forkserver' on available platforms, and 'spawn' otherwise.

close()

Clean up the environment's resources.

Return type

None

env_is_wrapped(*wrapper_class*, *indices=None*)

Check if worker environments are wrapped with a given wrapper

Parameters

- **wrapper_class** (*type[Wrapper]*)
- **indices** (*None | int | Iterable[int]*)

Return type

list[bool]

env_method(*method_name*, **method_args*, *indices=None*, ***method_kwargs*)

Call instance methods of vectorized environments.

Parameters

- **method_name** (*str*)

- **indices** (*None | int | Iterable[int]*)

Return type
list[*Any*]

get_attr(*attr_name*, *indices=None*)

Return attribute from vectorized environment (see base class).

Parameters

- **attr_name** (*str*)
- **indices** (*None | int | Iterable[int]*)

Return type
list[*Any*]

get_images()

Return RGB images from each environment when available

Return type
Sequence[*ndarray* | None]

has_attr(*attr_name*)

Check if an attribute exists for a vectorized environment. (see base class).

Parameters
attr_name (*str*)

Return type
bool

reset()

Reset all the environments and return an array of observations, or a tuple of observation arrays.

If step_async is still doing work, that work will be cancelled and step_wait() should not be called until step_async() is invoked again.

Returns
observation

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

set_attr(*attr_name*, *value*, *indices=None*)

Set attribute inside vectorized environments (see base class).

Parameters

- **attr_name** (*str*)
- **value** (*Any*)
- **indices** (*None | int | Iterable[int]*)

Return type
None

step_async(*actions*)

Tell all the environments to start taking a step with the given actions. Call step_wait() to get the results of the step.

You should not call this if a step_async run is already pending.

Parameters
`actions` (*ndarray*)

Return type
None

`step_wait()`

Wait for the step taken with step_async().

Returns
observation, reward, done, information

Return type
tuple[*ndarray* | dict[str, *ndarray*] | tuple[*ndarray*, . . .], *ndarray*, *ndarray*, list[dict]]

1.7.7 Wrappers

VecFrameStack

`class stable_baselines3.common.vec_env.VecFrameStack(`*venv*, *n_stack*, *channels_order=None*`)`

Frame stacking wrapper for vectorized environment. Designed for image observations.

Parameters

- **venv** (`VecEnv`) – Vectorized environment to wrap
- **n_stack** (*int*) – Number of frames to stack
- **channels_order** (*str | Mapping[str, str] | None*) – If "first", stack on first image dimension. If "last", stack on last dimension. If None, automatically detect channel to stack over in case of image observation or default to "last" (default). Alternatively channels_order can be a dictionary which can be used with environments with Dict observation spaces

`reset()`

Reset all environments

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

`step_wait()`

Wait for the step taken with step_async().

Returns
observation, reward, done, information

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

StackedObservations

`class stable_baselines3.common.vec_env.stacked_observations.StackedObservations(`*num_envs*, *n_stack*, *observation_space*, *channels_order=None*`)`

Frame stacking wrapper for data.

Dimension to stack over is either first (channels-first) or last (channels-last), which is detected automatically using `common.preprocessing.is_image_space_channels_first` if observation is an image space.

Parameters

- **num_envs** (*int*) – Number of environments
- **n_stack** (*int*) – Number of frames to stack
- **observation_space** (*Box | Dict*) – Environment observation space
- **channels_order** (*str | Mapping[str, str | None] | None*) – If "first", stack on first image dimension. If "last", stack on last dimension. If None, automatically detect channel to stack over in case of image observation or default to "last". For Dict space, channels_order can also be a dictionary.

`static compute_stacking`(*n_stack*, *observation_space*, *channels_order=None*)

Calculates the parameters in order to stack observations

Parameters

- **n_stack** (*int*) – Number of observations to stack
- **observation_space** (*Box*) – Observation space
- **channels_order** (*str | None*) – Order of the channels

Returns

Tuple of channels_first, stack_dimension, stackedobs, repeat_axis

Return type

tuple[bool, int, tuple[int, ...], int]

`reset`(*observation*)

Reset the stacked_obs, add the reset observation to the stack, and return the stack.

Parameters

observation (*TObs*) – Reset observation

Returns

The stacked reset observation

Return type

TObs

`update`(*observations*, *dones*, *infos*)

Add the observations to the stack and use the dones to update the infos.

Parameters

- **observations** (*TObs*) – Observations
- **dones** (*ndarray*) – Dones
- **infos** (*list[dict[str, Any]]*) – Infos

Returns

Tuple of the stacked observations and the updated infos

Return type

tuple[*TObs*, list[dict[str, *Any*]]]

VecNormalize

`class stable_baselines3.common.vec_env.VecNormalize`(*venv*, *training=True*, *norm_obs=True*, *norm_reward=True*, *clip_obs=10.0*, *clip_reward=10.0*, *gamma=0.99*, *epsilon=1e-08*, *norm_obs_keys=None*)

A moving average, normalizing wrapper for vectorized environment. has support for saving/loading moving average,

Parameters

- **venv** (VecEnv) – the vectorized environment to wrap
- **training** (*bool*) – Whether to update or not the moving average
- **norm_obs** (*bool*) – Whether to normalize observation or not (default: True)
- **norm_reward** (*bool*) – Whether to normalize rewards or not (default: True)
- **clip_obs** (*float*) – Max absolute value for observation
- **clip_reward** (*float*) – Max value absolute for discounted reward
- **gamma** (*float*) – discount factor
- **epsilon** (*float*) – To avoid division by zero
- **norm_obs_keys** (*list[str] | None*) – Which keys from observation dict to normalize. If not specified, all keys will be normalized.

`get_original_obs()`

Returns an unnormalized version of the observations from the most recent step or reset.

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

`get_original_reward()`

Returns an unnormalized version of the rewards from the most recent step.

Return type
ndarray

`static load(`*load_path*, *venv*`)`

Loads a saved VecNormalize object.

Parameters

- **load_path** (*str*) – the path to load from.
- **venv** (VecEnv) – the VecEnv to wrap.

Returns

Return type
VecNormalize

`normalize_obs(`*obs*`)`

Normalize observations using this VecNormalize's observations statistics. Calling this method does not update statistics.

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

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

`normalize_reward(`*reward*`)`

Normalize rewards using this VecNormalize's rewards statistics. Calling this method does not update statistics.

Parameters
reward (*ndarray*)

Return type
ndarray

reset()

Reset all environments :return: first observation of the episode

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

save(*save_path*)

Save current VecNormalize object with all running statistics and settings (e.g. clip_obs)

Parameters
save_path (*str*) – The path to save to

Return type
None

set_venv(*venv*)

Sets the vector environment to wrap to venv.

Also sets attributes derived from this such as *num_env*.

Parameters
venv (VecEnv)

Return type
None

step_wait()

Apply sequence of actions to sequence of environments actions -> (observations, rewards, dones)

where dones is a boolean vector indicating whether each element is new.

Return type
tuple[*ndarray* | dict[str, *ndarray*] | tuple[*ndarray*, ...], *ndarray*, *ndarray*, list[dict]]

VecVideoRecorder

class stable_baselines3.common.vec_env.**VecVideoRecorder**(*venv*, *video_folder*, *record_video_trigger*, *video_length=200*, *name_prefix='rl-video'*)

Wraps a VecEnv or VecEnvWrapper object to record rendered image as mp4 video. It requires ffmpeg or avconv to be installed on the machine.

Note: for now it only allows to record one video and all videos must have at least two frames.

The video recorder code was adapted from Gymnasium v1.0.

Parameters

- **venv** (VecEnv)
- **video_folder** (*str*) – Where to save videos
- **record_video_trigger** (*Callable[[int], bool]*) – Function that defines when to start recording. The function takes the current number of step, and returns whether we should start recording or not.
- **video_length** (*int*) – Length of recorded videos

- **name_prefix** (*str*) – Prefix to the video name

`close()`

Closes the wrapper then the video recorder.

Return type
None

`reset()`

Reset all the environments and return an array of observations, or a tuple of observation arrays.

If step_async is still doing work, that work will be cancelled and step_wait() should not be called until step_async() is invoked again.

Returns
observation

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

`step_wait()`

Wait for the step taken with step_async().

Returns
observation, reward, done, information

Return type
tuple[*ndarray* | dict[str, *ndarray*] | tuple[*ndarray*, . . .], *ndarray*, *ndarray*, list[dict]]

VecCheckNan

class `stable_baselines3.common.vec_env.VecCheckNan`(*venv*, *raise_exception=False*, *warn_once=True*, *check_inf=True*)

NaN and inf checking wrapper for vectorized environment, will raise a warning by default, allowing you to know from what the NaN of inf originated from.

Parameters

- **venv** (VecEnv) – the vectorized environment to wrap
- **raise_exception** (*bool*) – Whether to raise a ValueError, instead of a UserWarning
- **warn_once** (*bool*) – Whether to only warn once.
- **check_inf** (*bool*) – Whether to check for +inf or -inf as well

`check_array_value`(*name*, *value*)

Check for inf and NaN for a single numpy array.

Parameters

- **name** (*str*) – Name of the value being check
- **value** (*ndarray*) – Value (numpy array) to check

Returns
A list of issues found.

Return type
list[tuple[str, str]]

`reset()`

Reset all the environments and return an array of observations, or a tuple of observation arrays.

If step_async is still doing work, that work will be cancelled and step_wait() should not be called until step_async() is invoked again.

Returns
observation

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

`step_async`(*actions*)

Tell all the environments to start taking a step with the given actions. Call step_wait() to get the results of the step.

You should not call this if a step_async run is already pending.

Parameters
`actions` (*ndarray*)

Return type
None

`step_wait()`

Wait for the step taken with step_async().

Returns
observation, reward, done, information

Return type
tuple[*ndarray* | dict[str, *ndarray*] | tuple[*ndarray*, . . .], *ndarray*, *ndarray*, list[dict]]

VecTransposeImage

`class stable_baselines3.common.vec_env.VecTransposeImage`(*venv*, *skip=False*)

Re-order channels, from HxWxC to CxHxW. It is required for PyTorch convolution layers.

Parameters

- **venv** (`VecEnv`)
- **skip** (*bool*) – Skip this wrapper if needed as we rely on heuristic to apply it or not, which may result in unwanted behavior, see GH issue #671.

`close()`

Clean up the environment's resources.

Return type
None

`reset()`

Reset all environments

Return type
ndarray | dict

`step_wait()`

Wait for the step taken with step_async().

Returns
observation, reward, done, information

Return type
tuple[*ndarray* | dict[str, *ndarray*] | tuple[*ndarray*, ...], *ndarray*, *ndarray*, list[dict]]

`static transpose_image`(*image*)

Transpose an image or batch of images (re-order channels).

Parameters
image (*ndarray*)

Returns

Return type
ndarray

`transpose_observations`(*observations*)

Transpose (if needed) and return new observations.

Parameters
observations (*ndarray | dict*)

Returns
Transposed observations

Return type
ndarray | dict

`static transpose_space`(*observation_space*, *key=''*)

Transpose an observation space (re-order channels).

Parameters

- **observation_space** (*Box*)
- **key** (*str*) – In case of dictionary space, the key of the observation space.

Returns

Return type
Box

VecMonitor

`class stable_baselines3.common.vec_env.VecMonitor`(*venv*, *filename=None*, *info_keywords=()*)

A vectorized monitor wrapper for *vectorized* Gym environments, it is used to record the episode reward, length, time and other data.

Some environments like openai/procgen or gym3 directly initialize the vectorized environments, without giving us a chance to use the `Monitor` wrapper. So this class simply does the job of the `Monitor` wrapper on a vectorized level.

Parameters

- **venv** (VecEnv) – The vectorized environment
- **filename** (*str | None*) – the location to save a log file, can be None for no log
- **info_keywords** (*tuple[str, ...]*) – extra information to log, from the information return of env.step()

`close`()

Clean up the environment's resources.

Return type
None

reset()

Reset all the environments and return an array of observations, or a tuple of observation arrays.

If step_async is still doing work, that work will be cancelled and step_wait() should not be called until step_async() is invoked again.

Returns
observation

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

step_wait()

Wait for the step taken with step_async().

Returns
observation, reward, done, information

Return type
tuple[*ndarray* | dict[str, *ndarray*] | tuple[*ndarray*, . . .], *ndarray*, *ndarray*, list[dict]]

VecExtractDictObs

class stable_baselines3.common.vec_env.**VecExtractDictObs**(*venv*, *key*)

A vectorized wrapper for extracting dictionary observations.

Parameters

- **venv** (VecEnv) – The vectorized environment
- **key** (*str*) – The key of the dictionary observation

reset()

Reset all the environments and return an array of observations, or a tuple of observation arrays.

If step_async is still doing work, that work will be cancelled and step_wait() should not be called until step_async() is invoked again.

Returns
observation

Return type
ndarray

step_wait()

Wait for the step taken with step_async().

Returns
observation, reward, done, information

Return type
tuple[*ndarray* | dict[str, *ndarray*] | tuple[*ndarray*, . . .], *ndarray*, *ndarray*, list[dict]]

1.8 Policy Networks

Stable Baselines3 provides policy networks for images (CnnPolicies), other type of input features (MlpPolicies) and multiple different inputs (MultiInputPolicies).

Warning

For A2C and PPO, continuous actions are clipped during training and testing (to avoid out of bound error). SAC, DDPG and TD3 squash the action, using a `tanh()` transformation, which handles bounds more correctly.

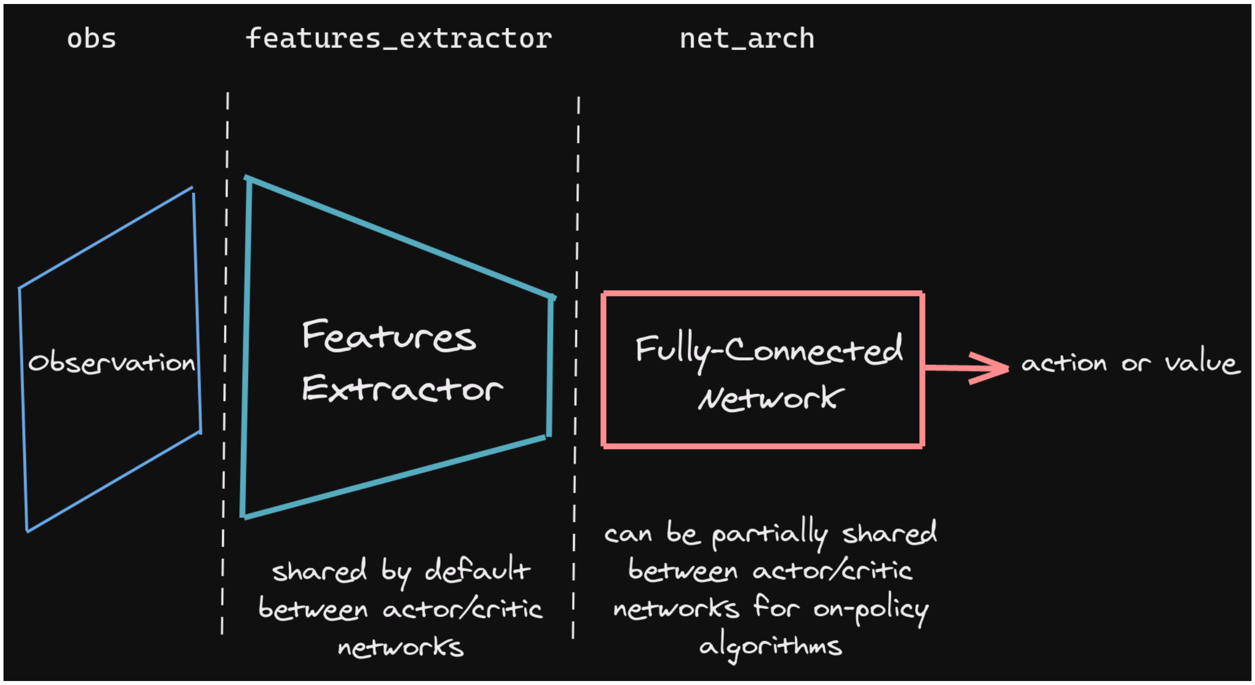
1.8.1 SB3 Policy

SB3 networks are separated into two main parts (see figure below):

- A features extractor (usually shared between actor and critic when applicable, to save computation) whose role is to extract features (i.e. convert to a feature vector) from high-dimensional observations, for instance, a CNN that extracts features from images. This is the `features_extractor_class` parameter. You can change the default parameters of that features extractor by passing a `features_extractor_kwargs` parameter.
- A (fully-connected) network that maps the features to actions/value. Its architecture is controlled by the `net_arch` parameter.

Note

All observations are first pre-processed (e.g. images are normalized, discrete obs are converted to one-hot vectors, ...) before being fed to the features extractor. In the case of vector observations, the features extractor is just a `Flatten` layer.


obs
features_extractor
net_arch
Observation
Features Extractor
Fully-Connected Network
action or value
shared by default between actor/critic networks
can be partially shared between actor/critic networks for on-policy algorithms

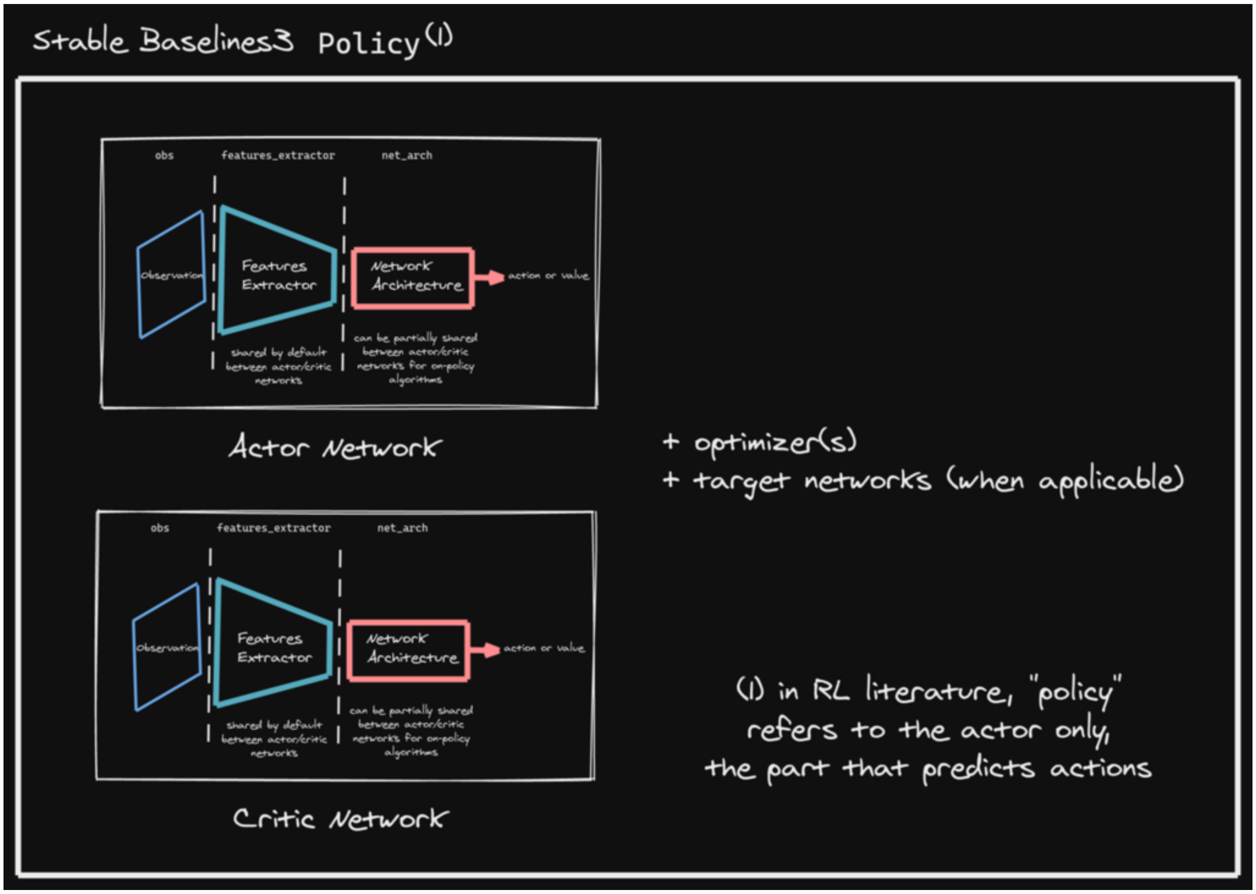
SB3 policies are usually composed of several networks (actor/critic networks + target networks when applicable) together with the associated optimizers.

Each of these network have a features extractor followed by a fully-connected network.

Note

When we refer to "policy" in Stable-Baselines3, this is usually an abuse of language compared to RL terminology.

In SB3, "policy" refers to the class that handles all the networks useful for training, so not only the network used to predict actions (the "learned controller").


Stable Baselines3 Policy (1)
obs
features_extractor
net_arch
Observation
Features Extractor
Network Architecture
action or value
shared by default between actor/critic networks
can be partially shared between actor/critic networks for on-policy algorithms
Actor Network
obs
features_extractor
net_arch
Observation
Features Extractor
Network Architecture
action or value
shared by default between actor/critic networks
can be partially shared between actor/critic networks for on-policy algorithms
Critic Network
+ optimizer(s)
+ target networks (when applicable)
(1) in RL literature, "policy" refers to the actor only, the part that predicts actions

1.8.2 Default Network Architecture

The default network architecture used by SB3 depends on the algorithm and the observation space. You can visualize the architecture by printing `model.policy` (see issue #329).

For 1D observation space, a 2 layers fully connected net is used with:

- 64 units (per layer) for PPO/A2C/DQN
- 256 units for SAC
- [400, 300] units for TD3/DDPG (values are taken from the original TD3 paper)

For image observation spaces, the "Nature CNN" (see code for more details) is used for feature extraction, and SAC/TD3 also keeps the same fully connected network after it. The other algorithms only have a linear layer after the CNN. The CNN is shared between actor and critic for A2C/PPO (on-policy algorithms) to reduce computation. Off-policy algorithms (TD3, DDPG, SAC, ...) have separate feature extractors: one for the actor and one for the critic, since the best performance is obtained with this configuration.

For mixed observations (dictionary observations), the two architectures from above are used, i.e., CNN for images and then two layers fully-connected network (with a smaller output size for the CNN).

1.8.3 Custom Network Architecture

One way of customising the policy network architecture is to pass arguments when creating the model, using `policy_kwargs` parameter:

> **Note**
>
> An extra linear layer will be added on top of the layers specified in `net_arch`, in order to have the right output dimensions and activation functions (e.g. Softmax for discrete actions).
>
> In the following example, as CartPole's action space has a dimension of 2, the final dimensions of the `net_arch`'s layers will be:

```
        obs
        <4>
   /            \
 <32>          <32>
  |              |
 <32>          <32>
  |              |
 <2>            <1>
action         value
```

```python
import gymnasium as gym
import torch as th

from stable_baselines3 import PPO

# Custom actor (pi) and value function (vf) networks
# of two layers of size 32 each with Relu activation function
# Note: an extra linear layer will be added on top of the pi and the vf nets, respectively
policy_kwargs = dict(activation_fn=th.nn.ReLU,
                     net_arch=dict(pi=[32, 32], vf=[32, 32]))
# Create the agent
model = PPO("MlpPolicy", "CartPole-v1", policy_kwargs=policy_kwargs, verbose=1)
# Retrieve the environment
env = model.get_env()
# Train the agent
model.learn(total_timesteps=20_000)
# Save the agent
model.save("ppo_cartpole")

del model
# the policy_kwargs are automatically loaded
model = PPO.load("ppo_cartpole", env=env)
```

1.8.4 Custom Feature Extractor

If you want to have a custom features extractor (e.g. custom CNN when using images), you can define class that derives from `BaseFeaturesExtractor` and then pass it to the model when training.

Note

For on-policy algorithms, the features extractor is shared by default between the actor and the critic to save computation (when applicable). However, this can be changed setting `share_features_extractor=False` in the `policy_kwargs` (both for on-policy and off-policy algorithms).

```python
import torch as th
import torch.nn as nn
from gymnasium import spaces

from stable_baselines3 import PPO
from stable_baselines3.common.torch_layers import BaseFeaturesExtractor


class CustomCNN(BaseFeaturesExtractor):
    """
    :param observation_space: (gym.Space)
    :param features_dim: (int) Number of features extracted.
        This corresponds to the number of unit for the last layer.
    """

    def __init__(self, observation_space: spaces.Box, features_dim: int = 256):
        super().__init__(observation_space, features_dim)
        # We assume CxHxW images (channels first)
        # Re-ordering will be done by pre-preprocessing or wrapper
        n_input_channels = observation_space.shape[0]
        self.cnn = nn.Sequential(
            nn.Conv2d(n_input_channels, 32, kernel_size=8, stride=4, padding=0),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2, padding=0),
            nn.ReLU(),
            nn.Flatten(),
        )

        # Compute shape by doing one forward pass
        with th.no_grad():
            n_flatten = self.cnn(
                th.as_tensor(observation_space.sample()[None]).float()
            ).shape[1]

        self.linear = nn.Sequential(nn.Linear(n_flatten, features_dim), nn.ReLU())

    def forward(self, observations: th.Tensor) -> th.Tensor:
        return self.linear(self.cnn(observations))

policy_kwargs = dict(
    features_extractor_class=CustomCNN,
    features_extractor_kwargs=dict(features_dim=128),
)
model = PPO("CnnPolicy", "BreakoutNoFrameskip-v4", policy_kwargs=policy_kwargs,␣
↪verbose=1)
model.learn(1000)
```

1.8.5 Multiple Inputs and Dictionary Observations

Stable Baselines3 supports handling of multiple inputs by using `Dict` Gym space. This can be done using `MultiInputPolicy`, which by default uses the `CombinedExtractor` features extractor to turn multiple inputs into a single vector, handled by the `net_arch` network.

By default, `CombinedExtractor` processes multiple inputs as follows:

1. If input is an image (automatically detected, see `common.preprocessing.is_image_space`), process image with Nature Atari CNN network and output a latent vector of size `256`.
2. If input is not an image, flatten it (no layers).
3. Concatenate all previous vectors into one long vector and pass it to policy.

Much like above, you can define custom features extractors. The following example assumes the environment has two keys in the observation space dictionary: “image” is a (1,H,W) image (channel first), and “vector” is a (D,) dimensional vector. We process “image” with a simple downsampling and “vector” with a single linear layer.

```python
import gymnasium as gym
import torch as th
from torch import nn

from stable_baselines3.common.torch_layers import BaseFeaturesExtractor

class CustomCombinedExtractor(BaseFeaturesExtractor):
    def __init__(self, observation_space: gym.spaces.Dict):
        # We do not know features-dim here before going over all the items,
        # so put something dummy for now. PyTorch requires calling
        # nn.Module.__init__ before adding modules
        super().__init__(observation_space, features_dim=1)

        extractors = {}

        total_concat_size = 0
        # We need to know size of the output of this extractor,
        # so go over all the spaces and compute output feature sizes
        for key, subspace in observation_space.spaces.items():
            if key == "image":
                # We will just downsample one channel of the image by 4x4 and flatten.
                # Assume the image is single-channel (subspace.shape[0] == 0)
                extractors[key] = nn.Sequential(nn.MaxPool2d(4), nn.Flatten())
                total_concat_size += subspace.shape[1] // 4 * subspace.shape[2] // 4
            elif key == "vector":
                # Run through a simple MLP
                extractors[key] = nn.Linear(subspace.shape[0], 16)
                total_concat_size += 16

        self.extractors = nn.ModuleDict(extractors)

        # Update the features dim manually
        self._features_dim = total_concat_size

    def forward(self, observations) -> th.Tensor:
        encoded_tensor_list = []
```

(continues on next page)

(continued from previous page)

```
        # self.extractors contain nn.Modules that do all the processing.
        for key, extractor in self.extractors.items():
            encoded_tensor_list.append(extractor(observations[key]))
        # Return a (B, self._features_dim) PyTorch tensor, where B is batch dimension.
        return th.cat(encoded_tensor_list, dim=1)
```

1.8.6 On-Policy Algorithms

Custom Networks

If you need a network architecture that is different for the actor and the critic when using `PPO`, `A2C` or `TRPO`, you can pass a dictionary of the following structure: `dict(pi=[<actor network architecture>], vf=[<critic network architecture>])`.

For example, if you want a different architecture for the actor (aka `pi`) and the critic (value-function aka `vf`) networks, then you can specify `net_arch=dict(pi=[32, 32], vf=[64, 64])`.

Otherwise, to have actor and critic that share the same network architecture, you only need to specify `net_arch=[128, 128]` (here, two hidden layers of 128 units each, this is equivalent to `net_arch=dict(pi=[128, 128], vf=[128, 128])`).

If shared layers are needed, you need to implement a custom policy network (see *advanced example below*).

Examples

Same architecture for actor and critic with two layers of size 128: `net_arch=[128, 128]`

```
        obs
   /            \
 <128>          <128>
  |              |
 <128>          <128>
  |              |
action         value
```

Different architectures for actor and critic: `net_arch=dict(pi=[32, 32], vf=[64, 64])`

```
        obs
   /            \
 <32>          <64>
  |              |
 <32>          <64>
  |              |
action         value
```

Advanced Example

If your task requires even more granular control over the policy/value architecture, you can redefine the policy directly:

```
from typing import Callable, Dict, List, Optional, Tuple, Type, Union

from gymnasium import spaces
import torch as th
from torch import nn
```

(continues on next page)

(continued from previous page)

```
from stable_baselines3 import PPO
from stable_baselines3.common.policies import ActorCriticPolicy


class CustomNetwork(nn.Module):
    """
    Custom network for policy and value function.
    It receives as input the features extracted by the features extractor.

    :param feature_dim: dimension of the features extracted with the features_extractor␣
↪(e.g. features from a CNN)
    :param last_layer_dim_pi: (int) number of units for the last layer of the policy␣
↪network
    :param last_layer_dim_vf: (int) number of units for the last layer of the value␣
↪network
    """

    def __init__(
        self,
        feature_dim: int,
        last_layer_dim_pi: int = 64,
        last_layer_dim_vf: int = 64,
    ):
        super().__init__()

        # IMPORTANT:
        # Save output dimensions, used to create the distributions
        self.latent_dim_pi = last_layer_dim_pi
        self.latent_dim_vf = last_layer_dim_vf

        # Policy network
        self.policy_net = nn.Sequential(
            nn.Linear(feature_dim, last_layer_dim_pi), nn.ReLU()
        )
        # Value network
        self.value_net = nn.Sequential(
            nn.Linear(feature_dim, last_layer_dim_vf), nn.ReLU()
        )

    def forward(self, features: th.Tensor) -> Tuple[th.Tensor, th.Tensor]:
        """
        :return: (th.Tensor, th.Tensor) latent_policy, latent_value of the specified␣
↪network.
            If all layers are shared, then ``latent_policy == latent_value``
        """
        return self.forward_actor(features), self.forward_critic(features)

    def forward_actor(self, features: th.Tensor) -> th.Tensor:
        return self.policy_net(features)

    def forward_critic(self, features: th.Tensor) -> th.Tensor:
```

(continues on next page)

(continued from previous page)

```
        return self.value_net(features)


class CustomActorCriticPolicy(ActorCriticPolicy):
    def __init__(
        self,
        observation_space: spaces.Space,
        action_space: spaces.Space,
        lr_schedule: Callable[[float], float],
        *args,
        **kwargs,
    ):
        # Disable orthogonal initialization
        kwargs["ortho_init"] = False
        super().__init__(
            observation_space,
            action_space,
            lr_schedule,
            # Pass remaining arguments to base class
            *args,
            **kwargs,
        )


    def _build_mlp_extractor(self) -> None:
        self.mlp_extractor = CustomNetwork(self.features_dim)


model = PPO(CustomActorCriticPolicy, "CartPole-v1", verbose=1)
model.learn(5000)
```

1.8.7 Off-Policy Algorithms

If you need a network architecture that is different for the actor and the critic when using `SAC`, `DDPG`, `TQC` or `TD3`, you can pass a dictionary of the following structure: `dict(pi=[<actor network architecture>], qf=[<critic network architecture>])`.

For example, if you want a different architecture for the actor (aka `pi`) and the critic (Q-function aka `qf`) networks, then you can specify `net_arch=dict(pi=[64, 64], qf=[400, 300])`.

Otherwise, to have actor and critic that share the same network architecture, you only need to specify `net_arch=[256, 256]` (here, two hidden layers of 256 units each).

Note

For advanced customization of off-policy algorithms policies, please take a look at the code. A good understanding of the algorithm used is required, see discussion in issue #425

```
from stable_baselines3 import SAC

# Custom actor architecture with two layers of 64 units each
```

(continues on next page)

(continued from previous page)

```python
# Custom critic architecture with two layers of 400 and 300 units
policy_kwargs = dict(net_arch=dict(pi=[64, 64], qf=[400, 300]))
# Create the agent
model = SAC("MlpPolicy", "Pendulum-v1", policy_kwargs=policy_kwargs, verbose=1)
model.learn(5000)
```

1.9 Using Custom Environments

To use the RL baselines with custom environments, they just need to follow the *gymnasium* interface. That is to say, your environment must implement the following methods (and inherits from Gym Class):

Note

If you are using images as input, the observation must be of type `np.uint8` and be within a space `Box` bounded by [0, 255] (`Box(low=0, high=255, shape=(<your image shape>)`). By default, the observation is normalized by SB3 pre-processing (dividing by 255 to have values in [0, 1], i.e. `Box(low=0, high=1)`) when using CNN policies. Images can be either channel-first or channel-last.

If you want to use `CnnPolicy` or `MultiInputPolicy` with image-like observation (3D tensor) that are already normalized, you must pass `normalize_images=False` to the policy (using `policy_kwargs` parameter, `policy_kwargs=dict(normalize_images=False)`) and make sure your image is in the **channel-first** format.

Note

Although SB3 supports both channel-last and channel-first images as input, we recommend using the channel-first convention when possible. Under the hood, when a channel-last image is passed, SB3 uses a `VecTransposeImage` wrapper to re-order the channels.

Note

SB3 doesn't support `Discrete` and `MultiDiscrete` spaces with `start!=0`. However, you can update your environment or use a wrapper to make your env compatible with SB3:

```python
import gymnasium as gym

class ShiftWrapper(gym.Wrapper):
"""Allow to use Discrete() action spaces with start!=0"""
def __init__(self, env: gym.Env) -> None:
    super().__init__(env)
    assert isinstance(env.action_space, gym.spaces.Discrete)
    self.action_space = gym.spaces.Discrete(env.action_space.n, start=0)

def step(self, action: int):
    return self.env.step(action + self.env.action_space.start)
```

```python
import gymnasium as gym
import numpy as np
```

(continues on next page)

(continued from previous page)

```
from gymnasium import spaces


class CustomEnv(gym.Env):
    """Custom Environment that follows gym interface."""

    metadata = {"render_modes": ["human"], "render_fps": 30}

    def __init__(self, arg1, arg2, ...):
        super().__init__()
        # Define action and observation space
        # They must be gym.spaces objects
        # Example when using discrete actions:
        self.action_space = spaces.Discrete(N_DISCRETE_ACTIONS)
        # Example for using image as input (channel-first; channel-last also works):
        self.observation_space = spaces.Box(low=0, high=255,
                                            shape=(N_CHANNELS, HEIGHT, WIDTH), dtype=np.
↪uint8)

    def step(self, action):
        ...
        return observation, reward, terminated, truncated, info

    def reset(self, seed=None, options=None):
        ...
        return observation, info

    def render(self):
        ...

    def close(self):
        ...
```

Then you can define and train a RL agent with:

```
# Instantiate the env
env = CustomEnv(arg1, ...)
# Define and Train the agent
model = A2C("CnnPolicy", env).learn(total_timesteps=1000)
```

To check that your environment follows the Gym interface that SB3 supports, please use:

```
from stable_baselines3.common.env_checker import check_env

env = CustomEnv(arg1, ...)
# It will check your custom environment and output additional warnings if needed
check_env(env)
```

Gymnasium also have its own env checker but it checks a superset of what SB3 supports (SB3 does not support all Gym features).

We have created a colab notebook for a concrete example of creating a custom environment along with an example of using it with Stable-Baselines3 interface.

Alternatively, you may look at Gymnasium built-in environments.

Optionally, you can also register the environment with gym, that will allow you to create the RL agent in one line (and use `gym.make()` to instantiate the env):

```python
from gymnasium.envs.registration import register
# Example for the CartPole environment
register(
    # unique identifier for the env `name-version`
    id="CartPole-v1",
    # path to the class for creating the env
    # Note: entry_point also accept a class as input (and not only a string)
    entry_point="gym.envs.classic_control:CartPoleEnv",
    # Max number of steps per episode, using a `TimeLimitWrapper`
    max_episode_steps=500,
)
```

In the project, for testing purposes, we use a custom environment named `IdentityEnv` defined in this file. An example of how to use it can be found here.

1.10 Callbacks

A callback is a set of functions that will be called at given stages of the training procedure. You can use callbacks to access internal state of the RL model during training. It allows one to do monitoring, auto saving, model manipulation, progress bars, …

1.10.1 Custom Callback

To build a custom callback, you need to create a class that derives from `BaseCallback`. This will give you access to events (`_on_training_start`, `_on_step`) and useful variables (like *self.model* for the RL model).

You can find two examples of custom callbacks in the documentation: one for saving the best model according to the training reward (see *Examples*), and one for logging additional values with Tensorboard (see *Tensorboard section*).

```python
from stable_baselines3.common.callbacks import BaseCallback


class CustomCallback(BaseCallback):
    """
    A custom callback that derives from ``BaseCallback``.

    :param verbose: Verbosity level: 0 for no output, 1 for info messages, 2 for debug messages
    """
    def __init__(self, verbose: int = 0):
        super().__init__(verbose)
        # Those variables will be accessible in the callback
        # (they are defined in the base class)
        # The RL model
        # self.model = None  # type: BaseAlgorithm
        # An alias for self.model.get_env(), the environment used for training
        # self.training_env # type: VecEnv
        # Number of time the callback was called
        # self.n_calls = 0  # type: int
```

(continues on next page)

(continued from previous page)

```python
        # num_timesteps = n_envs * n times env.step() was called
        # self.num_timesteps = 0  # type: int
        # local and global variables
        # self.locals = {}  # type: Dict[str, Any]
        # self.globals = {}  # type: Dict[str, Any]
        # The logger object, used to report things in the terminal
        # self.logger # type: stable_baselines3.common.logger.Logger
        # Sometimes, for event callback, it is useful
        # to have access to the parent object
        # self.parent = None  # type: Optional[BaseCallback]

    def _on_training_start(self) -> None:
        """
        This method is called before the first rollout starts.
        """
        pass

    def _on_rollout_start(self) -> None:
        """
        A rollout is the collection of environment interaction
        using the current policy.
        This event is triggered before collecting new samples.
        """
        pass

    def _on_step(self) -> bool:
        """
        This method will be called by the model after each call to `env.step()`.

        For child callback (of an `EventCallback`), this will be called
        when the event is triggered.

        :return: If the callback returns False, training is aborted early.
        """
        return True

    def _on_rollout_end(self) -> None:
        """
        This event is triggered before updating the policy.
        """
        pass

    def _on_training_end(self) -> None:
        """
        This event is triggered before exiting the `learn()` method.
        """
        pass
```

Note

`self.num_timesteps` corresponds to the total number of steps taken in the environment, i.e., it is the number of environments multiplied by the number of time `env.step()` was called

For the other algorithms, `self.num_timesteps` is incremented by `n_envs` (number of environments) after each call to `env.step()`

Note

For off-policy algorithms like SAC, DDPG, TD3 or DQN, the notion of `rollout` corresponds to the steps taken in the environment between two updates.

1.10.2 Event Callback

Compared to Keras, Stable Baselines provides a second type of `BaseCallback`, named `EventCallback` that is meant to trigger events. When an event is triggered, then a child callback is called.

As an example, *EvalCallback* is an `EventCallback` that will trigger its child callback when there is a new best model. A child callback is for instance *StopTrainingOnRewardThreshold* that stops the training if the mean reward achieved by the RL model is above a threshold.

Note

We recommend taking a look at the source code of *EvalCallback* and *StopTrainingOnRewardThreshold* to have a better overview of what can be achieved with this kind of callbacks.

```
class EventCallback(BaseCallback):
    """
    Base class for triggering callback on event.

    :param callback: Callback that will be called when an event is triggered.
    :param verbose: Verbosity level: 0 for no output, 1 for info messages, 2 for debug␣
↪messages
    """
    def __init__(self, callback: BaseCallback, verbose: int = 0):
        super().__init__(verbose=verbose)
        self.callback = callback
        # Give access to the parent
        self.callback.parent = self
    ...

    def _on_event(self) -> bool:
        return self.callback()
```

1.10.3 Callback Collection

Stable Baselines provides you with a set of common callbacks for:

- saving the model periodically (*CheckpointCallback*)
- evaluating the model periodically and saving the best one (*EvalCallback*)
- chaining callbacks (*CallbackList*)
- triggering callback on events (*Event Callback*, *EveryNTimesteps*)

- logging data every N timesteps (*LogEveryNTimesteps*)
- stopping the training early based on a reward threshold (*StopTrainingOnRewardThreshold*)

CheckpointCallback

Callback for saving a model every `save_freq` calls to `env.step()`, you must specify a log folder (`save_path`) and optionally a prefix for the checkpoints (`rl_model` by default). If you are using this callback to stop and resume training, you may want to optionally save the replay buffer if the model has one (`save_replay_buffer`, `False` by default). Additionally, if your environment uses a *VecNormalize* wrapper, you can save the corresponding statistics using `save_vecnormalize` (`False` by default).

Warning

When using multiple environments, each call to `env.step()` will effectively correspond to `n_envs` steps. If you want the `save_freq` to be similar when using a different number of environments, you need to account for it using `save_freq = max(save_freq // n_envs, 1)`. The same goes for the other callbacks.

```python
from stable_baselines3 import SAC
from stable_baselines3.common.callbacks import CheckpointCallback

# Save a checkpoint every 1000 steps
checkpoint_callback = CheckpointCallback(
  save_freq=1000,
  save_path="./logs/",
  name_prefix="rl_model",
  save_replay_buffer=True,
  save_vecnormalize=True,
)

model = SAC("MlpPolicy", "Pendulum-v1")
model.learn(2000, callback=checkpoint_callback)
```

EvalCallback

Evaluate periodically the performance of an agent, using a separate test environment. It will save the best model if `best_model_save_path` folder is specified and save the evaluations results in a NumPy archive (`evaluations.npz`) if `log_path` folder is specified.

Note

You can pass child callbacks via `callback_after_eval` and `callback_on_new_best` arguments. `callback_after_eval` will be triggered after every evaluation, and `callback_on_new_best` will be triggered each time there is a new best model.

Warning

You need to make sure that `eval_env` is wrapped the same way as the training environment, for instance using the `VecTransposeImage` wrapper if you have a channel-last image as input. The `EvalCallback` class outputs a warning if it is not the case.

```
import gymnasium as gym

from stable_baselines3 import SAC
from stable_baselines3.common.callbacks import EvalCallback

# Separate evaluation env
eval_env = gym.make("Pendulum-v1")
# Use deterministic actions for evaluation
eval_callback = EvalCallback(eval_env, best_model_save_path="./logs/",
                             log_path="./logs/", eval_freq=500,
                             deterministic=True, render=False)

model = SAC("MlpPolicy", "Pendulum-v1")
model.learn(5000, callback=eval_callback)
```

ProgressBarCallback

Display a progress bar with the current progress, elapsed time and estimated remaining time. This callback is integrated inside SB3 via the `progress_bar` argument of the `learn()` method.

> **Note**
>
> `ProgressBarCallback` callback requires `tqdm` and `rich` packages to be installed. This is done automatically when using `pip install stable-baselines3[extra]`

```
from stable_baselines3 import PPO
from stable_baselines3.common.callbacks import ProgressBarCallback

model = PPO("MlpPolicy", "Pendulum-v1")
# Display progress bar using the progress bar callback
# this is equivalent to model.learn(100_000, callback=ProgressBarCallback())
model.learn(100_000, progress_bar=True)
```

CallbackList

Class for chaining callbacks, they will be called sequentially. Alternatively, you can pass directly a list of callbacks to the `learn()` method, it will be converted automatically to a `CallbackList`.

```
import gymnasium as gym

from stable_baselines3 import SAC
from stable_baselines3.common.callbacks import CallbackList, CheckpointCallback,␣
↪EvalCallback

checkpoint_callback = CheckpointCallback(save_freq=1000, save_path="./logs/")
# Separate evaluation env
eval_env = gym.make("Pendulum-v1")
eval_callback = EvalCallback(eval_env, best_model_save_path="./logs/best_model",
                             log_path="./logs/results", eval_freq=500)
# Create the callback list
callback = CallbackList([checkpoint_callback, eval_callback])
```

(continues on next page)

(continued from previous page)

```
model = SAC("MlpPolicy", "Pendulum-v1")
# Equivalent to:
# model.learn(5000, callback=[checkpoint_callback, eval_callback])
model.learn(5000, callback=callback)
```

StopTrainingOnRewardThreshold

Stop the training once a threshold in episodic reward (mean episode reward over the evaluations) has been reached (i.e., when the model is good enough). It must be used with the *EvalCallback* and use the event triggered by a new best model.

```
import gymnasium as gym

from stable_baselines3 import SAC
from stable_baselines3.common.callbacks import EvalCallback,␣
↪StopTrainingOnRewardThreshold

# Separate evaluation env
eval_env = gym.make("Pendulum-v1")
# Stop training when the model reaches the reward threshold
callback_on_best = StopTrainingOnRewardThreshold(reward_threshold=-200, verbose=1)
eval_callback = EvalCallback(eval_env, callback_on_new_best=callback_on_best, verbose=1)

model = SAC("MlpPolicy", "Pendulum-v1", verbose=1)
# Almost infinite number of timesteps, but the training will stop
# early as soon as the reward threshold is reached
model.learn(int(1e10), callback=eval_callback)
```

EveryNTimesteps

An *Event Callback* that will trigger its child callback every `n_steps` timesteps.

Note

Because of the way `VecEnv` work, `n_steps` is a lower bound between two events when using multiple environments.

```
import gymnasium as gym

from stable_baselines3 import PPO
from stable_baselines3.common.callbacks import CheckpointCallback, EveryNTimesteps

# this is equivalent to defining CheckpointCallback(save_freq=500)
# checkpoint_callback will be triggered every 500 steps
checkpoint_on_event = CheckpointCallback(save_freq=1, save_path="./logs/")
event_callback = EveryNTimesteps(n_steps=500, callback=checkpoint_on_event)

model = PPO("MlpPolicy", "Pendulum-v1", verbose=1)

model.learn(20_000, callback=event_callback)
```

LogEveryNTimesteps

A callback derived from *EveryNTimesteps* that will dump the logged data every `n_steps` timesteps.

```python
import gymnasium as gym

from stable_baselines3 import PPO
from stable_baselines3.common.callbacks import LogEveryNTimesteps

event_callback = LogEveryNTimesteps(n_steps=1_000)

model = PPO("MlpPolicy", "Pendulum-v1", verbose=1)

# Disable auto-logging by passing `log_interval=None`
model.learn(10_000, callback=event_callback, log_interval=None)
```

StopTrainingOnMaxEpisodes

Stop the training upon reaching the maximum number of episodes, regardless of the model's `total_timesteps` value. Also, presumes that, for multiple environments, the desired behavior is that the agent trains on each env for `max_episodes` and in total for `max_episodes * n_envs` episodes.

Note

For multiple environments, the agent will train for a total of `max_episodes * n_envs` episodes. However, it can't be guaranteed that this training will occur for an exact number of `max_episodes` per environment. Thus, there is an assumption that, on average, each environment ran for `max_episodes`.

```python
from stable_baselines3 import A2C
from stable_baselines3.common.callbacks import StopTrainingOnMaxEpisodes

# Stops training when the model reaches the maximum number of episodes
callback_max_episodes = StopTrainingOnMaxEpisodes(max_episodes=5, verbose=1)

model = A2C("MlpPolicy", "Pendulum-v1", verbose=1)
# Almost infinite number of timesteps, but the training will stop
# early as soon as the max number of episodes is reached
model.learn(int(1e10), callback=callback_max_episodes)
```

StopTrainingOnNoModelImprovement

Stop the training if there is no new best model (no new best mean reward) after more than a specific number of consecutive evaluations. The idea is to save time in experiments when you know that the learning curves are somehow well-behaved and, therefore, after many evaluations without improvement the learning has probably stabilized. It must be used with the *EvalCallback* and use the event triggered after every evaluation.

```python
import gymnasium as gym

from stable_baselines3 import SAC
from stable_baselines3.common.callbacks import EvalCallback,␣
↪StopTrainingOnNoModelImprovement
```

(continues on next page)

(continued from previous page)

```
# Separate evaluation env
eval_env = gym.make("Pendulum-v1")
# Stop training if there is no improvement after more than 3 evaluations
stop_train_callback = StopTrainingOnNoModelImprovement(max_no_improvement_evals=3, min_
↪evals=5, verbose=1)
eval_callback = EvalCallback(eval_env, eval_freq=1000, callback_after_eval=stop_train_
↪callback, verbose=1)

model = SAC("MlpPolicy", "Pendulum-v1", learning_rate=1e-3, verbose=1)
# Almost infinite number of timesteps, but the training will stop early
# as soon as the number of consecutive evaluations without model
# improvement is greater than 3
model.learn(int(1e10), callback=eval_callback)
```

class `stable_baselines3.common.callbacks.`**`BaseCallback`**(*verbose=0*)

Base class for callback.

Parameters

`verbose` (*`int`*) – Verbosity level: 0 for no output, 1 for info messages, 2 for debug messages

`init_callback`(*model*)

Initialize the callback by saving references to the RL model and the training environment for convenience.

Parameters

`model` (`base_class.BaseAlgorithm`)

Return type

None

`on_step`()

This method will be called by the model after each call to `env.step()`.

For child callback (of an `EventCallback`), this will be called when the event is triggered.

Returns

If the callback returns False, training is aborted early.

Return type

bool

`update_child_locals`(*locals_*)

Update the references to the local variables on sub callbacks.

Parameters

- **`locals`** – the local variables during rollout collection
- **`locals_`** (*`dict[str, Any]`*)

Return type

None

`update_locals`(*locals_*)

Update the references to the local variables.

Parameters

- **`locals`** – the local variables during rollout collection
- **`locals_`** (*`dict[str, Any]`*)

Return type
None

`class stable_baselines3.common.callbacks.`**`CallbackList`**`(`*callbacks*`)`

Class for chaining callbacks.

Parameters
`callbacks` (*`list[BaseCallback]`*) – A list of callbacks that will be called sequentially.

`update_child_locals``(`*locals_*`)`

Update the references to the local variables.

Parameters

- **`locals`** – the local variables during rollout collection
- **`locals_`** (*`dict[str, Any]`*)

Return type
None

`class stable_baselines3.common.callbacks.`**`CheckpointCallback`**`(`*save_freq*, *save_path*, *name_prefix='rl_model'*, *save_replay_buffer=False*, *save_vecnormalize=False*, *verbose=0*`)`

Callback for saving a model every `save_freq` calls to `env.step()`. By default, it only saves model checkpoints, you need to pass `save_replay_buffer=True`, and `save_vecnormalize=True` to also save replay buffer checkpoints and normalization statistics checkpoints.

⚠ **Warning**

When using multiple environments, each call to `env.step()` will effectively correspond to `n_envs` steps. To account for that, you can use `save_freq = max(save_freq // n_envs, 1)`

Parameters

- **`save_freq`** (*`int`*) – Save checkpoints every `save_freq` call of the callback.
- **`save_path`** (*`str`*) – Path to the folder where the model will be saved.
- **`name_prefix`** (*`str`*) – Common prefix to the saved models
- **`save_replay_buffer`** (*`bool`*) – Save the model replay buffer
- **`save_vecnormalize`** (*`bool`*) – Save the `VecNormalize` statistics
- **`verbose`** (*`int`*) – Verbosity level: 0 for no output, 2 for indicating when saving model checkpoint

`class stable_baselines3.common.callbacks.`**`ConvertCallback`**`(`*callback*, *verbose=0*`)`

Convert functional callback (old-style) to object.

Parameters

- **`callback`** (*`Callable[[dict[str, Any], dict[str, Any]], bool] | None`*)
- **`verbose`** (*`int`*) – Verbosity level: 0 for no output, 1 for info messages, 2 for debug messages

class stable_baselines3.common.callbacks.**EvalCallback**(*eval_env*, *callback_on_new_best=None*, *callback_after_eval=None*, *n_eval_episodes=5*, *eval_freq=10000*, *log_path=None*, *best_model_save_path=None*, *deterministic=True*, *render=False*, *verbose=1*, *warn=True*)

Callback for evaluating an agent.

> **Warning**
>
> When using multiple environments, each call to `env.step()` will effectively correspond to `n_envs` steps. To account for that, you can use `eval_freq = max(eval_freq // n_envs, 1)`

Parameters

- **eval_env** (*Env | VecEnv*) – The environment used for initialization
- **callback_on_new_best** (*BaseCallback | None*) – Callback to trigger when there is a new best model according to the `mean_reward`
- **callback_after_eval** (*BaseCallback | None*) – Callback to trigger after every evaluation
- **n_eval_episodes** (*int*) – The number of episodes to test the agent
- **eval_freq** (*int*) – Evaluate the agent every `eval_freq` call of the callback.
- **log_path** (*str | None*) – Path to a folder where the evaluations (`evaluations.npz`) will be saved. It will be updated at each evaluation.
- **best_model_save_path** (*str | None*) – Path to a folder where the best model according to performance on the eval env will be saved.
- **deterministic** (*bool*) – Whether the evaluation should use a stochastic or deterministic actions.
- **render** (*bool*) – Whether to render or not the environment during evaluation
- **verbose** (*int*) – Verbosity level: 0 for no output, 1 for indicating information about evaluation results
- **warn** (*bool*) – Passed to `evaluate_policy` (warns if `eval_env` has not been wrapped with a Monitor wrapper)

update_child_locals(*locals_*)

Update the references to the local variables.

Parameters

- **locals** – the local variables during rollout collection
- **locals_** (*dict[str, Any]*)

Return type

None

class stable_baselines3.common.callbacks.**EventCallback**(*callback=None*, *verbose=0*)

Base class for triggering callback on event.

Parameters

- **callback** (BaseCallback | None) – Callback that will be called when an event is triggered.
- **verbose** (*int*) – Verbosity level: 0 for no output, 1 for info messages, 2 for debug messages

init_callback(*model*)

Initialize the callback by saving references to the RL model and the training environment for convenience.

Parameters
model (base_class.BaseAlgorithm)

Return type
None

update_child_locals(*locals_*)

Update the references to the local variables.

Parameters

- **locals** – the local variables during rollout collection
- **locals_** (*dict[str, Any]*)

Return type
None

class stable_baselines3.common.callbacks.**EveryNTimesteps**(*n_steps*, *callback*)

Trigger a callback every `n_steps` timesteps

Parameters

- **n_steps** (*int*) – Number of timesteps between two trigger.
- **callback** (BaseCallback) – Callback that will be called when the event is triggered.

class stable_baselines3.common.callbacks.**LogEveryNTimesteps**(*n_steps*)

Log data every `n_steps` timesteps

Parameters
n_steps (*int*) – Number of timesteps between two trigger.

class stable_baselines3.common.callbacks.**ProgressBarCallback**

Display a progress bar when training SB3 agent using tqdm and rich packages.

class stable_baselines3.common.callbacks.**StopTrainingOnMaxEpisodes**(*max_episodes*, *verbose=0*)

Stop the training once a maximum number of episodes are played.

For multiple environments presumes that, the desired behavior is that the agent trains on each env for `max_episodes` and in total for `max_episodes * n_envs` episodes.

Parameters

- **max_episodes** (*int*) – Maximum number of episodes to stop training.
- **verbose** (*int*) – Verbosity level: 0 for no output, 1 for indicating information about when training ended by reaching `max_episodes`

class stable_baselines3.common.callbacks.**StopTrainingOnNoModelImprovement**(*max_no_improvement_evals*, *min_evals=0*, *verbose=0*)

Stop the training early if there is no new best model (new best mean reward) after more than N consecutive evaluations.

It is possible to define a minimum number of evaluations before start to count evaluations without improvement.

It must be used with the `EvalCallback`.

Parameters

- **max_no_improvement_evals** (*int*) – Maximum number of consecutive evaluations without a new best model.
- **min_evals** (*int*) – Number of evaluations before start to count evaluations without improvements.
- **verbose** (*int*) – Verbosity level: 0 for no output, 1 for indicating when training ended because no new best model

class stable_baselines3.common.callbacks.**StopTrainingOnRewardThreshold**(*reward_threshold*, *verbose=0*)

Stop the training once a threshold in episodic reward has been reached (i.e. when the model is good enough).

It must be used with the `EvalCallback`.

Parameters

- **reward_threshold** (*float*) – Minimum expected reward per episode to stop training.
- **verbose** (*int*) – Verbosity level: 0 for no output, 1 for indicating when training ended because episodic reward threshold reached

1.11 Tensorboard Integration

1.11.1 Basic Usage

To use Tensorboard with stable baselines3, you simply need to pass the location of the log folder to the RL agent:

```python
from stable_baselines3 import A2C

model = A2C("MlpPolicy", "CartPole-v1", verbose=1, tensorboard_log="./a2c_cartpole_
↪tensorboard/")
model.learn(total_timesteps=10_000)
```

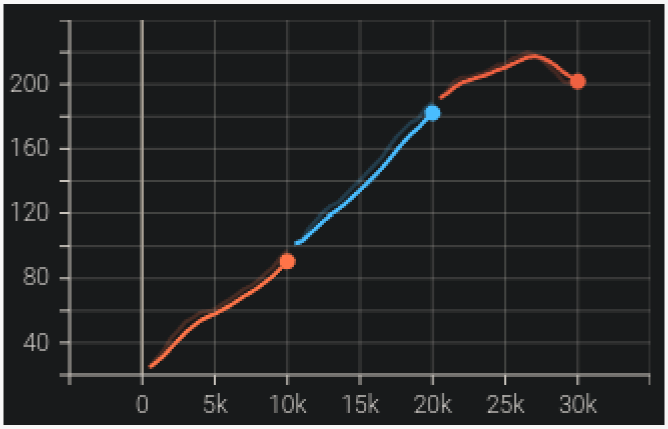
You can also define custom logging name when training (by default it is the algorithm name)

```python
from stable_baselines3 import A2C

model = A2C("MlpPolicy", "CartPole-v1", verbose=1, tensorboard_log="./a2c_cartpole_
↪tensorboard/")
model.learn(total_timesteps=10_000, tb_log_name="first_run")
# Pass reset_num_timesteps=False to continue the training curve in tensorboard
# By default, it will create a new curve
# Keep tb_log_name constant to have continuous curve (see note below)
model.learn(total_timesteps=10_000, tb_log_name="second_run", reset_num_timesteps=False)
model.learn(total_timesteps=10_000, tb_log_name="third_run", reset_num_timesteps=False)
```

Note

If you specify different `tb_log_name` in subsequent runs, you will have split graphs, like in the figure below. If you want them to be continuous, you must keep the same `tb_log_name` (see issue #975). And, if you still managed to get your graphs split by other means, just put tensorboard log files into the same folder.



Once the learn function is called, you can monitor the RL agent during or after the training, with the following bash command:

```
tensorboard --logdir ./a2c_cartpole_tensorboard/
```

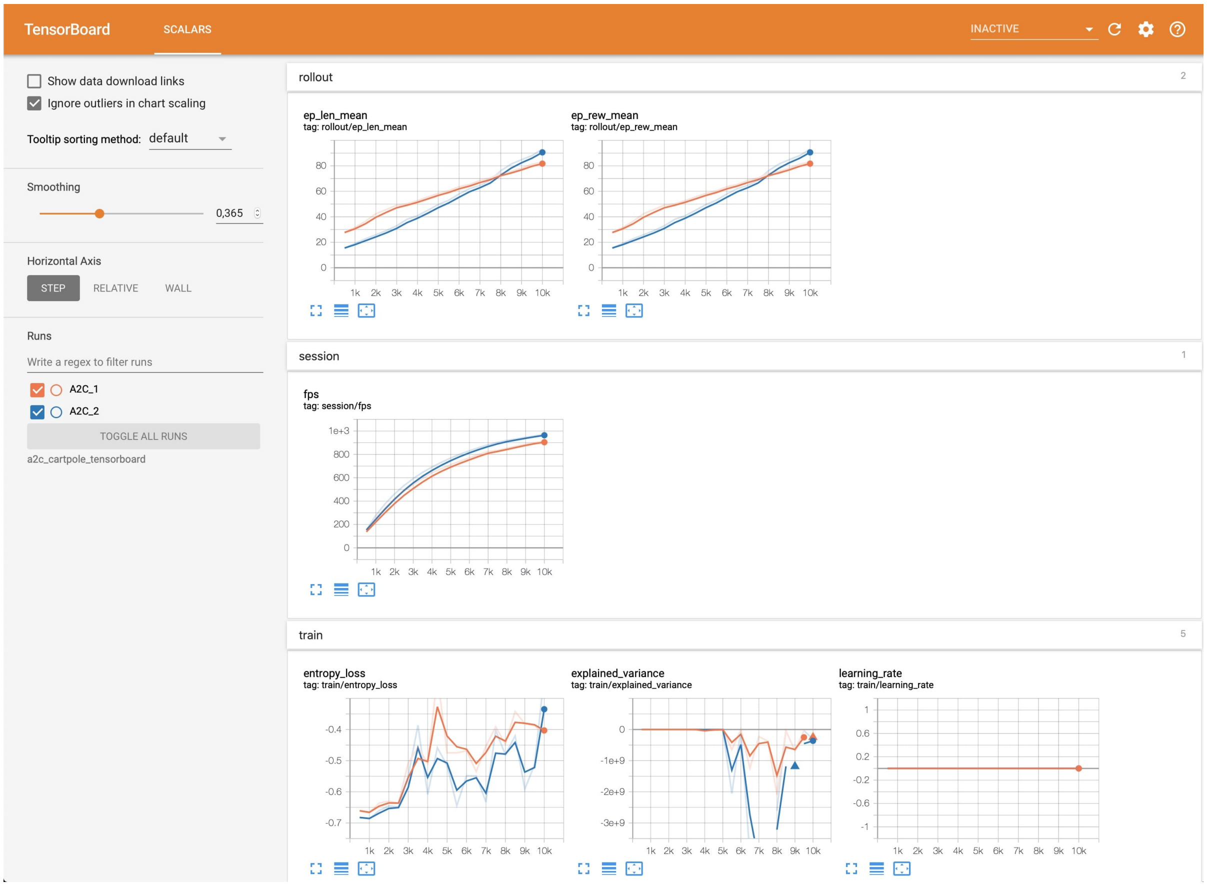
Note

You can find explanations about the logger output and names in the *Logger* section.

you can also add past logging folders:

```
tensorboard --logdir ./a2c_cartpole_tensorboard/;./ppo2_cartpole_tensorboard/
```

It will display information such as the episode reward (when using a `Monitor` wrapper), the model losses and other parameter unique to some models.



1.11.2 Logging More Values

Using a callback, you can easily log more values with TensorBoard. Here is a simple example on how to log both additional tensor or arbitrary scalar value:

```python
import numpy as np

from stable_baselines3 import SAC
from stable_baselines3.common.callbacks import BaseCallback

model = SAC("MlpPolicy", "Pendulum-v1", tensorboard_log="/tmp/sac/", verbose=1)


class TensorboardCallback(BaseCallback):
    """
    Custom callback for plotting additional values in tensorboard.
    """

    def __init__(self, verbose=0):
        super().__init__(verbose)

    def _on_step(self) -> bool:
        # Log scalar value (here a random variable)
        value = np.random.random()
        self.logger.record("random_value", value)
```

(continues on next page)

(continued from previous page)

```
        return True


model.learn(50000, callback=TensorboardCallback())
```

Note

If you want to log values more often than the default to tensorboard, you manually call `self.logger.dump(self.num_timesteps)` in a callback (see issue #506).

1.11.3 Logging Images

TensorBoard supports periodic logging of image data, which helps evaluating agents at various stages during training.

Warning

To support image logging pillow must be installed otherwise, TensorBoard ignores the image and logs a warning.

Here is an example of how to render an image to TensorBoard at regular intervals:

```
from stable_baselines3 import SAC
from stable_baselines3.common.callbacks import BaseCallback
from stable_baselines3.common.logger import Image

model = SAC("MlpPolicy", "Pendulum-v1", tensorboard_log="/tmp/sac/", verbose=1)


class ImageRecorderCallback(BaseCallback):
    def __init__(self, verbose=0):
        super().__init__(verbose)

    def _on_step(self):
        image = self.training_env.render(mode="rgb_array")
        # "HWC" specify the dataformat of the image, here channel last
        # (H for height, W for width, C for channel)
        # See https://pytorch.org/docs/stable/tensorboard.html
        # for supported formats
        self.logger.record("trajectory/image", Image(image, "HWC"), exclude=("stdout",
↪"log", "json", "csv"))
        return True


model.learn(50000, callback=ImageRecorderCallback())
```

1.11.4 Logging Figures/Plots

TensorBoard supports periodic logging of figures/plots created with matplotlib, which helps evaluate agents at various stages during training.

Warning

To support figure logging matplotlib must be installed otherwise, TensorBoard ignores the figure and logs a warning.

Here is an example of how to store a plot in TensorBoard at regular intervals:

```
import numpy as np
import matplotlib.pyplot as plt

from stable_baselines3 import SAC
from stable_baselines3.common.callbacks import BaseCallback
from stable_baselines3.common.logger import Figure

model = SAC("MlpPolicy", "Pendulum-v1", tensorboard_log="/tmp/sac/", verbose=1)


class FigureRecorderCallback(BaseCallback):
    def __init__(self, verbose=0):
        super().__init__(verbose)

    def _on_step(self):
        # Plot values (here a random variable)
        figure = plt.figure()
        figure.add_subplot().plot(np.random.random(3))
        # Close the figure after logging it
        self.logger.record("trajectory/figure", Figure(figure, close=True), exclude=(
↪"stdout", "log", "json", "csv"))
        plt.close()
        return True


model.learn(50000, callback=FigureRecorderCallback())
```

1.11.5 Logging Videos

TensorBoard supports periodic logging of video data, which helps evaluate agents at various stages during training.

Warning

To support video logging moviepy must be installed otherwise, TensorBoard ignores the video and logs a warning.

Here is an example of how to render an episode and log the resulting video to TensorBoard at regular intervals:

```
from typing import Any, Dict

import gymnasium as gym
import torch as th
import numpy as np

from stable_baselines3 import A2C
from stable_baselines3.common.callbacks import BaseCallback
```

(continues on next page)

(continued from previous page)

```
from stable_baselines3.common.evaluation import evaluate_policy
from stable_baselines3.common.logger import Video


class VideoRecorderCallback(BaseCallback):
    def __init__(self, eval_env: gym.Env, render_freq: int, n_eval_episodes: int = 1,␣
↪deterministic: bool = True):
        """
        Records a video of an agent's trajectory traversing ``eval_env`` and logs it to␣
↪TensorBoard

        :param eval_env: A gym environment from which the trajectory is recorded
        :param render_freq: Render the agent's trajectory every eval_freq call of the␣
↪callback.
        :param n_eval_episodes: Number of episodes to render
        :param deterministic: Whether to use deterministic or stochastic policy
        """
        super().__init__()
        self._eval_env = eval_env
        self._render_freq = render_freq
        self._n_eval_episodes = n_eval_episodes
        self._deterministic = deterministic

    def _on_step(self) -> bool:
        if self.n_calls % self._render_freq == 0:
            screens = []

            def grab_screens(_locals: Dict[str, Any], _globals: Dict[str, Any]) -> None:
                """
                Renders the environment in its current state, recording the screen in␣
↪the captured `screens` list

                :param _locals: A dictionary containing all local variables of the␣
↪callback's scope
                :param _globals: A dictionary containing all global variables of the␣
↪callback's scope
                """
                # We expect `render()` to return a uint8 array with values in [0, 255] or␣
↪a float array
                # with values in [0, 1], as described in
                # https://pytorch.org/docs/stable/tensorboard.html#torch.utils.
↪tensorboard.writer.SummaryWriter.add_video
                screen = self._eval_env.render(mode="rgb_array")
                # PyTorch uses CxHxW vs HxWxC gym (and tensorflow) image convention
                screens.append(screen.transpose(2, 0, 1))

            evaluate_policy(
                self.model,
                self._eval_env,
                callback=grab_screens,
                n_eval_episodes=self._n_eval_episodes,
                deterministic=self._deterministic,
```

(continues on next page)

(continued from previous page)

```
            )
            self.logger.record(
                "trajectory/video",
                Video(th.from_numpy(np.asarray([screens])), fps=40),
                exclude=("stdout", "log", "json", "csv"),
            )
        return True


model = A2C("MlpPolicy", "CartPole-v1", tensorboard_log="runs/", verbose=1)
video_recorder = VideoRecorderCallback(gym.make("CartPole-v1"), render_freq=5000)
model.learn(total_timesteps=int(5e4), callback=video_recorder)
```

1.11.6 Logging Hyperparameters

TensorBoard supports logging of hyperparameters in its HPARAMS tab, which helps to compare agents trainings.

> **Warning**
>
> To display hyperparameters in the HPARAMS section, a `metric_dict` must be given (as well as a `hparam_dict`).

Here is an example of how to save hyperparameters in TensorBoard:

```
from stable_baselines3 import A2C
from stable_baselines3.common.callbacks import BaseCallback
from stable_baselines3.common.logger import HParam


class HParamCallback(BaseCallback):
    """
    Saves the hyperparameters and metrics at the start of the training, and logs them to␣
↪TensorBoard.
    """

    def _on_training_start(self) -> None:
        hparam_dict = {
            "algorithm": self.model.__class__.__name__,
            "learning rate": self.model.learning_rate,
            "gamma": self.model.gamma,
        }
        # define the metrics that will appear in the `HPARAMS` Tensorboard tab by␣
↪referencing their tag
        # Tensorbaord will find & display metrics from the `SCALARS` tab
        metric_dict = {
            "rollout/ep_len_mean": 0,
            "train/value_loss": 0.0,
        }
        self.logger.record(
            "hparams",
            HParam(hparam_dict, metric_dict),
            exclude=("stdout", "log", "json", "csv"),
```

(continues on next page)

(continued from previous page)

```
        )

    def _on_step(self) -> bool:
        return True


model = A2C("MlpPolicy", "CartPole-v1", tensorboard_log="runs/", verbose=1)
model.learn(total_timesteps=int(5e4), callback=HParamCallback())
```

1.11.7 Directly Accessing The Summary Writer

If you would like to log arbitrary data (in one of the formats supported by pytorch), you can get direct access to the underlying SummaryWriter in a callback:

> **Warning**
>
> This is method is not recommended and should only be used by advanced users.

> **Note**
>
> If you want a concrete example, you can watch how to log lap time with donkeycar env, or read the code in the RL Zoo. You might also want to take a look at issue #1160 and issue #1219.

```
from stable_baselines3 import SAC
from stable_baselines3.common.callbacks import BaseCallback
from stable_baselines3.common.logger import TensorBoardOutputFormat



model = SAC("MlpPolicy", "Pendulum-v1", tensorboard_log="/tmp/sac/", verbose=1)


class SummaryWriterCallback(BaseCallback):

    def _on_training_start(self):
        self._log_freq = 1000  # log every 1000 calls

        output_formats = self.logger.output_formats
        # Save reference to tensorboard formatter object
        # note: the failure case (not formatter found) is not handled here, should be␣
↪done with try/except.
        self.tb_formatter = next(formatter for formatter in output_formats if␣
↪isinstance(formatter, TensorBoardOutputFormat))

    def _on_step(self) -> bool:
        if self.n_calls % self._log_freq == 0:
            # You can have access to info from the env using self.locals.
            # for instance, when using one env (index 0 of locals["infos"]):
            # lap_count = self.locals["infos"][0]["lap_count"]
```

(continues on next page)

(continued from previous page)

```
            # self.tb_formatter.writer.add_scalar("train/lap_count", lap_count, self.num_
↪timesteps)

            self.tb_formatter.writer.add_text("direct_access", "this is a value", self.
↪num_timesteps)
            self.tb_formatter.writer.flush()


model.learn(50000, callback=SummaryWriterCallback())
```

1.12 Integrations

1.12.1 Weights & Biases

Weights & Biases provides a callback for experiment tracking that allows to visualize and share results.

The full documentation is available here: https://docs.wandb.ai/guides/integrations/other/stable-baselines-3

```
import gymnasium as gym
import wandb
from wandb.integration.sb3 import WandbCallback

from stable_baselines3 import PPO

config = {
    "policy_type": "MlpPolicy",
    "total_timesteps": 25000,
    "env_id": "CartPole-v1",
}
run = wandb.init(
    project="sb3",
    config=config,
    sync_tensorboard=True,  # auto-upload sb3's tensorboard metrics
    # monitor_gym=True,  # auto-upload the videos of agents playing the game
    # save_code=True,  # optional
)

model = PPO(config["policy_type"], config["env_id"], verbose=1, tensorboard_log=f"runs/
↪{run.id}")
model.learn(
    total_timesteps=config["total_timesteps"],
    callback=WandbCallback(
        model_save_path=f"models/{run.id}",
        verbose=2,
    ),
)
run.finish()
```

1.12.2 Hugging Face

The Hugging Face Hub is a central place where anyone can share and explore models. It allows you to host your saved models .

You can see the list of stable-baselines3 saved models here: https://huggingface.co/models?library=stable-baselines3 Most of them are available via the RL Zoo.

Official pre-trained models are saved in the SB3 organization on the hub: https://huggingface.co/sb3

We wrote a tutorial on how to use Hub and Stable-Baselines3 here.

Installation

```
pip install huggingface_sb3
```

Note

If you use the RL Zoo, pushing/loading models from the hub are already integrated:

```
# Download model and save it into the logs/ folder
# Only use TRUST_REMOTE_CODE=True with HF models that can be trusted (here the SB3␣
↪organization)
TRUST_REMOTE_CODE=True python -m rl_zoo3.load_from_hub --algo a2c --env LunarLander-
↪v2 -orga sb3 -f logs/
# Test the agent
python -m rl_zoo3.enjoy --algo a2c --env LunarLander-v2  -f logs/
# Push model, config and hyperparameters to the hub
python -m rl_zoo3.push_to_hub --algo a2c --env LunarLander-v2 -f logs/ -orga sb3 -m
↪"Initial commit"
```

Download a model from the Hub

You need to copy the repo-id that contains your saved model. For instance `sb3/demo-hf-CartPole-v1`:

```
import os

import gymnasium as gym

from huggingface_sb3 import load_from_hub
from stable_baselines3 import PPO
from stable_baselines3.common.evaluation import evaluate_policy


# Allow the use of `pickle.load()` when downloading model from the hub
# Please make sure that the organization from which you download can be trusted
os.environ["TRUST_REMOTE_CODE"] = "True"

# Retrieve the model from the hub
## repo_id = id of the model repository from the Hugging Face Hub (repo_id =
↪{organization}/{repo_name})
## filename = name of the model zip file from the repository
checkpoint = load_from_hub(
    repo_id="sb3/demo-hf-CartPole-v1",
```

(continues on next page)

(continued from previous page)

```
    filename="ppo-CartPole-v1.zip",
)
model = PPO.load(checkpoint)

# Evaluate the agent and watch it
eval_env = gym.make("CartPole-v1")
mean_reward, std_reward = evaluate_policy(
    model, eval_env, render=True, n_eval_episodes=5, deterministic=True, warn=False
)
print(f"mean_reward={mean_reward:.2f} +/- {std_reward}")
```

You need to define two parameters:

- `repo-id`: the name of the Hugging Face repo you want to download.
- `filename`: the file you want to download.

Upload a model to the Hub

You can easily upload your models using two different functions:

1. `package_to_hub()`: save the model, evaluate it, generate a model card and record a replay video of your agent before pushing the complete repo to the Hub.
2. `push_to_hub()`: simply push a file to the Hub.

First, you need to be logged in to Hugging Face to upload a model:

- If you're using Colab/Jupyter Notebooks:

```
from huggingface_hub import notebook_login
notebook_login()
```

- Otherwise:

```
huggingface-cli login
```

Then, in this example, we train a PPO agent to play CartPole-v1 and push it to a new repo `sb3/demo-hf-CartPole-v1`

With `package_to_hub()`

```
from stable_baselines3 import PPO
from stable_baselines3.common.env_util import make_vec_env

from huggingface_sb3 import package_to_hub

# Create the environment
env_id = "CartPole-v1"
env = make_vec_env(env_id, n_envs=1)

# Create the evaluation environment
eval_env = make_vec_env(env_id, n_envs=1)

# Instantiate the agent
model = PPO("MlpPolicy", env, verbose=1)
```

(continues on next page)

(continued from previous page)

```python

# Train the agent
model.learn(total_timesteps=int(5000))

# This method saves, evaluates, generates a model card and records a replay video of your agent before pushing the repo to the hub
package_to_hub(model=model,
             model_name="ppo-CartPole-v1",
             model_architecture="PPO",
             env_id=env_id,
             eval_env=eval_env,
             repo_id="sb3/demo-hf-CartPole-v1",
             commit_message="Test commit")
```

You need to define seven parameters:

- `model`: your trained model.
- `model_architecture`: name of the architecture of your model (DQN, PPO, A2C, SAC…).
- `env_id`: name of the environment.
- `eval_env`: environment used to evaluate the agent.
- `repo-id`: the name of the Hugging Face repo you want to create or update. It's <your huggingface username>/<the repo name>.
- `commit-message`.
- `filename`: the file you want to push to the Hub.

With `push_to_hub()`

```python
from stable_baselines3 import PPO
from stable_baselines3.common.env_util import make_vec_env

from huggingface_sb3 import push_to_hub

# Create the environment
env_id = "CartPole-v1"
env = make_vec_env(env_id, n_envs=1)

# Instantiate the agent
model = PPO("MlpPolicy", env, verbose=1)

# Train the agent
model.learn(total_timesteps=int(5000))

# Save the model
model.save("ppo-CartPole-v1")

# Push this saved model .zip file to the hf repo
# If this repo does not exist it will be created
## repo_id = id of the model repository from the Hugging Face Hub (repo_id = {organization}/{repo_name})
```

(continues on next page)

(continued from previous page)

```
## filename: the name of the file == "name" inside model.save("ppo-CartPole-v1")
push_to_hub(
  repo_id="sb3/demo-hf-CartPole-v1",
  filename="ppo-CartPole-v1.zip",
  commit_message="Added CartPole-v1 model trained with PPO",
)
```

You need to define three parameters:

- `repo-id`: the name of the Hugging Face repo you want to create or update. It's <your huggingface username>/<the repo name>.
- `filename`: the file you want to push to the Hub.
- `commit-message`.

1.12.3 MLFLow

If you want to use MLFLow to track your SB3 experiments, you can adapt the following code which defines a custom logger output:

```
import sys
from typing import Any, Dict, Tuple, Union

import mlflow
import numpy as np

from stable_baselines3 import SAC
from stable_baselines3.common.logger import HumanOutputFormat, KVWriter, Logger


class MLflowOutputFormat(KVWriter):
    """
    Dumps key/value pairs into MLflow's numeric format.
    """

    def write(
        self,
        key_values: Dict[str, Any],
        key_excluded: Dict[str, Union[str, Tuple[str, ...]]],
        step: int = 0,
    ) -> None:

        for (key, value), (_, excluded) in zip(
            sorted(key_values.items()), sorted(key_excluded.items())
        ):

            if excluded is not None and "mlflow" in excluded:
                continue

            if isinstance(value, np.ScalarType):
                if not isinstance(value, str):
                    mlflow.log_metric(key, value, step)
```

(continues on next page)

(continued from previous page)

```
loggers = Logger(
    folder=None,
    output_formats=[HumanOutputFormat(sys.stdout), MLflowOutputFormat()],
)

with mlflow.start_run():
    model = SAC("MlpPolicy", "Pendulum-v1", verbose=2)
    # Set custom logger
    model.set_logger(loggers)
    model.learn(total_timesteps=10000, log_interval=1)
```

1.13 RL Baselines3 Zoo

RL Baselines3 Zoo is a training framework for Reinforcement Learning (RL).

It provides scripts for training, evaluating agents, tuning hyperparameters, plotting results and recording videos.

In addition, it includes a collection of tuned hyperparameters for common environments and RL algorithms, and agents trained with those settings.

Goals of this repository:

1. Provide a simple interface to train and enjoy RL agents
2. Benchmark the different Reinforcement Learning algorithms
3. Provide tuned hyperparameters for each environment and RL algorithm
4. Have fun with the trained agents!

Documentation is available online: https://rl-baselines3-zoo.readthedocs.io/

1.13.1 Installation

Option 1: install the python package `pip install rl_zoo3`

or:

1. Clone the repository:

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

Note

You can remove the `--recursive` option if you don't want to download the trained agents

Note

If you only need the training/plotting scripts and additional callbacks/wrappers from the RL Zoo, you can also install it via pip: `pip install rl_zoo3`

2. Install dependencies

```
apt-get install swig cmake ffmpeg
# full dependencies
pip install -r requirements.txt
# minimal dependencies
pip install -e .
```

1.13.2 Train an Agent

The hyperparameters for each environment are defined in `hyperparameters/algo_name.yml`.

If the environment exists in this file, then you can train an agent using:

```
python -m rl_zoo3.train --algo algo_name --env env_id
```

For example (with evaluation and checkpoints):

```
python -m rl_zoo3.train --algo ppo --env CartPole-v1 --eval-freq 10000 --save-freq 50000
```

Continue training (here, load pretrained agent for Breakout and continue training for 5000 steps):

```
python -m rl_zoo3.train --algo a2c --env BreakoutNoFrameskip-v4 -i trained_agents/a2c/
↪BreakoutNoFrameskip-v4_1/BreakoutNoFrameskip-v4.zip -n 5000
```

1.13.3 Enjoy a Trained Agent

If the trained agent exists, then you can see it in action using:

```
python -m rl_zoo3.enjoy --algo algo_name --env env_id
```

For example, enjoy A2C on Breakout during 5000 timesteps:

```
python -m rl_zoo3.enjoy --algo a2c --env BreakoutNoFrameskip-v4 --folder rl-trained-
↪agents/ -n 5000
```

1.13.4 Hyperparameter Optimization

We use Optuna for optimizing the hyperparameters.

Tune the hyperparameters for PPO, using a random sampler and median pruner, 2 parallels jobs, with a budget of 1000 trials and a maximum of 50000 steps:

```
python -m rl_zoo3.train --algo ppo --env MountainCar-v0 -n 50000 -optimize --n-trials␣
↪1000 --n-jobs 2 \
  --sampler random --pruner median
```

1.13.5 Colab Notebook: Try it Online!

You can train agents online using Google colab notebook.

Note

You can find more information about the rl baselines3 zoo in the repo README. For instance, how to record a video of a trained agent.

1.14 SB3 Contrib

We implement experimental features in a separate contrib repository: SB3-Contrib

This allows Stable-Baselines3 (SB3) to maintain a stable and compact core, while still providing the latest features, like RecurrentPPO (PPO LSTM), Truncated Quantile Critics (TQC), Augmented Random Search (ARS), Trust Region Policy Optimization (TRPO) or Quantile Regression DQN (QR-DQN).

1.14.1 Why create this repository?

Over the span of stable-baselines and stable-baselines3, the community has been eager to contribute in form of better logging utilities, environment wrappers, extended support (e.g. different action spaces) and learning algorithms.

However sometimes these utilities were too niche to be considered for stable-baselines or proved to be too difficult to integrate well into the existing code without creating a mess. sb3-contrib aims to fix this by not requiring the neatest code integration with existing code and not setting limits on what is too niche: almost everything remotely useful goes! We hope this allows us to provide reliable implementations following stable-baselines usual standards (consistent style, documentation, etc) beyond the relatively small scope of utilities in the main repository.

Features

See documentation for the full list of included features.

RL Algorithms:

- Augmented Random Search (ARS)
- Quantile Regression DQN (QR-DQN)
- PPO with invalid action masking (Maskable PPO)
- PPO with recurrent policy (RecurrentPPO aka PPO LSTM)
- Truncated Quantile Critics (TQC)
- Trust Region Policy Optimization (TRPO)
- Batch Normalization in Deep Reinforcement Learning (CrossQ)

Gym Wrappers:

- Time Feature Wrapper

Documentation

Documentation is available online: https://sb3-contrib.readthedocs.io/

Installation

To install Stable-Baselines3 contrib with pip, execute:

```
pip install sb3-contrib
```

We recommend to use the `master` version of Stable Baselines3 and SB3-Contrib.

To install Stable Baselines3 `master` version:

```
pip install git+https://github.com/DLR-RM/stable-baselines3
```

To install Stable Baselines3 contrib `master` version:

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

Example

SB3-Contrib follows the SB3 API and folder structure. So, if you are familiar with SB3, using SB3-Contrib should be easy too.

Here is an example of training 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=10000, log_interval=4)
model.save("qrdqn_cartpole")
```

1.15 Stable Baselines Jax (SBX)

Stable Baselines Jax (SBX) is a proof of concept version of Stable-Baselines3 in Jax.

It provides a minimal number of features compared to SB3 but can be much faster (up to 20x times!): https://twitter.com/araffin2/status/1590714558628253698

Implemented algorithms:

- Soft Actor-Critic (SAC) and SAC-N
- Truncated Quantile Critics (TQC)
- Dropout Q-Functions for Doubly Efficient Reinforcement Learning (DroQ)
- Proximal Policy Optimization (PPO)
- Deep Q Network (DQN)
- Twin Delayed DDPG (TD3)
- Deep Deterministic Policy Gradient (DDPG)
- Batch Normalization in Deep Reinforcement Learning (CrossQ)
- Simplicity Bias for Scaling Up Parameters in Deep Reinforcement Learning (SimBa)

As SBX follows SB3 API, it is also compatible with the RL Zoo. For that you will need to create two files:

`train_sbx.py`:

```
import rl_zoo3
import rl_zoo3.train
from rl_zoo3.train import train
from sbx import DDPG, DQN, PPO, SAC, TD3, TQC, CrossQ

rl_zoo3.ALGOS["ddpg"] = DDPG
rl_zoo3.ALGOS["dqn"] = DQN
# See SBX readme to use DroQ configuration
# rl_zoo3.ALGOS["droq"] = DroQ
rl_zoo3.ALGOS["sac"] = SAC
rl_zoo3.ALGOS["ppo"] = PPO
rl_zoo3.ALGOS["td3"] = TD3
```

(continues on next page)

(continued from previous page)

```python
rl_zoo3.ALGOS["tqc"] = TQC
rl_zoo3.ALGOS["crossq"] = CrossQ
rl_zoo3.train.ALGOS = rl_zoo3.ALGOS
rl_zoo3.exp_manager.ALGOS = rl_zoo3.ALGOS


if __name__ == "__main__":
    train()
```

Then you can call `python train_sbx.py --algo sac --env Pendulum-v1` and use the RL Zoo CLI.

`enjoy_sbx.py`:

```python
import rl_zoo3
import rl_zoo3.enjoy
from rl_zoo3.enjoy import enjoy
from sbx import DDPG, DQN, PPO, SAC, TD3, TQC, CrossQ

rl_zoo3.ALGOS["ddpg"] = DDPG
rl_zoo3.ALGOS["dqn"] = DQN
# See SBX readme to use DroQ configuration
# rl_zoo3.ALGOS["droq"] = DroQ
rl_zoo3.ALGOS["sac"] = SAC
rl_zoo3.ALGOS["ppo"] = PPO
rl_zoo3.ALGOS["td3"] = TD3
rl_zoo3.ALGOS["tqc"] = TQC
rl_zoo3.ALGOS["crossq"] = CrossQ
rl_zoo3.enjoy.ALGOS = rl_zoo3.ALGOS
rl_zoo3.exp_manager.ALGOS = rl_zoo3.ALGOS


if __name__ == "__main__":
    enjoy()
```

1.16 Imitation Learning

The imitation library implements imitation learning algorithms on top of Stable-Baselines3, including:

- Behavioral Cloning
- DAgger with synthetic examples
- Adversarial Inverse Reinforcement Learning (AIRL)
- Generative Adversarial Imitation Learning (GAIL)
- Deep RL from Human Preferences (DRLHP)

You can install imitation with `pip install imitation`. The imitation documentation has more details on how to use the library, including a quick start guide for the impatient.

1.17 Migrating from Stable-Baselines

This is a guide to migrate from Stable-Baselines (SB2) to Stable-Baselines3 (SB3).

It also references the main changes.

1.17.1 Overview

Overall Stable-Baselines3 (SB3) keeps the high-level API of Stable-Baselines (SB2). Most of the changes are to ensure more consistency and are internal ones. Because of the backend change, from Tensorflow to PyTorch, the internal code is much more readable and easy to debug at the cost of some speed (dynamic graph vs static graph., see Issue #90) However, the algorithms were extensively benchmarked on Atari games and continuous control PyBullet envs (see Issue #48 and Issue #49) so you should not expect performance drop when switching from SB2 to SB3.

1.17.2 How to migrate?

In most cases, replacing `from stable_baselines` with `from stable_baselines3` will be sufficient. Some files were moved to the common folder (cf below) and could result to import errors. Some algorithms were removed because of their complexity to improve the maintainability of the project. We recommend reading this guide carefully to understand all the changes that were made. You can also take a look at the rl-zoo3 and compare the imports to the rl-zoo of SB2 to have a concrete example of successful migration.

Note

If you experience massive slow-down switching to PyTorch, you may need to play with the number of threads used, using `torch.set_num_threads(1)` or `OMP_NUM_THREADS=1`, see issue #122 and issue #90.

1.17.3 Breaking Changes

- SB3 requires python 3.7+ (instead of python 3.5+ for SB2)
- Dropped MPI support
- Dropped layer normalized policies (`MlpLnLstmPolicy`, `CnnLnLstmPolicy`)
- LSTM policies (`` `MlpLstmPolicy` ``, `` `CnnLstmPolicy` ``) are not supported for the time being (see PR #53 for a recurrent PPO implementation)
- Dropped parameter noise for DDPG and DQN
- PPO is now closer to the original implementation (no clipping of the value function by default), cf PPO section below
- Orthogonal initialization is only used by A2C/PPO
- The features extractor (CNN extractor) is shared between policy and q-networks for DDPG/SAC/TD3 and only the policy loss used to update it (much faster)
- Tensorboard legacy logging was dropped in favor of having one logger for the terminal and Tensorboard (cf *Tensorboard integration*)
- We dropped ACKTR/ACER support because of their complexity compared to simpler alternatives (PPO, SAC, TD3) performing as good.
- We dropped GAIL support as we are focusing on model-free RL only, you can however take a look at the *imitation project* which implements GAIL and other imitation learning algorithms on top of SB3.
- `action_probability` is currently not implemented in the base class
- `pretrain()` method for behavior cloning was removed (see issue #27)

You can take a look at the issue about SB3 implementation design for more details.

Moved Files

- bench/monitor.py -> common/monitor.py
- logger.py -> common/logger.py
- results_plotter.py -> common/results_plotter.py
- common/cmd_util.py -> common/env_util.py

Utility functions are no longer exported from common module, you should import them with their absolute path, e.g.:

```
from stable_baselines3.common.env_util import make_atari_env, make_vec_env
from stable_baselines3.common.utils import set_random_seed
```

instead of from stable_baselines3.common import make_atari_env

Changes and renaming in parameters

Base-class (all algorithms)

- load_parameters -> set_parameters
  - get/set_parameters return a dictionary mapping object names to their respective PyTorch tensors and other objects representing their parameters, instead of simpler mapping of parameter name to a NumPy array. These functions also return PyTorch tensors rather than NumPy arrays.

Policies

- cnn_extractor -> features_extractor, as features_extractor in now used with MlpPolicy too

A2C

- epsilon -> rms_prop_eps
- lr_schedule is part of learning_rate (it can be a callable).
- alpha, momentum are modifiable through policy_kwargs key optimizer_kwargs.

⚠ **Warning**

PyTorch implementation of RMSprop differs from Tensorflow's, which leads to different and potentially more unstable results. Use stable_baselines3.common.sb2_compat.rmsprop_tf_like.RMSpropTFLike optimizer to match the results with TensorFlow's implementation. This can be done through policy_kwargs: A2C(policy_kwargs=dict(optimizer_class=RMSpropTFLike, optimizer_kwargs=dict(eps=1e-5)))

PPO

- cliprange -> clip_range
- cliprange_vf -> clip_range_vf
- nminibatches -> batch_size

> **Warning**
>
> nminibatches gave different batch size depending on the number of environments: batch_size = (n_steps * n_envs) // nminibatches

- clip_range_vf behavior for PPO is slightly different: Set it to None (default) to deactivate clipping (in SB2, you had to pass -1, None meant to use clip_range for the clipping)
- lam -> gae_lambda
- noptepochs -> n_epochs

PPO default hyperparameters are the one tuned for continuous control environment. We recommend taking a look at the *RL Zoo* for hyperparameters tuned for Atari games.

DQN

Only the vanilla DQN is implemented right now but extensions will follow. Default hyperparameters are taken from the Nature paper, except for the optimizer and learning rate that were taken from Stable Baselines defaults.

DDPG

DDPG now follows the same interface as SAC/TD3. For state/reward normalization, you should use VecNormalize as for all other algorithms.

SAC/TD3

SAC/TD3 now accept any number of critics, e.g. policy_kwargs=dict(n_critics=3), instead of only two before.

> **Note**
>
> SAC/TD3 default hyperparameters (including network architecture) now match the ones from the original papers. DDPG is using TD3 defaults.

SAC

SAC implementation matches the latest version of the original implementation: it uses two Q function networks and two target Q function networks instead of two Q function networks and one Value function network (SB2 implementation, first version of the original implementation). Despite this change, no change in performance should be expected.

> **Note**
>
> SAC predict() method has now deterministic=False by default for consistency. To match SB2 behavior, you need to explicitly pass deterministic=True

HER

The HER implementation now only supports online sampling of the new goals. This is done in a vectorized version. The goal selection strategy RANDOM is no longer supported.

New logger API

- Methods were renamed in the logger:
  - `logkv` -> `record`, `writekvs` -> `write`, `writeseq` -> `write_sequence`,
  - `logkvs` -> `record_dict`, `dumpkvs` -> `dump`,
  - `getkvs` -> `get_log_dict`, `logkv_mean` -> `record_mean`,

Internal Changes

Please read the *Developer Guide* section.

1.17.4 New Features (SB3 vs SB2)

- Much cleaner and consistent base code (and no more warnings =D!) and static type checks
- Independent saving/loading/predict for policies
- A2C now supports Generalized Advantage Estimation (GAE) and advantage normalization (both are deactivated by default)
- Generalized State-Dependent Exploration (gSDE) exploration is available for A2C/PPO/SAC. It allows using RL directly on real robots (cf https://arxiv.org/abs/2005.05719)
- Better saving/loading: optimizers are now included in the saved parameters and there are two new methods `save_replay_buffer` and `load_replay_buffer` for the replay buffer when using off-policy algorithms (DQN/DDPG/SAC/TD3)
- You can pass `optimizer_class` and `optimizer_kwargs` to `policy_kwargs` in order to easily customize optimizers
- Seeding now works properly to have deterministic results
- Replay buffer does not grow, allocate everything at build time (faster)
- We added a memory efficient replay buffer variant (pass `optimize_memory_usage=True` to the constructor), it reduces drastically the memory used especially when using images
- You can specify an arbitrary number of critics for SAC/TD3 (e.g. `policy_kwargs=dict(n_critics=3)`)

1.18 Dealing with NaNs and infs

During the training of a model on a given environment, it is possible that the RL model becomes completely corrupted when a NaN or an inf is given or returned from the RL model.

1.18.1 How and why?

The issue arises when NaNs or infs do not crash, but simply get propagated through the training, until all the floating point number converge to NaN or inf. This is in line with the IEEE Standard for Floating-Point Arithmetic (IEEE 754) standard, as it says:

Note

Five possible exceptions can occur:

- Invalid operation ($\sqrt{-1}$, inf $\times 1$, NaN $\bmod 1$, $\ldots$) return NaN
- **Division by zero:**

    - if the operand is not zero $(1/0, -2/0, \ldots)$ returns $\pm \inf$
    - if the operand is zero $(0/0)$ returns signaling NaN
- Overflow (exponent too high to represent) returns $\pm \inf$
- Underflow (exponent too low to represent) returns 0
- Inexact (not representable exactly in base 2, eg: $1/5$) returns the rounded value (ex: `assert (1/5) * 3 == 0.6000000000000001`)

And of these, only `Division by zero` will signal an exception, the rest will propagate invalid values quietly.

In python, dividing by zero will indeed raise the exception: `ZeroDivisionError: float division by zero`, but ignores the rest.

The default in numpy, will warn: `RuntimeWarning: invalid value encountered` but will not halt the code.

1.18.2 Anomaly detection with PyTorch

To enable NaN detection in PyTorch you can do

```python
import torch as th
th.autograd.set_detect_anomaly(True)
```

1.18.3 Numpy parameters

Numpy has a convenient way of dealing with invalid value: numpy.seterr, which defines for the python process, how it should handle floating point error.

```python
import numpy as np

np.seterr(all="raise")  # define before your code.

print("numpy test:")

a = np.float64(1.0)
b = np.float64(0.0)
val = a / b  # this will now raise an exception instead of a warning.
print(val)
```

but this will also avoid overflow issues on floating point numbers:

```python
import numpy as np

np.seterr(all="raise")  # define before your code.

print("numpy overflow test:")

a = np.float64(10)
b = np.float64(1000)
val = a ** b  # this will now raise an exception
print(val)
```

but will not avoid the propagation issues:

```
import numpy as np

np.seterr(all="raise")  # define before your code.

print("numpy propagation test:")

a = np.float64("NaN")
b = np.float64(1.0)
val = a + b  # this will neither warn nor raise anything
print(val)
```

1.18.4 VecCheckNan Wrapper

In order to find when and from where the invalid value originated from, stable-baselines3 comes with a `VecCheckNan` wrapper.

It will monitor the actions, observations, and rewards, indicating what action or observation caused it and from what.

```
import gymnasium as gym
from gymnasium import spaces
import numpy as np

from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import DummyVecEnv, VecCheckNan

class NanAndInfEnv(gym.Env):
    """Custom Environment that raised NaNs and Infs"""
    metadata = {"render.modes": ["human"]}

    def __init__(self):
        super(NanAndInfEnv, self).__init__()
        self.action_space = spaces.Box(low=-np.inf, high=np.inf, shape=(1,), dtype=np.
↪float64)
        self.observation_space = spaces.Box(low=-np.inf, high=np.inf, shape=(1,),␣
↪dtype=np.float64)

    def step(self, _action):
        randf = np.random.rand()
        if randf > 0.99:
            obs = float("NaN")
        elif randf > 0.98:
            obs = float("inf")
        else:
            obs = randf
        return [obs], 0.0, False, {}

    def reset(self):
        return [0.0]

    def render(self, close=False):
        pass

# Create environment
```

(continues on next page)

(continued from previous page)

```
env = DummyVecEnv([lambda: NanAndInfEnv()])
env = VecCheckNan(env, raise_exception=True)

# Instantiate the agent
model = PPO("MlpPolicy", env)

# Train the agent
model.learn(total_timesteps=int(2e5))  # this will crash explaining that the invalid value originated from the environment.
```

1.18.5 RL Model hyperparameters

Depending on your hyperparameters, NaN can occur much more often. A great example of this: https://github.com/hill-a/stable-baselines/issues/340

Be aware, the hyperparameters given by default seem to work in most cases, however your environment might not play nice with them. If this is the case, try to read up on the effect each hyperparameter has on the model, so that you can try and tune them to get a stable model. Alternatively, you can try automatic hyperparameter tuning (included in the rl zoo).

1.18.6 Missing values from datasets

If your environment is generated from an external dataset, do not forget to make sure your dataset does not contain NaNs. As some datasets will sometimes fill missing values with NaNs as a surrogate value.

Here is some reading material about finding NaNs: https://pandas.pydata.org/pandas-docs/stable/user_guide/missing_data.html

And filling the missing values with something else (imputation): https://towardsdatascience.com/how-to-handle-missing-data-8646b18db0d4

1.19 Developer Guide

This guide is meant for those who want to understand the internals and the design choices of Stable-Baselines3.

At first, you should read the two issues where the design choices were discussed:

- https://github.com/hill-a/stable-baselines/issues/576
- https://github.com/hill-a/stable-baselines/issues/733

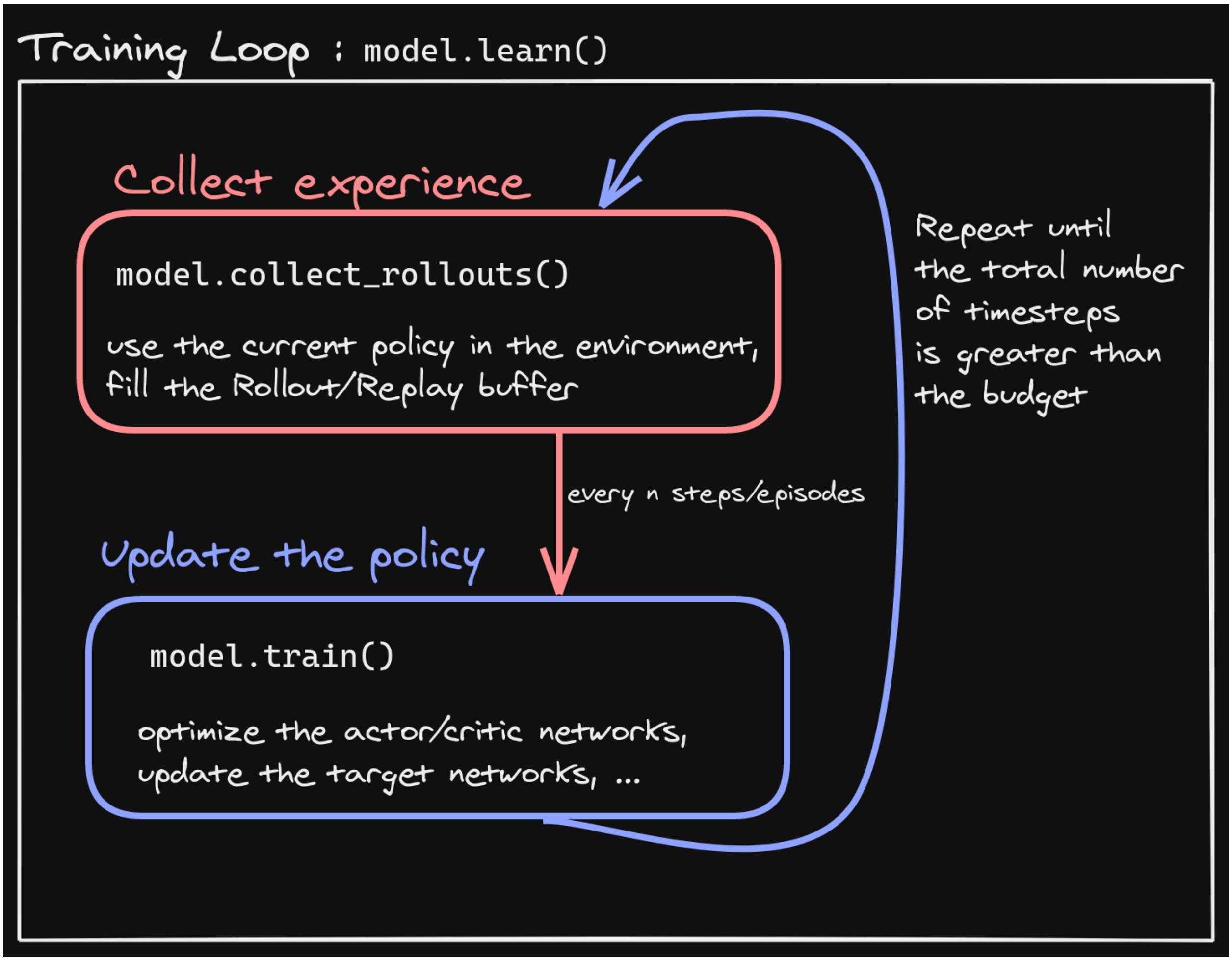
The library is not meant to be modular, although inheritance is used to reduce code duplication.

1.19.1 Algorithms Structure

Each algorithm (on-policy and off-policy ones) follows a common structure. Policy contains code for acting in the environment, and algorithm updates this policy. There is one folder per algorithm, and in that folder there is the algorithm and the policy definition (`policies.py`).

Each algorithm has two main methods:

- `.collect_rollouts()` which defines how new samples are collected, usually inherited from the base class. Those samples are then stored in a `RolloutBuffer` (discarded after the gradient update) or `ReplayBuffer`
- `.train()` which updates the parameters using samples from the buffer


Training Loop : model.learn()
Collect experience
model.collect_rollouts()
use the current policy in the environment,
fill the Rollout/Replay buffer
every n steps/episodes
Update the policy
model.train()
optimize the actor/critic networks,
update the target networks, ...
Repeat until
the total number
of timesteps
is greater than
the budget

1.19.2 Where to start?

The first thing you need to read and understand are the base classes in the `common/` folder:

- `BaseAlgorithm` in `base_class.py` which defines how an RL class should look like. It contains also all the "glue code" for saving/loading and the common operations (wrapping environments)
- `BasePolicy` in `policies.py` which defines how a policy class should look like. It contains also all the magic for the `.predict()` method, to handle as many spaces/cases as possible
- `OffPolicyAlgorithm` in `off_policy_algorithm.py` that contains the implementation of `collect_rollouts()` for the off-policy algorithms, and similarly `OnPolicyAlgorithm` in `on_policy_algorithm.py`.

All the environments handled internally are assumed to be `VecEnv` (`gym.Env` are automatically wrapped).

1.19.3 Pre-Processing

To handle different observation spaces, some pre-processing needs to be done (e.g. one-hot encoding for discrete observation). Most of the code for pre-processing is in `common/preprocessing.py` and `common/policies.py`.

For images, environment is automatically wrapped with `VecTransposeImage` if observations are detected to be images with channel-last convention to transform it to PyTorch's channel-first convention.

1.19.4 Policy Structure

When we refer to "policy" in Stable-Baselines3, this is usually an abuse of language compared to RL terminology. In SB3, "policy" refers to the class that handles all the networks useful for training, so not only the network used to predict actions (the "learned controller"). For instance, the `TD3` policy contains the actor, the critic and the target networks.

To avoid the hassle of importing specific policy classes for specific algorithm (e.g. both A2C and PPO use `ActorCriticPolicy`), SB3 uses names like "MlpPolicy" and "CnnPolicy" to refer policies using small feed-forward networks or convolutional networks, respectively. Importing `[algorithm]/policies.py` registers an appropriate policy for that algorithm under those names.

1.19.5 Probability distributions

When needed, the policies handle the different probability distributions. All distributions are located in `common/distributions.py` and follow the same interface. Each distribution corresponds to a type of action space (e.g. `Categorical` is the one used for discrete actions. For continuous actions, we can use multiple distributions ("DiagGaussian", "SquashedGaussian" or "StateDependentDistribution")

1.19.6 State-Dependent Exploration

State-Dependent Exploration (SDE) is a type of exploration that allows to use RL directly on real robots, that was the starting point for the Stable-Baselines3 library. I (@araffin) published a paper about a generalized version of SDE (the one implemented in SB3): https://arxiv.org/abs/2005.05719

1.19.7 Misc

The rest of the `common/` is composed of helpers (e.g. evaluation helpers) or basic components (like the callbacks). The `type_aliases.py` file contains common type hint aliases like `GymStepReturn`.

Et voilà?

After reading this guide and the mentioned files, you should be now able to understand the design logic behind the library ;)

1.20 On saving and loading

Stable Baselines3 (SB3) stores both neural network parameters and algorithm-related parameters such as exploration schedule, number of environments and observation/action space. This allows continual learning and easy use of trained agents without training, but it is not without its issues. Following describes the format used to save agents in SB3 along with its pros and shortcomings.

Terminology used in this page:

- *parameters* refer to neural network parameters (also called "weights"). This is a dictionary mapping variable name to a PyTorch tensor.
- *data* refers to RL algorithm parameters, e.g. learning rate, exploration schedule, action/observation space. These depend on the algorithm used. This is a dictionary mapping classes variable names to their values.

1.20.1 Zip-archive

A zip-archived JSON dump, PyTorch state dictionaries and PyTorch variables. The data dictionary (class parameters) is stored as a JSON file, model parameters and optimizers are serialized with `torch.save()` function and these files are stored under a single .zip archive.

Any objects that are not JSON serializable are serialized with cloudpickle and stored as base64-encoded string in the JSON file, along with some information that was stored in the serialization. This allows inspecting stored objects without deserializing the object itself.

This format allows skipping elements in the file, i.e. we can skip deserializing objects that are broken/non-serializable. This can be done via `custom_objects` argument to load functions.

Note

If you encounter loading issue, for instance pickle issues or error after loading (see #171 or #573), you can pass `print_system_info=True` to compare the system on which the model was trained vs the current one `model = PPO.load("ppo_saved", print_system_info=True)`

File structure:

```
saved_model.zip/
├── data              JSON file of class-parameters (dictionary)
├── *.optimizer.pth   PyTorch optimizers serialized
├── policy.pth        PyTorch state dictionary of the policy saved
├── pytorch_variables.pth Additional PyTorch variables
├── _stable_baselines3_version contains the SB3 version with which the model was saved
├── system_info.txt contains system info (os, python version, ...) on which the model␣
↪was saved
```

Pros:

- More robust to unserializable objects (one bad object does not break everything).
- Saved files can be inspected/extracted with zip-archive explorers and by other languages.

Cons:

- More complex implementation.
- Still relies partly on cloudpickle for complex objects (e.g. custom functions) with can lead to incompatibilities between Python versions.

1.21 Exporting models

After training an agent, you may want to deploy/use it in another language or framework, like tensorflowjs. Stable Baselines3 does not include tools to export models to other frameworks, but this document aims to cover parts that are required for exporting along with more detailed stories from users of Stable Baselines3.

1.21.1 Background

In Stable Baselines3, the controller is stored inside policies which convert observations into actions. Each learning algorithm (e.g. DQN, A2C, SAC) contains a policy object which represents the currently learned behavior, accessible via `model.policy`.

Policies hold enough information to do the inference (i.e. predict actions), so it is enough to export these policies (cf *examples*) to do inference in another framework.

Warning

When using CNN policies, the observation is normalized during pre-preprocessing. This pre-processing is done *inside* the policy (dividing by 255 to have values in [0, 1])

1.21.2 Export to ONNX

If you are using PyTorch 2.0+ and ONNX Opset 14+, you can easily export SB3 policies using the following code:

> ⚠ **Warning**
>
> The following returns normalized actions and doesn’t include the post-processing step that is done with continuous actions (clip or unscale the action to the correct space).

```python
import torch as th
from typing import Tuple

from stable_baselines3 import PPO
from stable_baselines3.common.policies import BasePolicy


class OnnxableSB3Policy(th.nn.Module):
    def __init__(self, policy: BasePolicy):
        super().__init__()
        self.policy = policy

    def forward(self, observation: th.Tensor) -> Tuple[th.Tensor, th.Tensor, th.Tensor]:
        # NOTE: Preprocessing is included, but postprocessing
        # (clipping/inscaling actions) is not,
        # If needed, you also need to transpose the images so that they are channel first
        # use deterministic=False if you want to export the stochastic policy
        # policy() returns `actions, values, log_prob` for PPO
        return self.policy(observation, deterministic=True)


# Example: model = PPO("MlpPolicy", "Pendulum-v1")
PPO("MlpPolicy", "Pendulum-v1").save("PathToTrainedModel")
model = PPO.load("PathToTrainedModel.zip", device="cpu")

onnx_policy = OnnxableSB3Policy(model.policy)

observation_size = model.observation_space.shape
dummy_input = th.randn(1, *observation_size)
th.onnx.export(
    onnx_policy,
    dummy_input,
    "my_ppo_model.onnx",
    opset_version=17,
    input_names=["input"],
)

##### Load and test with onnx

import onnx
import onnxruntime as ort
import numpy as np
```

(continues on next page)

(continued from previous page)

```
onnx_path = "my_ppo_model.onnx"
onnx_model = onnx.load(onnx_path)
onnx.checker.check_model(onnx_model)

observation = np.zeros((1, *observation_size)).astype(np.float32)
ort_sess = ort.InferenceSession(onnx_path)
actions, values, log_prob = ort_sess.run(None, {"input": observation})

print(actions, values, log_prob)

# Check that the predictions are the same
with th.no_grad():
    print(model.policy(th.as_tensor(observation), deterministic=True))
```

For exporting `MultiInputPolicy`, please have a look at GH#1873.

For SAC the procedure is similar. The example shown only exports the actor network as the actor is sufficient to roll out the trained policies.

```
import torch as th

from stable_baselines3 import SAC


class OnnxablePolicy(th.nn.Module):
    def __init__(self, actor: th.nn.Module):
        super().__init__()
        self.actor = actor

    def forward(self, observation: th.Tensor) -> th.Tensor:
        # NOTE: You may have to postprocess (unnormalize) actions
        # to the correct bounds (see commented code below)
        return self.actor(observation, deterministic=True)


# Example: model = SAC("MlpPolicy", "Pendulum-v1")
SAC("MlpPolicy", "Pendulum-v1").save("PathToTrainedModel.zip")
model = SAC.load("PathToTrainedModel.zip", device="cpu")
onnxable_model = OnnxablePolicy(model.policy.actor)

observation_size = model.observation_space.shape
dummy_input = th.randn(1, *observation_size)
th.onnx.export(
    onnxable_model,
    dummy_input,
    "my_sac_actor.onnx",
    opset_version=17,
    input_names=["input"],
)

##### Load and test with onnx

import onnxruntime as ort
```

(continues on next page)

(continued from previous page)

```
import numpy as np

onnx_path = "my_sac_actor.onnx"

observation = np.zeros((1, *observation_size)).astype(np.float32)
ort_sess = ort.InferenceSession(onnx_path)
scaled_action = ort_sess.run(None, {"input": observation})[0]

print(scaled_action)

# Post-process: rescale to correct space
# Rescale the action from [-1, 1] to [low, high]
# low, high = model.action_space.low, model.action_space.high
# post_processed_action = low + (0.5 * (scaled_action + 1.0) * (high - low))

# Check that the predictions are the same
with th.no_grad():
    print(model.actor(th.as_tensor(observation), deterministic=True))
```

For more discussion around the topic, please refer to GH#383 and GH#1349.

1.21.3 Trace/Export to C++

You can use PyTorch JIT to trace and save a trained model that can be reused in other applications (for instance inference code written in C++).

There is a draft PR in the RL Zoo about C++ export: https://github.com/DLR-RM/rl-baselines3-zoo/pull/228

```
# See "ONNX export" for imports and OnnxablePolicy
jit_path = "sac_traced.pt"

# Trace and optimize the module
traced_module = th.jit.trace(onnxable_model.eval(), dummy_input)
frozen_module = th.jit.freeze(traced_module)
frozen_module = th.jit.optimize_for_inference(frozen_module)
th.jit.save(frozen_module, jit_path)

##### Load and test with torch

import torch as th

dummy_input = th.randn(1, *observation_size)
loaded_module = th.jit.load(jit_path)
action_jit = loaded_module(dummy_input)
```

1.21.4 Export to tensorflowjs / ONNX-JS

TODO: contributors help is welcomed! Probably a good starting point: https://github.com/elliotwaite/pytorch-to-javascript-with-onnx-js

1.21.5 Export to TFLite / Coral (Edge TPU)

Full example code: https://github.com/chunky/sb3_to_coral

Google created a chip called the "Coral" for deploying AI to the edge. It's available in a variety of form factors, including USB (using the Coral on a Raspberry Pi, with a SB3-developed model, was the original motivation for the code example above).

The Coral chip is fast, with very low power consumption, but only has limited on-device training abilities. More information is on the webpage here: https://coral.ai.

To deploy to a Coral, one must work via TFLite, and quantize the network to reflect the Coral's capabilities. The full chain to go from SB3 to Coral is: SB3 (Torch) => ONNX => TensorFlow => TFLite => Coral.

The code linked above is a complete, minimal, example that:

1. Creates a model using SB3
2. Follows the path of exports all the way to TFLite and Google Coral
3. Demonstrates the forward pass for most exported variants

There are a number of pitfalls along the way to the complete conversion that this example covers, including:

- Making the Gym's observation work with ONNX properly
- Quantising the TFLite model appropriately to align with Gym while still taking advantage of Coral
- Using OnnxablePolicy described as described in the above example

1.21.6 Manual export

You can also manually export required parameters (weights) and construct the network in your desired framework.

You can access parameters of the model via agents' `get_parameters` function. As policies are also PyTorch modules, you can also access `model.policy.state_dict()` directly. To find the architecture of the networks for each algorithm, best is to check the `policies.py` file located in their respective folders.

> **Note**
>
> In most cases, we recommend using PyTorch methods `state_dict()` and `load_state_dict()` from the policy, unless you need to access the optimizers' state dict too. In that case, you need to call `get_parameters()`.

Abstract base classes for RL algorithms.

1.22 Base RL Class

Common interface for all the RL algorithms

`class stable_baselines3.common.base_class.`**`BaseAlgorithm`**(*policy*, *env*, *learning_rate*, *policy_kwargs=None*, *stats_window_size=100*, *tensorboard_log=None*, *verbose=0*, *device='auto'*, *support_multi_env=False*, *monitor_wrapper=True*, *seed=None*, *use_sde=False*, *sde_sample_freq=-1*, *supported_action_spaces=None*)

The base of RL algorithms

Parameters

- **policy** (*BasePolicy*) – The policy model to use (MlpPolicy, CnnPolicy, ...)
- **env** (*Env | VecEnv | str | None*) – The environment to learn from (if registered in Gym, can be str. Can be None for loading trained models)
- **learning_rate** (*float | Callable[[float], float]*) – learning rate for the optimizer, it can be a function of the current progress remaining (from 1 to 0)
- **policy_kwargs** (*dict[str, Any] | None*) – Additional arguments to be passed to the policy on creation
- **stats_window_size** (*int*) – Window size for the rollout logging, specifying the number of episodes to average the reported success rate, mean episode length, and mean reward over
- **tensorboard_log** (*str | None*) – the log location for tensorboard (if None, no logging)
- **verbose** (*int*) – Verbosity level: 0 for no output, 1 for info messages (such as device or wrappers used), 2 for debug messages
- **device** (*device | str*) – Device on which the code should run. By default, it will try to use a Cuda compatible device and fallback to cpu if it is not possible.
- **support_multi_env** (*bool*) – Whether the algorithm supports training with multiple environments (as in A2C)
- **monitor_wrapper** (*bool*) – When creating an environment, whether to wrap it or not in a Monitor wrapper.
- **seed** (*int | None*) – Seed for the pseudo random generators
- **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)
- **supported_action_spaces** (*tuple[type[Space], ...] | None*) – The action spaces supported by the algorithm.

dump_logs()

Write log data. (Implemented by OffPolicyAlgorithm and OnPolicyAlgorithm)

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

`abstractmethod learn(`*total_timesteps*, *callback=None*, *log_interval=100*, *tb_log_name='run'*, *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** (*SelfBaseAlgorithm*)

Returns
the trained model

Return type
SelfBaseAlgorithm

`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

1.22.1 Base Off-Policy Class

The base RL algorithm for Off-Policy algorithm (ex: SAC/TD3)

class stable_baselines3.common.off_policy_algorithm.**OffPolicyAlgorithm**(*policy*, *env*, *learning_rate*, *buffer_size=1000000*, *learning_starts=100*, *batch_size=256*, *tau=0.005*, *gamma=0.99*, *train_freq=(1, 'step')*, *gradient_steps=1*, *action_noise=None*, *replay_buffer_class=None*, *replay_buffer_kwargs=None*, *optimize_memory_usage=False*, *policy_kwargs=None*, *stats_window_size=100*, *tensorboard_log=None*, *verbose=0*, *device='auto'*, *support_multi_env=False*, *monitor_wrapper=True*, *seed=None*, *use_sde=False*, *sde_sample_freq=-1*, *use_sde_at_warmup=False*, *sde_support=True*, *supported_action_spaces=None*)

The base for Off-Policy algorithms (ex: SAC/TD3)

Parameters

- **policy** (*BasePolicy*) – The policy model to use (MlpPolicy, CnnPolicy, …)
- **env** (*Env | VecEnv | str*) – The environment to learn from (if registered in Gym, can be str. Can be None for loading trained models)
- **learning_rate** (*float | Callable[[float], float]*) – learning rate for the optimizer, 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 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
- **policy_kwargs** (*dict[str, Any] | None*) – Additional arguments to be passed to the policy on creation
- **stats_window_size** (*int*) – Window size for the rollout logging, specifying the number of episodes to average the reported success rate, mean episode length, and mean reward over
- **tensorboard_log** (*str | None*) – the log location for tensorboard (if None, no logging)
- **verbose** (*int*) – Verbosity level: 0 for no output, 1 for info messages (such as device or wrappers used), 2 for debug messages
- **device** (*device | str*) – Device on which the code should run. By default, it will try to use a Cuda compatible device and fallback to cpu if it is not possible.
- **support_multi_env** (*bool*) – Whether the algorithm supports training with multiple environments (as in A2C)
- **monitor_wrapper** (*bool*) – When creating an environment, whether to wrap it or not in a Monitor wrapper.
- **seed** (*int | None*) – Seed for the pseudo random generators
- **use_sde** (*bool*) – Whether to use State Dependent Exploration (SDE) 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)
- **sde_support** (*bool*) – Whether the model support gSDE or not
- **supported_action_spaces** (*tuple[type[Space], ...] | None*) – The action spaces supported by the algorithm.

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

`learn(total_timesteps, callback=None, log_interval=4, tb_log_name='run', 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** (*SelfOffPolicyAlgorithm*)

Returns
the trained model

Return type
SelfOffPolicyAlgorithm

`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

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

train(*gradient_steps*, *batch_size*)

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

Parameters

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

Return type

None

1.22.2 Base On-Policy Class

The base RL algorithm for On-Policy algorithm (ex: A2C/PPO)

class stable_baselines3.common.on_policy_algorithm.**OnPolicyAlgorithm**(*policy*, *env*, *learning_rate*, *n_steps*, *gamma*, *gae_lambda*, *ent_coef*, *vf_coef*, *max_grad_norm*, *use_sde*, *sde_sample_freq*, *rollout_buffer_class=None*, *rollout_buffer_kwargs=None*, *stats_window_size=100*, *tensorboard_log=None*, *monitor_wrapper=True*, *policy_kwargs=None*, *verbose=0*, *seed=None*, *device='auto'*, *_init_setup_model=True*, *supported_action_spaces=None*)

The base for On-Policy algorithms (ex: A2C/PPO).

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)
- **gamma** (*float*) – Discount factor

- **gae_lambda** (*float*) – Factor for trade-off of bias vs variance for Generalized Advantage Estimator. Equivalent to classic advantage when set to 1.
- **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
- **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.
- **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)
- **monitor_wrapper** (*bool*) – When creating an environment, whether to wrap it or not in a Monitor wrapper.
- **policy_kwargs** (*dict[str, Any] | None*) – additional arguments to be passed to the policy on creation
- **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
- **supported_action_spaces** (*tuple[type[Space], ...] | None*) – The action spaces supported by the algorithm.

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

learn(*total_timesteps*, *callback=None*, *log_interval=1*, *tb_log_name='OnPolicyAlgorithm'*, *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** (*SelfOnPolicyAlgorithm*)

Returns
the trained model

Return type
SelfOnPolicyAlgorithm

train()

Consume current rollout data and update policy parameters. Implemented by individual algorithms.

Return type
None

1.23 A2C

A synchronous, deterministic variant of Asynchronous Advantage Actor Critic (A3C). It uses multiple workers to avoid the use of a replay buffer.

> ⚠ **Warning**
>
> If you find training unstable or want to match performance of stable-baselines A2C, consider using `RMSpropTFLike` optimizer from `stable_baselines3.common.sb2_compat.rmsprop_tf_like`. You can change optimizer with `A2C(policy_kwargs=dict(optimizer_class=RMSpropTFLike, optimizer_kwargs=dict(eps=1e-5)))`. Read more here.

1.23.1 Notes

- Original paper: https://arxiv.org/abs/1602.01783
- OpenAI blog post: https://openai.com/blog/baselines-acktr-a2c/

1.23.2 Can I use?

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

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

1.23.3 Example

This example is only to demonstrate the use of the library and its functions, and the trained agents may not solve the environments. Optimized hyperparameters can be found in RL Zoo repository.

Train a A2C agent on `CartPole-v1` using 4 environments.

```python
from stable_baselines3 import A2C
from stable_baselines3.common.env_util import make_vec_env

# Parallel environments
vec_env = make_vec_env("CartPole-v1", n_envs=4)

model = A2C("MlpPolicy", vec_env, verbose=1)
model.learn(total_timesteps=25000)
model.save("a2c_cartpole")

del model # remove to demonstrate saving and loading

model = A2C.load("a2c_cartpole")

obs = vec_env.reset()
while True:
    action, _states = model.predict(obs)
    obs, rewards, dones, info = vec_env.step(action)
    vec_env.render("human")
```

Note

A2C is meant to be run primarily on the CPU, especially when you are not using a CNN. To improve CPU utilization, try turning off the GPU and using `SubprocVecEnv` instead of the default `DummyVecEnv`:

```python
from stable_baselines3 import A2C
from stable_baselines3.common.env_util import make_vec_env
from stable_baselines3.common.vec_env import SubprocVecEnv

if __name__=="__main__":
    env = make_vec_env("CartPole-v1", n_envs=8, vec_env_cls=SubprocVecEnv)
    model = A2C("MlpPolicy", env, device="cpu")
    model.learn(total_timesteps=25_000)
```

For more information, see *Vectorized Environments*, Issue #1245 or the Multiprocessing notebook.

1.23.4 Results

Atari Games

The complete learning curves are available in the associated PR #110.

PyBullet Environments

Results on the PyBullet benchmark (2M steps) using 6 seeds. The complete learning curves are available in the associated issue #48.

Note

Hyperparameters from the gSDE paper were used (as they are tuned for PyBullet envs).

Gaussian means that the unstructured Gaussian noise is used for exploration, *gSDE* (generalized State-Dependent Exploration) is used otherwise.

Environments	A2C	A2C	PPO	PPO
	Gaussian	gSDE	Gaussian	gSDE
HalfCheetah	2003 +/- 54	2032 +/- 122	1976 +/- 479	2826 +/- 45
Ant	2286 +/- 72	2443 +/- 89	2364 +/- 120	2782 +/- 76
Hopper	1627 +/- 158	1561 +/- 220	1567 +/- 339	2512 +/- 21
Walker2D	577 +/- 65	839 +/- 56	1230 +/- 147	2019 +/- 64

How to replicate the results?

Clone the rl-zoo repo:

```bash
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):

```bash
python train.py --algo a2c --env $ENV_ID --eval-episodes 10 --eval-freq 10000
```

Plot the results (here for PyBullet envs only):

```bash
python scripts/all_plots.py -a a2c -e HalfCheetah Ant Hopper Walker2D -f logs/ -o logs/
↪a2c_results
python scripts/plot_from_file.py -i logs/a2c_results.pkl -latex -l A2C
```

1.23.5 Parameters

class stable_baselines3.a2c.**A2C**(*policy*, *env*, *learning_rate=0.0007*, *n_steps=5*, *gamma=0.99*, *gae_lambda=1.0*, *ent_coef=0.0*, *vf_coef=0.5*, *max_grad_norm=0.5*, *rms_prop_eps=1e-05*, *use_rms_prop=True*, *use_sde=False*, *sde_sample_freq=-1*, *rollout_buffer_class=None*, *rollout_buffer_kwargs=None*, *normalize_advantage=False*, *stats_window_size=100*, *tensorboard_log=None*, *policy_kwargs=None*, *verbose=0*, *seed=None*, *device='auto'*, *_init_setup_model=True*)

Advantage Actor Critic (A2C)

Paper: https://arxiv.org/abs/1602.01783 Code: This implementation borrows code from https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail and and Stable Baselines (https://github.com/hill-a/stable-baselines)

Introduction to A2C: https://hackernoon.com/intuitive-rl-intro-to-advantage-actor-critic-a2c-4ff545978752

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)
- **gamma** (*float*) – Discount factor
- **gae_lambda** (*float*) – Factor for trade-off of bias vs variance for Generalized Advantage Estimator. Equivalent to classic advantage when set to 1.
- **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
- **rms_prop_eps** (*float*) – RMSProp epsilon. It stabilizes square root computation in denominator of RMSProp update
- **use_rms_prop** (*bool*) – Whether to use RMSprop (default) or Adam as optimizer
- **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
- **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 *A2C 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

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

learn(*total_timesteps*, *callback=None*, *log_interval=100*, *tb_log_name='A2C'*, *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** (*SelfA2C*)

Returns

the trained model

Return type

SelfA2C

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 (one gradient step over whole data).

Return type
None

1.23.6 A2C Policies

`stable_baselines3.a2c.`**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

stable_baselines3.a2c.**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

`stable_baselines3.a2c.`**`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

1.24 DDPG

Deep Deterministic Policy Gradient (DDPG) combines the trick for DQN with the deterministic policy gradient, to obtain an algorithm for continuous actions.

Note

As `DDPG` can be seen as a special case of its successor *TD3*, they share the same policies and same implementation.

Available Policies

MlpPolicy	alias of `TD3Policy`
`CnnPolicy`	Policy class (with both actor and critic) for TD3.

continues on next page

Table 1 – continued from previous page

`MultiInputPolicy`	Policy class (with both actor and critic) for TD3 to be used with Dict observation spaces.

1.24.1 Notes

- Deterministic Policy Gradient: http://proceedings.mlr.press/v32/silver14.pdf
- DDPG Paper: https://arxiv.org/abs/1509.02971
- OpenAI Spinning Guide for DDPG: https://spinningup.openai.com/en/latest/algorithms/ddpg.html

1.24.2 Can I use?

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

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

1.24.3 Example

This example is only to demonstrate the use of the library and its functions, and the trained agents may not solve the environments. Optimized hyperparameters can be found in RL Zoo repository.

```
import gymnasium as gym
import numpy as np

from stable_baselines3 import DDPG
from stable_baselines3.common.noise import NormalActionNoise, OrnsteinUhlenbeckActionNoise

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

# The noise objects for DDPG
n_actions = env.action_space.shape[-1]
action_noise = NormalActionNoise(mean=np.zeros(n_actions), sigma=0.1 * np.ones(n_actions))

model = DDPG("MlpPolicy", env, action_noise=action_noise, verbose=1)
model.learn(total_timesteps=10000, log_interval=10)
model.save("ddpg_pendulum")
vec_env = model.get_env()

del model # remove to demonstrate saving and loading

model = DDPG.load("ddpg_pendulum")
```

(continues on next page)

(continued from previous page)

```
obs = vec_env.reset()
while True:
    action, _states = model.predict(obs)
    obs, rewards, dones, info = vec_env.step(action)
    env.render("human")
```

1.24.4 Results

PyBullet Environments

Results on the PyBullet benchmark (1M steps) using 6 seeds. The complete learning curves are available in the associated issue #48.

Note

Hyperparameters of *TD3* from the gSDE paper were used for DDPG.

Gaussian means that the unstructured Gaussian noise is used for exploration, *gSDE* (generalized State-Dependent Exploration) is used otherwise.

Environments	DDPG	TD3	SAC
	Gaussian	Gaussian	gSDE
HalfCheetah	2272 +/- 69	2774 +/- 35	2984 +/- 202
Ant	1651 +/- 407	3305 +/- 43	3102 +/- 37
Hopper	1201 +/- 211	2429 +/- 126	2262 +/- 1
Walker2D	882 +/- 186	2063 +/- 185	2136 +/- 67

How to replicate the results?

Clone the rl-zoo repo:

```
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 ddpg --env $ENV_ID --eval-episodes 10 --eval-freq 10000
```

Plot the results:

```
python scripts/all_plots.py -a ddpg -e HalfCheetah Ant Hopper Walker2D -f logs/ -o logs/
↪ddpg_results
python scripts/plot_from_file.py -i logs/ddpg_results.pkl -latex -l DDPG
```

1.24.5 Parameters

class `stable_baselines3.ddpg.DDPG`(*policy*, *env*, *learning_rate=0.001*, *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*, *tensorboard_log=None*, *policy_kwargs=None*, *verbose=0*, *seed=None*, *device='auto'*, *_init_setup_model=True*)

Deep Deterministic Policy Gradient (DDPG).

Deterministic Policy Gradient: http://proceedings.mlr.press/v32/silver14.pdf DDPG Paper: https://arxiv.org/abs/1509.02971 Introduction to DDPG: https://spinningup.openai.com/en/latest/algorithms/ddpg.html

Note: we treat DDPG as a special case of its successor TD3.

Parameters

- **policy** (*TD3Policy*) – 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]*) – 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 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
- **policy_kwargs** (*dict[str, Any]* | *None*) – additional arguments to be passed to the policy on creation. See *DDPG 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
- **tensorboard_log** (*str | None*)

`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='DDPG'*, *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** (*SelfDDPG*)

Returns

the trained model

Return type

SelfDDPG

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

1.24.6 DDPG Policies

`stable_baselines3.ddpg.MlpPolicy`

alias of `TD3Policy`

class `stable_baselines3.td3.policies.TD3Policy`(*observation_space*, *action_space*, *lr_schedule*, *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*, *n_critics=2*, *share_features_extractor=False*)

Policy class (with both actor and critic) for TD3.

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
- **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(*observation*, *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

- **observation** (*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.

Parameters

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

Return type

None

class stable_baselines3.ddpg.**CnnPolicy**(*observation_space*, *action_space*, *lr_schedule*, *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*, *n_critics=2*, *share_features_extractor=False*)

Policy class (with both actor and critic) for TD3.

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
- **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)

class stable_baselines3.ddpg.**MultiInputPolicy**(*observation_space*, *action_space*, *lr_schedule*, *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*, *n_critics=2*, *share_features_extractor=False*)

Policy class (with both actor and critic) for TD3 to be used with Dict observation spaces.

Parameters

- **observation_space** (*Dict*) – 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
- **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)

1.25 DQN

Deep Q Network (DQN) builds on Fitted Q-Iteration (FQI) and make use of different tricks to stabilize the learning with neural networks: it uses a replay buffer, a target network and gradient clipping.

Available Policies

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

1.25.1 Notes

- Original paper: https://arxiv.org/abs/1312.5602
- Further reference: https://www.nature.com/articles/nature14236
- Tutorial "From Tabular Q-Learning to DQN": https://github.com/araffin/rlss23-dqn-tutorial

> **Note**
>
> This implementation provides only vanilla Deep Q-Learning and has no extensions such as Double-DQN, Dueling-DQN and Prioritized Experience Replay.

1.25.2 Can I use?

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

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

1.25.3 Example

This example is only to demonstrate the use of the library and its functions, and the trained agents may not solve the environments. Optimized hyperparameters can be found in RL Zoo repository.

```python
import gymnasium as gym

from stable_baselines3 import DQN

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

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

del model # remove to demonstrate saving and loading

model = DQN.load("dqn_cartpole")

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

1.25.4 Results

Atari Games

The complete learning curves are available in the associated PR #110.

How to replicate the results?

Clone the rl-zoo repo:

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

Run the benchmark (replace `$ENV_ID` by the env id, for instance `BreakoutNoFrameskip-v4`):

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

Plot the results:

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

1.25.5 Parameters

class `stable_baselines3.dqn.`**`DQN`**(*policy*, *env*, *learning_rate=0.0001*, *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*, *target_update_interval=10000*, *exploration_fraction=0.1*, *exploration_initial_eps=1.0*, *exploration_final_eps=0.05*, *max_grad_norm=10*, *stats_window_size=100*, *tensorboard_log=None*, *policy_kwargs=None*, *verbose=0*, *seed=None*, *device='auto'*, *_init_setup_model=True*)

Deep Q-Network (DQN)

Paper: https://arxiv.org/abs/1312.5602, https://www.nature.com/articles/nature14236 Default hyperparameters are taken from the Nature paper, except for the optimizer and learning rate that were taken from Stable Baselines defaults.

Parameters

- **policy** (*DQNPolicy*) – 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`) 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
- **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*) – The maximum value for the gradient 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 *DQN 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

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='DQN'*, *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** (*SelfDQN*)

Returns
the trained model

Return type
SelfDQN

`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*)

Overrides the base_class predict function to include epsilon-greedy exploration.

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.

Returns
the model's action and the next 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

1.25.6 DQN Policies

stable_baselines3.dqn.**MlpPolicy**

alias of DQNPolicy

class stable_baselines3.dqn.policies.**DQNPolicy**(*observation_space*, *action_space*, *lr_schedule*, *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 Q-Value Net and target net for DQN

Parameters

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

Parameters

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

Return type

None

class stable_baselines3.dqn.**CnnPolicy**(*observation_space*, *action_space*, *lr_schedule*, *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 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)
- **net_arch** (*list[int] | None*) – The specification of the policy and value networks.
- **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 stable_baselines3.dqn.**MultiInputPolicy**(*observation_space*, *action_space*, *lr_schedule*, *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 DQN when using dict observations as input.

Parameters

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

1.26 HER

Hindsight Experience Replay (HER)

HER is an algorithm that works with off-policy methods (DQN, SAC, TD3 and DDPG for example). HER uses the fact that even if a desired goal was not achieved, other goal may have been achieved during a rollout. It creates “virtual” transitions by relabeling transitions (changing the desired goal) from past episodes.

> ⚠ **Warning**
>
> Starting from Stable Baselines3 v1.1.0, `HER` is no longer a separate algorithm but a replay buffer class `HerReplayBuffer` that must be passed to an off-policy algorithm when using `MultiInputPolicy` (to have Dict observation support).

Warning

HER requires the environment to follow the legacy gym_robotics.GoalEnv interface In short, the `gym.Env` must have: - a vectorized implementation of `compute_reward()` - a dictionary observation space with three keys: `observation`, `achieved_goal` and `desired_goal`

Warning

Because it needs access to `env.compute_reward()` HER must be loaded with the env. If you just want to use the trained policy without instantiating the environment, we recommend saving the policy only.

Note

Compared to other implementations, the `future` goal sampling strategy is inclusive: the current transition can be used when re-sampling.

1.26.1 Notes

- Original paper: https://arxiv.org/abs/1707.01495
- OpenAI paper: Plappert et al. (2018)
- OpenAI blog post: https://openai.com/blog/ingredients-for-robotics-research/

1.26.2 Can I use?

Please refer to the used model (DQN, QR-DQN, SAC, TQC, TD3, or DDPG) for that section.

1.26.3 Example

This example is only to demonstrate the use of the library and its functions, and the trained agents may not solve the environments. Optimized hyperparameters can be found in RL Zoo repository.

```
from stable_baselines3 import HerReplayBuffer, DDPG, DQN, SAC, TD3
from stable_baselines3.her.goal_selection_strategy import GoalSelectionStrategy
from stable_baselines3.common.envs import BitFlippingEnv

model_class = DQN  # works also with SAC, DDPG and TD3
N_BITS = 15

env = BitFlippingEnv(n_bits=N_BITS, continuous=model_class in [DDPG, SAC, TD3], max_
↪steps=N_BITS)

# Available strategies (cf paper): future, final, episode
goal_selection_strategy = "future" # equivalent to GoalSelectionStrategy.FUTURE

# Initialize the model
model = model_class(
    "MultiInputPolicy",
    env,
    replay_buffer_class=HerReplayBuffer,
```

(continues on next page)

(continued from previous page)

```
    # Parameters for HER
    replay_buffer_kwargs=dict(
        n_sampled_goal=4,
        goal_selection_strategy=goal_selection_strategy,
    ),
    verbose=1,
)

# Train the model
model.learn(1000)

model.save("./her_bit_env")
# Because it needs access to `env.compute_reward()`
# HER must be loaded with the env
model = model_class.load("./her_bit_env", env=env)

obs, info = env.reset()
for _ in range(100):
    action, _ = model.predict(obs, deterministic=True)
    obs, reward, terminated, truncated, _ = env.step(action)
    if terminated or truncated:
        obs, info = env.reset()
```

1.26.4 Results

This implementation was tested on the parking env using 3 seeds.

The complete learning curves are available in the associated PR #120.

How to replicate the results?

Clone the rl-zoo repo:

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

Run the benchmark:

```
python train.py --algo tqc --env parking-v0 --eval-episodes 10 --eval-freq 10000
```

Plot the results:

```
python scripts/all_plots.py -a tqc -e parking-v0 -f logs/ --no-million
```

1.26.5 Parameters

1.26.6 HER Replay Buffer

class stable_baselines3.her.**HerReplayBuffer**(*buffer_size*, *observation_space*, *action_space*, *env*, *device='auto'*, *n_envs=1*, *optimize_memory_usage=False*, *handle_timeout_termination=True*, *n_sampled_goal=4*, *goal_selection_strategy='future'*, *copy_info_dict=False*)

Hindsight Experience Replay (HER) buffer. Paper: https://arxiv.org/abs/1707.01495

Replay buffer for sampling HER (Hindsight Experience Replay) transitions.

Note

Compared to other implementations, the `future` goal sampling strategy is inclusive: the current transition can be used when re-sampling.

Parameters

- **buffer_size** (*int*) – Max number of element in the buffer
- **observation_space** (*Dict*) – Observation space
- **action_space** (*Space*) – Action space
- **env** (`VecEnv` *| None*) – The training environment
- **device** (*device | str*) – PyTorch device
- **n_envs** (*int*) – Number of parallel environments
- **optimize_memory_usage** (*bool*) – Enable a memory efficient variant Disabled for now (see https://github.com/DLR-RM/stable-baselines3/pull/243#discussion_r531535702)
- **handle_timeout_termination** (*bool*) – Handle timeout termination (due to timelimit) separately and treat the task as infinite horizon task. https://github.com/DLR-RM/stable-baselines3/issues/284
- **n_sampled_goal** (*int*) – Number of virtual transitions to create per real transition, by sampling new goals.
- **goal_selection_strategy** (`GoalSelectionStrategy` *| str*) – Strategy for sampling goals for replay. One of [‘episode’, ‘final’, ‘future’]
- **copy_info_dict** (*bool*) – Whether to copy the info dictionary and pass it to `compute_reward()` method. Please note that the copy may cause a slowdown. False by default.

add(*obs*, *next_obs*, *action*, *reward*, *done*, *infos*)

Add elements to the buffer.

Parameters

- **obs** (*dict[str, ndarray]*)
- **next_obs** (*dict[str, ndarray]*)
- **action** (*ndarray*)
- **reward** (*ndarray*)
- **done** (*ndarray*)
- **infos** (*list[dict[str, Any]]*)

Return type

None

extend(**args*, ***kwargs*)

Add a new batch of transitions to the buffer

Return type

None

reset()

Reset the buffer.

Return type
None

sample(*batch_size*, *env=None*)

Sample elements from the replay buffer.

Parameters

- **batch_size** (*int*) – Number of element to sample
- **env** (VecNormalize | *None*) – Associated VecEnv to normalize the observations/rewards when sampling

Returns
Samples

Return type
DictReplayBufferSamples

set_env(*env*)

Sets the environment.

Parameters
env (VecEnv)

Return type
None

size()

Returns
The current size of the buffer

Return type
int

static swap_and_flatten(*arr*)

Swap and then flatten axes 0 (buffer_size) and 1 (n_envs) to convert shape from [n_steps, n_envs, …] (when … is the shape of the features) to [n_steps * n_envs, …] (which maintain the order)

Parameters
arr (*ndarray*)

Returns

Return type
ndarray

to_torch(*array*, *copy=True*)

Convert a numpy array to a PyTorch tensor. Note: it copies the data by default

Parameters

- **array** (*ndarray*)
- **copy** (*bool*) – Whether to copy or not the data (may be useful to avoid changing things by reference). This argument is inoperative if the device is not the CPU.

Returns

Return type
Tensor

`truncate_last_trajectory()`

If called, we assume that the last trajectory in the replay buffer was finished (and truncate it). If not called, we assume that we continue the same trajectory (same episode).

Return type
None

1.26.7 Goal Selection Strategies

`class stable_baselines3.her.GoalSelectionStrategy(value)`

The strategies for selecting new goals when creating artificial transitions.

1.27 PPO

The Proximal Policy Optimization algorithm combines ideas from A2C (having multiple workers) and TRPO (it uses a trust region to improve the actor).

The main idea is that after an update, the new policy should be not too far from the old policy. For that, ppo uses clipping to avoid too large update.

Note

PPO contains several modifications from the original algorithm not documented by OpenAI: advantages are normalized and value function can be also clipped.

1.27.1 Notes

- Original paper: https://arxiv.org/abs/1707.06347
- Clear explanation of PPO on Arxiv Insights channel: https://www.youtube.com/watch?v=5P7I-xPq8u8
- OpenAI blog post: https://openai.com/research/openai-baselines-ppo
- Spinning Up guide: https://spinningup.openai.com/en/latest/algorithms/ppo.html
- 37 implementation details blog: https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/

1.27.2 Can I use?

Note

A recurrent version of PPO is available in our contrib repo: https://sb3-contrib.readthedocs.io/en/master/modules/ppo_recurrent.html

However we advise users to start with simple frame-stacking as a simpler, faster and usually competitive alternative, more info in our report: https://wandb.ai/sb3/no-vel-envs/reports/PPO-vs-RecurrentPPO-aka-PPO-LSTM-on-environments-with-masked-velocity–VmlldzoxOTI4NjE4 See also Procgen paper appendix Fig 11.. In practice, you can stack multiple observations using `VecFrameStack`.

- Recurrent policies:
- Multi processing: ✓

- Gym spaces:

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

1.27.3 Example

This example is only to demonstrate the use of the library and its functions, and the trained agents may not solve the environments. Optimized hyperparameters can be found in RL Zoo repository.

Train a PPO agent on `CartPole-v1` using 4 environments.

```python
import gymnasium as gym

from stable_baselines3 import PPO
from stable_baselines3.common.env_util import make_vec_env

# Parallel environments
vec_env = make_vec_env("CartPole-v1", n_envs=4)

model = PPO("MlpPolicy", vec_env, verbose=1)
model.learn(total_timesteps=25000)
model.save("ppo_cartpole")

del model # remove to demonstrate saving and loading

model = PPO.load("ppo_cartpole")

obs = vec_env.reset()
while True:
    action, _states = model.predict(obs)
    obs, rewards, dones, info = vec_env.step(action)
    vec_env.render("human")
```

Note

PPO is meant to be run primarily on the CPU, especially when you are not using a CNN. To improve CPU utilization, try turning off the GPU and using `SubprocVecEnv` instead of the default `DummyVecEnv`:

```python
from stable_baselines3 import PPO
from stable_baselines3.common.env_util import make_vec_env
from stable_baselines3.common.vec_env import SubprocVecEnv

if __name__=="__main__":
    env = make_vec_env("CartPole-v1", n_envs=8, vec_env_cls=SubprocVecEnv)
    model = PPO("MlpPolicy", env, device="cpu")
    model.learn(total_timesteps=25_000)
```

For more information, see *Vectorized Environments*, Issue #1245 or the Multiprocessing notebook.

1.27.4 Results

Atari Games

The complete learning curves are available in the associated PR #110.

PyBullet Environments

Results on the PyBullet benchmark (2M steps) using 6 seeds. The complete learning curves are available in the associated issue #48.

Note

Hyperparameters from the gSDE paper were used (as they are tuned for PyBullet envs).

Gaussian means that the unstructured Gaussian noise is used for exploration, *gSDE* (generalized State-Dependent Exploration) is used otherwise.

Environments	A2C	A2C	PPO	PPO
	Gaussian	gSDE	Gaussian	gSDE
HalfCheetah	2003 +/- 54	2032 +/- 122	1976 +/- 479	2826 +/- 45
Ant	2286 +/- 72	2443 +/- 89	2364 +/- 120	2782 +/- 76
Hopper	1627 +/- 158	1561 +/- 220	1567 +/- 339	2512 +/- 21
Walker2D	577 +/- 65	839 +/- 56	1230 +/- 147	2019 +/- 64

How to replicate the results?

Clone the rl-zoo repo:

```
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 --env $ENV_ID --eval-episodes 10 --eval-freq 10000
```

Plot the results (here for PyBullet envs only):

```
python scripts/all_plots.py -a ppo -e HalfCheetah Ant Hopper Walker2D -f logs/ -o logs/
↪ppo_results
python scripts/plot_from_file.py -i logs/ppo_results.pkl -latex -l PPO
```

1.27.5 Parameters

class stable_baselines3.ppo.**PPO**(*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*, *use_sde=False*, *sde_sample_freq=-1*, *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)

Paper: https://arxiv.org/abs/1707.06347 Code: This implementation borrows code from OpenAI Spinning Up (https://github.com/openai/spinningup/) https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail and Stable Baselines (PPO2 from https://github.com/hill-a/stable-baselines)

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. 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
- **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
- **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
- **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 *PPO 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

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

`learn`(*total_timesteps*, *callback=None*, *log_interval=1*, *tb_log_name='PPO'*, *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** (*SelfPPO*)

Returns
the trained model

Return type
SelfPPO

`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

1.27.6 PPO Policies

`stable_baselines3.ppo.`**`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

`stable_baselines3.ppo.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

`stable_baselines3.ppo.`**`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

1.28 SAC

Soft Actor Critic (SAC) Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor.

SAC is the successor of Soft Q-Learning SQL and incorporates the double Q-learning trick from TD3. A key feature of SAC, and a major difference with common RL algorithms, is that it is trained to maximize a trade-off between expected return and entropy, a measure of randomness in the policy.

Available Policies

MlpPolicy	alias of `SACPolicy`
CnnPolicy	Policy class (with both actor and critic) for SAC.

continues on next page

Table 3 – continued from previous page

`MultiInputPolicy`	Policy class (with both actor and critic) for SAC.

1.28.1 Notes

- Original paper: https://arxiv.org/abs/1801.01290
- OpenAI Spinning Guide for SAC: https://spinningup.openai.com/en/latest/algorithms/sac.html
- Original Implementation: https://github.com/haarnoja/sac
- Blog post on using SAC with real robots: https://bair.berkeley.edu/blog/2018/12/14/sac/

Note

In our implementation, we use an entropy coefficient (as in OpenAI Spinning or Facebook Horizon), which is the equivalent to the inverse of reward scale in the original SAC paper. The main reason is that it avoids having too high errors when updating the Q functions.

Note

When automatically adjusting the temperature (alpha/entropy coefficient), we optimize the logarithm of the entropy coefficient instead of the entropy coefficient itself. This is consistent with the original implementation and has proven to be more stable (see issues GH#36, #55 and others).

Note

The default policies for SAC differ a bit from others MlpPolicy: it uses ReLU instead of tanh activation, to match the original paper

1.28.2 Can I use?

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

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

1.28.3 Example

This example is only to demonstrate the use of the library and its functions, and the trained agents may not solve the environments. Optimized hyperparameters can be found in RL Zoo repository.

```python
import gymnasium as gym

from stable_baselines3 import SAC

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

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

del model # remove to demonstrate saving and loading

model = SAC.load("sac_pendulum")

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

1.28.4 Results

PyBullet Environments

Results on the PyBullet benchmark (1M steps) using 3 seeds. The complete learning curves are available in the associated issue #48.

Note

Hyperparameters from the gSDE paper were used (as they are tuned for PyBullet envs).

Gaussian means that the unstructured Gaussian noise is used for exploration, *gSDE* (generalized State-Dependent Exploration) is used otherwise.

Environments	SAC	SAC	TD3
	Gaussian	gSDE	Gaussian
HalfCheetah	2757 +/- 53	2984 +/- 202	2774 +/- 35
Ant	3146 +/- 35	3102 +/- 37	3305 +/- 43
Hopper	2422 +/- 168	2262 +/- 1	2429 +/- 126
Walker2D	2184 +/- 54	2136 +/- 67	2063 +/- 185

How to replicate the results?

Clone the rl-zoo repo:

```
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 sac --env $ENV_ID --eval-episodes 10 --eval-freq 10000
```

Plot the results:

```
python scripts/all_plots.py -a sac -e HalfCheetah Ant Hopper Walker2D -f logs/ -o logs/
↪sac_results
python scripts/plot_from_file.py -i logs/sac_results.pkl -latex -l SAC
```

1.28.5 Parameters

class stable_baselines3.sac.**SAC**(*policy*, *env*, *learning_rate=0.0003*, *buffer_size=1000000*, *learning_starts=100*, *batch_size=256*, *tau=0.005*, *gamma=0.99*, *train_freq=1*, *gradient_steps=1*, *action_noise=None*, *replay_buffer_class=None*, *replay_buffer_kwargs=None*, *optimize_memory_usage=False*, *ent_coef='auto'*, *target_update_interval=1*, *target_entropy='auto'*, *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*)

Soft Actor-Critic (SAC) Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor, This implementation borrows code from original implementation (https://github.com/haarnoja/sac) from OpenAI Spinning Up (https://github.com/openai/spinningup), from the softlearning repo (https://github.com/rail-berkeley/softlearning/) and from Stable Baselines (https://github.com/hill-a/stable-baselines) Paper: https://arxiv.org/abs/1801.01290 Introduction to SAC: https://spinningup.openai.com/en/latest/algorithms/sac.html

Note: we use double q target and not value target as discussed in https://github.com/hill-a/stable-baselines/issues/270

Parameters

- **policy** (*SACPolicy*) – 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]*) – 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 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
- **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'`)
- **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 *SAC 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

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='SAC'*, *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** (*SelfSAC*)

Returns
the trained model

Return type
SelfSAC

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

1.28.6 SAC Policies

`stable_baselines3.sac.`**`MlpPolicy`**

alias of `SACPolicy`

`class` `stable_baselines3.sac.policies.`**`SACPolicy`**(*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_critics=2*, *share_features_extractor=False*)

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

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 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

class stable_baselines3.sac.**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_critics=2*, *share_features_extractor=False*)

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

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_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*)

class stable_baselines3.sac.**MultiInputPolicy**(*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.CombinedExtractor'>*, *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 SAC.

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_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*)

1.29 TD3

Twin Delayed DDPG (TD3) Addressing Function Approximation Error in Actor-Critic Methods.

TD3 is a direct successor of *DDPG* and improves it using three major tricks: clipped double Q-Learning, delayed policy update and target policy smoothing. We recommend reading OpenAI Spinning guide on TD3 to learn more about those.

Available Policies

MlpPolicy	alias of `TD3Policy`
CnnPolicy	Policy class (with both actor and critic) for TD3.
MultiInputPolicy	Policy class (with both actor and critic) for TD3 to be used with Dict observation spaces.

1.29.1 Notes

- Original paper: https://arxiv.org/pdf/1802.09477.pdf
- OpenAI Spinning Guide for TD3: https://spinningup.openai.com/en/latest/algorithms/td3.html
- Original Implementation: https://github.com/sfujim/TD3

Note

The default policies for TD3 differ a bit from others MlpPolicy: it uses ReLU instead of tanh activation, to match the original paper

1.29.2 Can I use?

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

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

1.29.3 Example

This example is only to demonstrate the use of the library and its functions, and the trained agents may not solve the environments. Optimized hyperparameters can be found in RL Zoo repository.

```
import gymnasium as gym
import numpy as np

from stable_baselines3 import TD3
from stable_baselines3.common.noise import NormalActionNoise,␣
↪OrnsteinUhlenbeckActionNoise

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

# The noise objects for TD3
n_actions = env.action_space.shape[-1]
action_noise = NormalActionNoise(mean=np.zeros(n_actions), sigma=0.1 * np.ones(n_
↪actions))

model = TD3("MlpPolicy", env, action_noise=action_noise, verbose=1)
model.learn(total_timesteps=10000, log_interval=10)
model.save("td3_pendulum")
vec_env = model.get_env()

del model # remove to demonstrate saving and loading

model = TD3.load("td3_pendulum")

obs = vec_env.reset()
while True:
    action, _states = model.predict(obs)
    obs, rewards, dones, info = vec_env.step(action)
    vec_env.render("human")
```

1.29.4 Results

PyBullet Environments

Results on the PyBullet benchmark (1M steps) using 3 seeds. The complete learning curves are available in the associated issue #48.

> **Note**
>
> Hyperparameters from the gSDE paper were used (as they are tuned for PyBullet envs).

Gaussian means that the unstructured Gaussian noise is used for exploration, *gSDE* (generalized State-Dependent Exploration) is used otherwise.

Environments	SAC	SAC	TD3
	Gaussian	gSDE	Gaussian
HalfCheetah	2757 +/- 53	2984 +/- 202	2774 +/- 35
Ant	3146 +/- 35	3102 +/- 37	3305 +/- 43
Hopper	2422 +/- 168	2262 +/- 1	2429 +/- 126
Walker2D	2184 +/- 54	2136 +/- 67	2063 +/- 185

How to replicate the results?

Clone the rl-zoo repo:

```
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 td3 --env $ENV_ID --eval-episodes 10 --eval-freq 10000
```

Plot the results:

```
python scripts/all_plots.py -a td3 -e HalfCheetah Ant Hopper Walker2D -f logs/ -o logs/
↪td3_results
python scripts/plot_from_file.py -i logs/td3_results.pkl -latex -l TD3
```

1.29.5 Parameters

class `stable_baselines3.td3.`**`TD3`**(*policy*, *env*, *learning_rate=0.001*, *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*, *policy_delay=2*, *target_policy_noise=0.2*, *target_noise_clip=0.5*, *stats_window_size=100*, *tensorboard_log=None*, *policy_kwargs=None*, *verbose=0*, *seed=None*, *device='auto'*, *_init_setup_model=True*)

Twin Delayed DDPG (TD3) Addressing Function Approximation Error in Actor-Critic Methods.

Original implementation: https://github.com/sfujim/TD3 Paper: https://arxiv.org/abs/1802.09477 Introduction to TD3: https://spinningup.openai.com/en/latest/algorithms/td3.html

Parameters

- **policy** (*TD3Policy*) – 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]*) – 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 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
- **policy_delay** (*int*) – Policy and target networks will only be updated once every policy_delay steps per training steps. The Q values will be updated policy_delay more often (update every training step).
- **target_policy_noise** (*float*) – Standard deviation of Gaussian noise added to target policy (smoothing noise)
- **target_noise_clip** (*float*) – Limit for absolute value of target policy smoothing noise.
- **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 *TD3 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

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='TD3'*, *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** (*SelfTD3*)

Returns
the trained model

Return type
SelfTD3

`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

1.29.6 TD3 Policies

stable_baselines3.td3.**MlpPolicy**

alias of `TD3Policy`

class stable_baselines3.td3.policies.**TD3Policy**(*observation_space*, *action_space*, *lr_schedule*, *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*, *n_critics=2*, *share_features_extractor=False*)

Policy class (with both actor and critic) for TD3.

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
- **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(*observation*, *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

- **observation** (*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.

Parameters

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

Return type

None

class stable_baselines3.td3.**CnnPolicy**(*observation_space*, *action_space*, *lr_schedule*, *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*, *n_critics=2*, *share_features_extractor=False*)

Policy class (with both actor and critic) for TD3.

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
- **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)

class stable_baselines3.td3.**MultiInputPolicy**(*observation_space*, *action_space*, *lr_schedule*, *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*, *n_critics=2*, *share_features_extractor=False*)

Policy class (with both actor and critic) for TD3 to be used with Dict observation spaces.

Parameters

- **observation_space** (*Dict*) – 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
- **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)

1.30 Atari Wrappers

class stable_baselines3.common.atari_wrappers.**AtariWrapper**(*env*, *noop_max=30*, *frame_skip=4*, *screen_size=84*, *terminal_on_life_loss=True*, *clip_reward=True*, *action_repeat_probability=0.0*)

Atari 2600 preprocessings

Specifically:

- Noop reset: obtain initial state by taking random number of no-ops on reset.
- Frame skipping: 4 by default
- Max-pooling: most recent two observations
- Termination signal when a life is lost.
- Resize to a square image: 84x84 by default
- Grayscale observation
- Clip reward to {-1, 0, 1}
- Sticky actions: disabled by default

See https://danieltakeshi.github.io/2016/11/25/frame-skipping-and-preprocessing-for-deep-q-networks-on-atari-2600-games/ for a visual explanation.

> **Warning**
>
> Use this wrapper only with Atari v4 without frame skip: `env_id = "*NoFrameskip-v4"`.

Parameters

- **env** (*Env*) – Environment to wrap
- **noop_max** (*int*) – Max number of no-ops
- **frame_skip** (*int*) – Frequency at which the agent experiences the game. This correspond to repeating the action `frame_skip` times.
- **screen_size** (*int*) – Resize Atari frame
- **terminal_on_life_loss** (*bool*) – If True, then step() returns done=True whenever a life is lost.
- **clip_reward** (*bool*) – If True (default), the reward is clip to {-1, 0, 1} depending on its sign.
- **action_repeat_probability** (*float*) – Probability of repeating the last action

class stable_baselines3.common.atari_wrappers.**ClipRewardEnv**(*env*)

Clip the reward to {+1, 0, -1} by its sign.

Parameters
`env` (*Env*) – Environment to wrap

`reward`(*reward*)

Bin reward to {+1, 0, -1} by its sign.

Parameters
`reward` (*SupportsFloat*)

Returns

Return type
float

`class stable_baselines3.common.atari_wrappers.`**`EpisodicLifeEnv`**(*env*)

Make end-of-life == end-of-episode, but only reset on true game over. Done by DeepMind for the DQN and co. since it helps value estimation.

Parameters
`env` (*Env*) – Environment to wrap

`reset`(***kwargs*)

Calls the Gym environment reset, only when lives are exhausted. This way all states are still reachable even though lives are episodic, and the learner need not know about any of this behind-the-scenes.

Parameters
`kwargs` – Extra keywords passed to env.reset() call

Returns
the first observation of the environment

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

`step`(*action*)

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

Parameters
`action` (*int*)

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

`class stable_baselines3.common.atari_wrappers.`**`FireResetEnv`**(*env*)

Take action on reset for environments that are fixed until firing.

Parameters
`env` (*Env*) – Environment to wrap

`reset`(***kwargs*)

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

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

`class stable_baselines3.common.atari_wrappers.`**`MaxAndSkipEnv`**(*env*, *skip=4*)

Return only every `skip`-th frame (frameskipping) and return the max between the two last frames.

Parameters

- `env` (*Env*) – Environment to wrap
- `skip` (*int*) – Number of `skip`-th frame The same action will be taken `skip` times.

`step`(*action*)

Step the environment with the given action Repeat action, sum reward, and max over last observations.

Parameters
`action` (*int*) – the action

Returns
observation, reward, terminated, truncated, information

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

`class stable_baselines3.common.atari_wrappers.NoopResetEnv`(*env*, *noop_max=30*)

Sample initial states by taking random number of no-ops on reset. No-op is assumed to be action 0.

Parameters

- **env** (*Env*) – Environment to wrap
- **noop_max** (*int*) – Maximum value of no-ops to run

`reset`(***kwargs*)

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

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

`class stable_baselines3.common.atari_wrappers.StickyActionEnv`(*env*, *action_repeat_probability*)

Sticky action.

Paper: https://arxiv.org/abs/1709.06009 Official implementation: https://github.com/mgbellemare/Arcade-Learning-Environment

Parameters

- **env** (*Env*) – Environment to wrap
- **action_repeat_probability** (*float*) – Probability of repeating the last action

`reset`(***kwargs*)

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

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

`step`(*action*)

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

Parameters
`action` (*int*)

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

`class stable_baselines3.common.atari_wrappers.WarpFrame`(*env*, *width=84*, *height=84*)

Convert to grayscale and warp frames to 84x84 (default) as done in the Nature paper and later work.

Parameters

- **env** (*Env*) – Environment to wrap
- **width** (*int*) – New frame width
- **height** (*int*) – New frame height

`observation`(*frame*)

returns the current observation from a frame

Parameters
`frame` (*ndarray*) – environment frame

Returns
the observation

Return type
ndarray

1.31 Environments Utils

`stable_baselines3.common.env_util.is_wrapped`(*env*, *wrapper_class*)

Check if a given environment has been wrapped with a given wrapper.

Parameters

- `env` (*Env*) – Environment to check
- `wrapper_class` (*type[Wrapper]*) – Wrapper class to look for

Returns
True if environment has been wrapped with `wrapper_class`.

Return type
bool

`stable_baselines3.common.env_util.make_atari_env`(*env_id*, *n_envs=1*, *seed=None*, *start_index=0*, *monitor_dir=None*, *wrapper_kwargs=None*, *env_kwargs=None*, *vec_env_cls=None*, *vec_env_kwargs=None*, *monitor_kwargs=None*)

Create a wrapped, monitored VecEnv for Atari. It is a wrapper around `make_vec_env` that includes common preprocessing for Atari games.

Parameters

- `env_id` (*str | Callable[[...], Env]*) – either the env ID, the env class or a callable returning an env
- `n_envs` (*int*) – the number of environments you wish to have in parallel
- `seed` (*int | None*) – the initial seed for the random number generator
- `start_index` (*int*) – start rank index
- `monitor_dir` (*str | None*) – Path to a folder where the monitor files will be saved. If None, no file will be written, however, the env will still be wrapped in a Monitor wrapper to provide additional information about training.
- `wrapper_kwargs` (*dict[str, Any] | None*) – Optional keyword argument to pass to the `AtariWrapper`
- `env_kwargs` (*dict[str, Any] | None*) – Optional keyword argument to pass to the env constructor
- `vec_env_cls` (*type[DummyVecEnv] | type[SubprocVecEnv] | None*) – A custom `VecEnv` class constructor. Default: None.
- `vec_env_kwargs` (*dict[str, Any] | None*) – Keyword arguments to pass to the `VecEnv` class constructor.

- **monitor_kwargs** (*dict[str, Any] | None*) – Keyword arguments to pass to the `Monitor` class constructor.

Returns
The wrapped environment

Return type
VecEnv

`stable_baselines3.common.env_util.`**`make_vec_env`**(*env_id*, *n_envs=1*, *seed=None*, *start_index=0*, *monitor_dir=None*, *wrapper_class=None*, *env_kwargs=None*, *vec_env_cls=None*, *vec_env_kwargs=None*, *monitor_kwargs=None*, *wrapper_kwargs=None*)

Create a wrapped, monitored `VecEnv`. By default it uses a `DummyVecEnv` which is usually faster than a `SubprocVecEnv`.

Parameters

- **env_id** (*str | Callable[[...], Env]*) – either the env ID, the env class or a callable returning an env
- **n_envs** (*int*) – the number of environments you wish to have in parallel
- **seed** (*int | None*) – the initial seed for the random number generator
- **start_index** (*int*) – start rank index
- **monitor_dir** (*str | None*) – Path to a folder where the monitor files will be saved. If None, no file will be written, however, the env will still be wrapped in a Monitor wrapper to provide additional information about training.
- **wrapper_class** (*Callable[[Env], Env] | None*) – Additional wrapper to use on the environment. This can also be a function with single argument that wraps the environment in many things. Note: the wrapper specified by this parameter will be applied after the `Monitor` wrapper. if some cases (e.g. with TimeLimit wrapper) this can lead to undesired behavior. See here for more details: https://github.com/DLR-RM/stable-baselines3/issues/894
- **env_kwargs** (*dict[str, Any] | None*) – Optional keyword argument to pass to the env constructor
- **vec_env_cls** (*type[DummyVecEnv | SubprocVecEnv] | None*) – A custom `VecEnv` class constructor. Default: None.
- **vec_env_kwargs** (*dict[str, Any] | None*) – Keyword arguments to pass to the `VecEnv` class constructor.
- **monitor_kwargs** (*dict[str, Any] | None*) – Keyword arguments to pass to the `Monitor` class constructor.
- **wrapper_kwargs** (*dict[str, Any] | None*) – Keyword arguments to pass to the `Wrapper` class constructor.

Returns
The wrapped environment

Return type
VecEnv

`stable_baselines3.common.env_util.`**`unwrap_wrapper`**(*env*, *wrapper_class*)

Retrieve a `VecEnvWrapper` object by recursively searching.

Parameters

- **env** (*Env*) – Environment to unwrap
- **wrapper_class** (*type[Wrapper]*) – Wrapper to look for

Returns
Environment unwrapped till `wrapper_class` if it has been wrapped with it

Return type
Wrapper | None

1.32 Custom Environments

Those environments were created for testing purposes.

1.32.1 BitFlippingEnv

class stable_baselines3.common.envs.**BitFlippingEnv**(*n_bits=10*, *continuous=False*, *max_steps=None*, *discrete_obs_space=False*, *image_obs_space=False*, *channel_first=True*, *render_mode='human'*)

Simple bit flipping env, useful to test HER. The goal is to flip all the bits to get a vector of ones. In the continuous variant, if the ith action component has a value > 0, then the ith bit will be flipped. Uses a `MultiBinary` observation space by default.

Parameters

- **n_bits** (*int*) – Number of bits to flip
- **continuous** (*bool*) – Whether to use the continuous actions version or not, by default, it uses the discrete one
- **max_steps** (*int | None*) – Max number of steps, by default, equal to n_bits
- **discrete_obs_space** (*bool*) – Whether to use the discrete observation version or not, ie a one-hot encoding of all possible states
- **image_obs_space** (*bool*) – Whether to use an image observation version or not, ie a greyscale image of the state
- **channel_first** (*bool*) – Whether to use channel-first or last image.
- **render_mode** (*str*)

close()

After the user has finished using the environment, close contains the code necessary to “clean up” the environment.

This is critical for closing rendering windows, database or HTTP connections. Calling `close` on an already closed environment has no effect and won’t raise an error.

Return type
None

convert_if_needed(*state*)

Convert to discrete space if needed.

Parameters
state (*ndarray*)

Returns

Return type
int | *ndarray*

`convert_to_bit_vector`(*state*, *batch_size*)

Convert to bit vector if needed.

Parameters

- **state** (*int | ndarray*) – The state to be converted, which can be either an integer or a numpy array.
- **batch_size** (*int*) – The batch size.

Returns
The state converted into a bit vector.

Return type
ndarray

`render`()

Compute the render frames as specified by `render_mode` during the initialization of the environment.

The environment's `metadata` render modes (*env.metadata["render_modes"]*) should contain the possible ways to implement the render modes. In addition, list versions for most render modes is achieved through *gymnasium.make* which automatically applies a wrapper to collect rendered frames.

Note:
As the `render_mode` is known during `__init__`, the objects used to render the environment state should be initialised in `__init__`.

By convention, if the `render_mode` is:

- None (default): no render is computed.
- "human": The environment is continuously rendered in the current display or terminal, usually for human consumption. This rendering should occur during *`step()`* and *`render()`* doesn't need to be called. Returns `None`.
- "rgb_array": Return a single frame representing the current state of the environment. A frame is a `np.ndarray` with shape `(x, y, 3)` representing RGB values for an x-by-y pixel image.
- "ansi": Return a strings (`str`) or `StringIO.StringIO` containing a terminal-style text representation for each time step. The text can include newlines and ANSI escape sequences (e.g. for colors).
- "rgb_array_list" and "ansi_list": List based version of render modes are possible (except Human) through the wrapper, `gymnasium.wrappers.RenderCollection` that is automatically applied during `gymnasium.make(..., render_mode="rgb_array_list")`. The frames collected are popped after *`render()`* is called or *`reset()`*.

Note:
Make sure that your class's `metadata` `"render_modes"` key includes the list of supported modes.

Changed in version 0.25.0: The render function was changed to no longer accept parameters, rather these parameters should be specified in the environment initialised, i.e., `gymnasium.make("CartPole-v1", render_mode="human")`

Return type
ndarray | None

`reset`(** (Keyword-only parameters separator (PEP 3102))*, *seed=None*, *options=None*)

Resets the environment to an initial internal state, returning an initial observation and info.

This method generates a new starting state often with some randomness to ensure that the agent explores the state space and learns a generalised policy about the environment. This randomness can be controlled with the `seed` parameter otherwise if the environment already has a random number generator and `reset()` is called with `seed=None`, the RNG is not reset.

Therefore, `reset()` should (in the typical use case) be called with a seed right after initialization and then never again.

For Custom environments, the first line of `reset()` should be `super().reset(seed=seed)` which implements the seeding correctly.

Changed in version v0.25: The `return_info` parameter was removed and now info is expected to be returned.

Args:

seed (optional int): The seed that is used to initialize the environment's PRNG (*np_random*) and
the read-only attribute *np_random_seed*. If the environment does not already have a PRNG and `seed=None` (the default option) is passed, a seed will be chosen from some source of entropy (e.g. timestamp or /dev/urandom). However, if the environment already has a PRNG and `seed=None` is passed, the PRNG will *not* be reset and the env's `np_random_seed` will *not* be altered. If you pass an integer, the PRNG will be reset even if it already exists. Usually, you want to pass an integer *right after the environment has been initialized and then never again*. Please refer to the minimal example above to see this paradigm in action.

options (optional dict): Additional information to specify how the environment is reset (optional,
depending on the specific environment)

Returns:

observation (ObsType): Observation of the initial state. This will be an element of `observation_space`
(typically a numpy array) and is analogous to the observation returned by `step()`.

info (dictionary): This dictionary contains auxiliary information complementing `observation`. It should be analogous to
the `info` returned by `step()`.

Parameters

- **seed** (*int | None*)
- **options** (*dict | None*)

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

`step`(*action*)

Step into the env.

Parameters
action (*ndarray | int*)

Returns

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

1.32.2 SimpleMultiObsEnv

class stable_baselines3.common.envs.**SimpleMultiObsEnv**(*num_col=4*, *num_row=4*, *random_start=True*, *discrete_actions=True*, *channel_last=True*)

Base class for GridWorld-based MultiObs Environments 4x4 grid world.

```
 ____________
| 0  1  2   3|
| 4|¯5¯¯6¯| 7|
| 8|_9_10_|11|
|12 13  14 15|
--------------
```

start is 0 states 5, 6, 9, and 10 are blocked goal is 15 actions are = [left, down, right, up]

simple linear state env of 15 states but encoded with a vector and an image observation: each column is represented by a random vector and each row is represented by a random image, both sampled once at creation time.

Parameters

- **num_col** (*int*) – Number of columns in the grid
- **num_row** (*int*) – Number of rows in the grid
- **random_start** (*bool*) – If true, agent starts in random position
- **channel_last** (*bool*) – If true, the image will be channel last, else it will be channel first
- **discrete_actions** (*bool*)

get_state_mapping()

Uses the state to get the observation mapping.

Returns

observation dict {'vec': …, 'img': …}

Return type

dict[str, *ndarray*]

init_possible_transitions()

Initializes the transitions of the environment The environment exploits the cardinal directions of the grid by noting that they correspond to simple addition and subtraction from the cell id within the grid

- up => means moving up a row => means subtracting the length of a column
- down => means moving down a row => means adding the length of a column
- left => means moving left by one => means subtracting 1
- right => means moving right by one => means adding 1

Thus one only needs to specify in which states each action is possible in order to define the transitions of the environment

Return type

None

init_state_mapping(*num_col*, *num_row*)

Initializes the state_mapping array which holds the observation values for each state

Parameters

- **num_col** (*int*) – Number of columns.
- **num_row** (*int*) – Number of rows.

Return type
None

`render`(*mode='human'*)

Prints the log of the environment.

Parameters
mode (*str*)

Return type
None

`reset`(***, *seed=None*, *options=None*)

Resets the environment state and step count and returns reset observation.

Parameters

- **seed** (*int | None*)
- **options** (*dict | None*)

Returns
observation dict {'vec': …, 'img': …}

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

`step`(*action*)

Run one timestep of the environment's dynamics. When end of episode is reached, you are responsible for calling *reset()* to reset this environment's state. Accepts an action and returns a tuple (observation, reward, terminated, truncated, info).

Parameters
action (*int | ndarray*)

Returns
tuple (observation, reward, terminated, truncated, info).

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

1.33 Probability Distributions

Probability distributions used for the different action spaces:

- `CategoricalDistribution` -> Discrete
- `DiagGaussianDistribution` -> Box (continuous actions)
- `StateDependentNoiseDistribution` -> Box (continuous actions) when `use_sde=True`

The policy networks output parameters for the distributions (named `flat` in the methods). Actions are then sampled from those distributions.

For instance, in the case of discrete actions. The policy network outputs probability of taking each action. The `CategoricalDistribution` allows sampling from it, computes the entropy, the log probability (`log_prob`) and backpropagate the gradient.

In the case of continuous actions, a Gaussian distribution is used. The policy network outputs mean and (log) std of the distribution (assumed to be a `DiagGaussianDistribution`). Probability distributions.

class `stable_baselines3.common.distributions.`**`BernoulliDistribution`**(*action_dims*)

Bernoulli distribution for MultiBinary action spaces.

Parameters

- **`action_dim`** – Number of binary actions
- **`action_dims`** (*`int`*)

`actions_from_params`(*action_logits*, *deterministic=False*)

Returns samples from the probability distribution given its parameters.

Returns

actions

Parameters

- **`action_logits`** (*`Tensor`*)
- **`deterministic`** (*`bool`*)

Return type

Tensor

`entropy`()

Returns Shannon's entropy of the probability

Returns

the entropy, or None if no analytical form is known

Return type

Tensor

`log_prob`(*actions*)

Returns the log likelihood

Parameters

- **`x`** – the taken action
- **`actions`** (*`Tensor`*)

Returns

The log likelihood of the distribution

Return type

Tensor

`log_prob_from_params`(*action_logits*)

Returns samples and the associated log probabilities from the probability distribution given its parameters.

Returns

actions and log prob

Parameters

`action_logits` (*`Tensor`*)

Return type

tuple[*Tensor*, *Tensor*]

mode()

Returns the most likely action (deterministic output) from the probability distribution

Returns
the stochastic action

Return type
Tensor

proba_distribution(*action_logits*)

Set parameters of the distribution.

Returns
self

Parameters

- **self** (*SelfBernoulliDistribution*)
- **action_logits** (*Tensor*)

Return type
SelfBernoulliDistribution

proba_distribution_net(*latent_dim*)

Create the layer that represents the distribution: it will be the logits of the Bernoulli distribution.

Parameters
latent_dim (*int*) – Dimension of the last layer of the policy network (before the action layer)

Returns

Return type
Module

sample()

Returns a sample from the probability distribution

Returns
the stochastic action

Return type
Tensor

class stable_baselines3.common.distributions.**CategoricalDistribution**(*action_dim*)

Categorical distribution for discrete actions.

Parameters
action_dim (*int*) – Number of discrete actions

actions_from_params(*action_logits*, *deterministic=False*)

Returns samples from the probability distribution given its parameters.

Returns
actions

Parameters

- **action_logits** (*Tensor*)
- **deterministic** (*bool*)

Return type
Tensor

`entropy()`

Returns Shannon's entropy of the probability

Returns
the entropy, or None if no analytical form is known

Return type
Tensor

`log_prob(`*actions*`)`

Returns the log likelihood

Parameters

- **x** – the taken action
- **actions** (*Tensor*)

Returns
The log likelihood of the distribution

Return type
Tensor

`log_prob_from_params(`*action_logits*`)`

Returns samples and the associated log probabilities from the probability distribution given its parameters.

Returns
actions and log prob

Parameters
action_logits (*Tensor*)

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

`mode()`

Returns the most likely action (deterministic output) from the probability distribution

Returns
the stochastic action

Return type
Tensor

`proba_distribution(`*action_logits*`)`

Set parameters of the distribution.

Returns
self

Parameters

- **self** (*SelfCategoricalDistribution*)
- **action_logits** (*Tensor*)

Return type
SelfCategoricalDistribution

`proba_distribution_net`(*latent_dim*)

Create the layer that represents the distribution: it will be the logits of the Categorical distribution. You can then get probabilities using a softmax.

Parameters
`latent_dim` (*int*) – Dimension of the last layer of the policy network (before the action layer)

Returns

Return type
Module

`sample`()

Returns a sample from the probability distribution

Returns
the stochastic action

Return type
Tensor

class `stable_baselines3.common.distributions.DiagGaussianDistribution`(*action_dim*)

Gaussian distribution with diagonal covariance matrix, for continuous actions.

Parameters
`action_dim` (*int*) – Dimension of the action space.

`actions_from_params`(*mean_actions*, *log_std*, *deterministic=False*)

Returns samples from the probability distribution given its parameters.

Returns
actions

Parameters

- `mean_actions` (*Tensor*)
- `log_std` (*Tensor*)
- `deterministic` (*bool*)

Return type
Tensor

`entropy`()

Returns Shannon's entropy of the probability

Returns
the entropy, or None if no analytical form is known

Return type
Tensor | None

`log_prob`(*actions*)

Get the log probabilities of actions according to the distribution. Note that you must first call the `proba_distribution()` method.

Parameters
`actions` (*Tensor*)

Returns

Return type
Tensor

`log_prob_from_params`(*mean_actions*, *log_std*)

Compute the log probability of taking an action given the distribution parameters.

Parameters

- `mean_actions` (*Tensor*)
- `log_std` (*Tensor*)

Returns

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

`mode`()

Returns the most likely action (deterministic output) from the probability distribution

Returns
the stochastic action

Return type
Tensor

`proba_distribution`(*mean_actions*, *log_std*)

Create the distribution given its parameters (mean, std)

Parameters

- `mean_actions` (*Tensor*)
- `log_std` (*Tensor*)
- `self` (*SelfDiagGaussianDistribution*)

Returns

Return type
SelfDiagGaussianDistribution

`proba_distribution_net`(*latent_dim*, *log_std_init=0.0*)

Create the layers and parameter that represent the distribution: one output will be the mean of the Gaussian, the other parameter will be the standard deviation (log std in fact to allow negative values)

Parameters

- `latent_dim` (*int*) – Dimension of the last layer of the policy (before the action layer)
- `log_std_init` (*float*) – Initial value for the log standard deviation

Returns

Return type
tuple[*Module*, *Parameter*]

`sample`()

Returns a sample from the probability distribution

Returns
the stochastic action

Return type
Tensor

class stable_baselines3.common.distributions.**Distribution**

Abstract base class for distributions.

abstractmethod actions_from_params(**args*, ***kwargs*)

Returns samples from the probability distribution given its parameters.

Returns
actions

Return type
Tensor

abstractmethod entropy()

Returns Shannon's entropy of the probability

Returns
the entropy, or None if no analytical form is known

Return type
Tensor | None

get_actions(*deterministic=False*)

Return actions according to the probability distribution.

Parameters
deterministic (*bool*)

Returns

Return type
Tensor

abstractmethod log_prob(*x*)

Returns the log likelihood

Parameters
x (*Tensor*) – the taken action

Returns
The log likelihood of the distribution

Return type
Tensor

abstractmethod log_prob_from_params(**args*, ***kwargs*)

Returns samples and the associated log probabilities from the probability distribution given its parameters.

Returns
actions and log prob

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

abstractmethod mode()

Returns the most likely action (deterministic output) from the probability distribution

Returns
the stochastic action

Return type
Tensor

abstractmethod proba_distribution(**args*, ***kwargs*)

Set parameters of the distribution.

Returns
self

Parameters
self (*SelfDistribution*)

Return type
SelfDistribution

abstractmethod proba_distribution_net(**args*, ***kwargs*)

Create the layers and parameters that represent the distribution.

Subclasses must define this, but the arguments and return type vary between concrete classes.

Return type
Module | tuple[*Module*, *Parameter*]

abstractmethod sample()

Returns a sample from the probability distribution

Returns
the stochastic action

Return type
Tensor

class stable_baselines3.common.distributions.**MultiCategoricalDistribution**(*action_dims*)

MultiCategorical distribution for multi discrete actions.

Parameters
action_dims (*list[int]*) – List of sizes of discrete action spaces

actions_from_params(*action_logits*, *deterministic=False*)

Returns samples from the probability distribution given its parameters.

Returns
actions

Parameters

- **action_logits** (*Tensor*)
- **deterministic** (*bool*)

Return type
Tensor

entropy()

Returns Shannon's entropy of the probability

Returns
the entropy, or None if no analytical form is known

Return type
Tensor

log_prob(*actions*)

Returns the log likelihood

Parameters

- **x** – the taken action
- **actions** (*Tensor*)

Returns
The log likelihood of the distribution

Return type
Tensor

log_prob_from_params(*action_logits*)

Returns samples and the associated log probabilities from the probability distribution given its parameters.

Returns
actions and log prob

Parameters
action_logits (*Tensor*)

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

mode()

Returns the most likely action (deterministic output) from the probability distribution

Returns
the stochastic action

Return type
Tensor

proba_distribution(*action_logits*)

Set parameters of the distribution.

Returns
self

Parameters

- **self** (*SelfMultiCategoricalDistribution*)
- **action_logits** (*Tensor*)

Return type
SelfMultiCategoricalDistribution

proba_distribution_net(*latent_dim*)

Create the layer that represents the distribution: it will be the logits (flattened) of the MultiCategorical distribution. You can then get probabilities using a softmax on each sub-space.

Parameters
latent_dim (*int*) – Dimension of the last layer of the policy network (before the action layer)

Returns

Return type
Module

sample()

Returns a sample from the probability distribution

Returns
the stochastic action

Return type
Tensor

`class stable_baselines3.common.distributions.SquashedDiagGaussianDistribution`(*action_dim*, *epsilon=1e-06*)

Gaussian distribution with diagonal covariance matrix, followed by a squashing function (tanh) to ensure bounds.

Parameters

- **action_dim** (*int*) – Dimension of the action space.
- **epsilon** (*float*) – small value to avoid NaN due to numerical imprecision.

`entropy()`

Returns Shannon's entropy of the probability

Returns
the entropy, or None if no analytical form is known

Return type
Tensor | None

`log_prob`(*actions*, *gaussian_actions=None*)

Get the log probabilities of actions according to the distribution. Note that you must first call the `proba_distribution()` method.

Parameters

- **actions** (*Tensor*)
- **gaussian_actions** (*Tensor | None*)

Returns

Return type
Tensor

`log_prob_from_params`(*mean_actions*, *log_std*)

Compute the log probability of taking an action given the distribution parameters.

Parameters

- **mean_actions** (*Tensor*)
- **log_std** (*Tensor*)

Returns

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

`mode()`

Returns the most likely action (deterministic output) from the probability distribution

Returns
the stochastic action

Return type
Tensor

`proba_distribution`(*mean_actions*, *log_std*)

Create the distribution given its parameters (mean, std)

Parameters

- **mean_actions** (*Tensor*)
- **log_std** (*Tensor*)
- **self** (*SelfSquashedDiagGaussianDistribution*)

Returns

Return type
SelfSquashedDiagGaussianDistribution

`sample()`

Returns a sample from the probability distribution

Returns
the stochastic action

Return type
Tensor

`class stable_baselines3.common.distributions.StateDependentNoiseDistribution`(*action_dim*, *full_std=True*, *use_expln=False*, *squash_output=False*, *learn_features=False*, *epsilon=1e-06*)

Distribution class for using generalized State Dependent Exploration (gSDE). Paper: https://arxiv.org/abs/2005.05719

It is used to create the noise exploration matrix and compute the log probability of an action with that noise.

Parameters

- **action_dim** (*int*) – Dimension of the action space.
- **full_std** (*bool*) – Whether to use (n_features x n_actions) parameters for the std instead of only (n_features,)
- **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 ensures bounds are satisfied.
- **learn_features** (*bool*) – Whether to learn features for gSDE or not. This will enable gradients to be backpropagated through the features `latent_sde` in the code.
- **epsilon** (*float*) – small value to avoid NaN due to numerical imprecision.

`actions_from_params`(*mean_actions*, *log_std*, *latent_sde*, *deterministic=False*)

Returns samples from the probability distribution given its parameters.

Returns
actions

Parameters

- **mean_actions** (*Tensor*)
- **log_std** (*Tensor*)
- **latent_sde** (*Tensor*)

- **deterministic** (*bool*)

Return type
Tensor

entropy()

Returns Shannon's entropy of the probability

Returns
the entropy, or None if no analytical form is known

Return type
Tensor | None

get_std(*log_std*)

Get the standard deviation from the learned parameter (log of it by default). This ensures that the std is positive.

Parameters
log_std (*Tensor*)

Returns

Return type
Tensor

log_prob(*actions*)

Returns the log likelihood

Parameters

- **x** – the taken action
- **actions** (*Tensor*)

Returns
The log likelihood of the distribution

Return type
Tensor

log_prob_from_params(*mean_actions*, *log_std*, *latent_sde*)

Returns samples and the associated log probabilities from the probability distribution given its parameters.

Returns
actions and log prob

Parameters

- **mean_actions** (*Tensor*)
- **log_std** (*Tensor*)
- **latent_sde** (*Tensor*)

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

mode()

Returns the most likely action (deterministic output) from the probability distribution

Returns
the stochastic action

Return type
Tensor

`proba_distribution`(*mean_actions*, *log_std*, *latent_sde*)

Create the distribution given its parameters (mean, std)

Parameters

- **mean_actions** (*Tensor*)
- **log_std** (*Tensor*)
- **latent_sde** (*Tensor*)
- **self** (*SelfStateDependentNoiseDistribution*)

Returns

Return type
SelfStateDependentNoiseDistribution

`proba_distribution_net`(*latent_dim*, *log_std_init=-2.0*, *latent_sde_dim=None*)

Create the layers and parameter that represent the distribution: one output will be the deterministic action, the other parameter will be the standard deviation of the distribution that control the weights of the noise matrix.

Parameters

- **latent_dim** (*int*) – Dimension of the last layer of the policy (before the action layer)
- **log_std_init** (*float*) – Initial value for the log standard deviation
- **latent_sde_dim** (*int | None*) – Dimension of the last layer of the features extractor for gSDE. By default, it is shared with the policy network.

Returns

Return type
tuple[*Module*, *Parameter*]

`sample`()

Returns a sample from the probability distribution

Returns
the stochastic action

Return type
Tensor

`sample_weights`(*log_std*, *batch_size=1*)

Sample weights for the noise exploration matrix, using a centered Gaussian distribution.

Parameters

- **log_std** (*Tensor*)
- **batch_size** (*int*)

Return type
None

class `stable_baselines3.common.distributions.TanhBijector`(*epsilon=1e-06*)

Bijective transformation of a probability distribution using a squashing function (tanh)

Parameters
`epsilon` (*float*) – small value to avoid NaN due to numerical imprecision.

`static atanh(x)`

Inverse of Tanh

Taken from Pyro: https://github.com/pyro-ppl/pyro 0.5 * torch.log((1 + x) / (1 - x))

Parameters
`x` (*Tensor*)

Return type
Tensor

`static inverse(y)`

Inverse tanh.

Parameters
`y` (*Tensor*)

Returns

Return type
Tensor

`stable_baselines3.common.distributions.kl_divergence(dist_true, dist_pred)`

Wrapper for the PyTorch implementation of the full form KL Divergence

Parameters

- `dist_true` (`Distribution`) – the p distribution
- `dist_pred` (`Distribution`) – the q distribution

Returns
KL(dist_true||dist_pred)

Return type
Tensor

`stable_baselines3.common.distributions.make_proba_distribution(action_space, use_sde=False, dist_kwargs=None)`

Return an instance of Distribution for the correct type of action space

Parameters

- `action_space` (*Space*) – the input action space
- `use_sde` (*bool*) – Force the use of StateDependentNoiseDistribution instead of DiagGaussianDistribution
- `dist_kwargs` (*dict[str, Any] | None*) – Keyword arguments to pass to the probability distribution

Returns
the appropriate Distribution object

Return type
Distribution

`stable_baselines3.common.distributions.sum_independent_dims(tensor)`

Continuous actions are usually considered to be independent, so we can sum components of the `log_prob` or the entropy.

Parameters
: **tensor** (*Tensor*) – shape: (n_batch, n_actions) or (n_batch,)

Returns
: shape: (n_batch,) for (n_batch, n_actions) input, scalar for (n_batch,) input

Return type
: *Tensor*

1.34 Evaluation Helper

`stable_baselines3.common.evaluation.`**`evaluate_policy`**`(`*model*, *env*, *n_eval_episodes=10*, *deterministic=True*, *render=False*, *callback=None*, *reward_threshold=None*, *return_episode_rewards=False*, *warn=True*`)`

Runs the policy for `n_eval_episodes` episodes and outputs the average return per episode (sum of undiscounted rewards). If a vector env is passed in, this divides the episodes to evaluate onto the different elements of the vector env. This static division of work is done to remove bias. See https://github.com/DLR-RM/stable-baselines3/issues/402 for more details and discussion.

> **Note**
>
> If environment has not been wrapped with `Monitor` wrapper, reward and episode lengths are counted as it appears with `env.step` calls. If the environment contains wrappers that modify rewards or episode lengths (e.g. reward scaling, early episode reset), these will affect the evaluation results as well. You can avoid this by wrapping environment with `Monitor` wrapper before anything else.

Parameters

- **model** (*PolicyPredictor*) – The RL agent you want to evaluate. This can be any object that implements a `predict` method, such as an RL algorithm (`BaseAlgorithm`) or policy (`BasePolicy`).
- **env** (*Env | VecEnv*) – The gym environment or `VecEnv` environment.
- **n_eval_episodes** (*int*) – Number of episode to evaluate the agent
- **deterministic** (*bool*) – Whether to use deterministic or stochastic actions
- **render** (*bool*) – Whether to render the environment or not
- **callback** (*Callable[[dict[str, Any], dict[str, Any]], None] | None*) – callback function to perform additional checks, called `n_envs` times after each step. Gets locals() and globals() passed as parameters. See https://github.com/DLR-RM/stable-baselines3/issues/1912 for more details.
- **reward_threshold** (*float | None*) – Minimum expected reward per episode, this will raise an error if the performance is not met
- **return_episode_rewards** (*bool*) – If True, a list of rewards and episode lengths per episode will be returned instead of the mean.
- **warn** (*bool*) – If True (default), warns user about lack of a Monitor wrapper in the evaluation environment.

Returns
: Mean return per episode (sum of rewards), std of reward per episode. Returns (list[float], list[int])

when `return_episode_rewards` is True, first list containing per-episode return and second containing per-episode lengths (in number of steps).

Return type

tuple[float, float] | tuple[list[float], list[int]]

1.35 Gym Environment Checker

`stable_baselines3.common.env_checker.`**`check_env`**(*env*, *warn=True*, *skip_render_check=True*)

Check that an environment follows Gym API. This is particularly useful when using a custom environment. Please take a look at https://gymnasium.farama.org/api/env/ for more information about the API.

It also optionally check that the environment is compatible with Stable-Baselines.

Parameters

- **`env`** (*`Env`*) – The Gym environment that will be checked
- **`warn`** (*`bool`*) – Whether to output additional warnings mainly related to the interaction with Stable Baselines
- **`skip_render_check`** (*`bool`*) – Whether to skip the checks for the render method. True by default (useful for the CI)

Return type

None

1.36 Monitor Wrapper

`class` `stable_baselines3.common.monitor.`**`Monitor`**(*env*, *filename=None*, *allow_early_resets=True*, *reset_keywords=()*, *info_keywords=()*, *override_existing=True*)

A monitor wrapper for Gym environments, it is used to know the episode reward, length, time and other data.

Parameters

- **`env`** (*`Env`*) – The environment
- **`filename`** (*`str | None`*) – the location to save a log file, can be None for no log
- **`allow_early_resets`** (*`bool`*) – allows the reset of the environment before it is done
- **`reset_keywords`** (*`tuple[str, ...]`*) – extra keywords for the reset call, if extra parameters are needed at reset
- **`info_keywords`** (*`tuple[str, ...]`*) – extra information to log, from the information return of env.step()
- **`override_existing`** (*`bool`*) – appends to file if `filename` exists, otherwise override existing files (default)

`close`()

Closes the environment

Return type

None

`get_episode_lengths`()

Returns the number of timesteps of all the episodes

Returns

Return type
list[int]

`get_episode_rewards()`

Returns the rewards of all the episodes

Returns

Return type
list[float]

`get_episode_times()`

Returns the runtime in seconds of all the episodes

Returns

Return type
list[float]

`get_total_steps()`

Returns the total number of timesteps

Returns

Return type
int

`reset`(***kwargs*)

Calls the Gym environment reset. Can only be called if the environment is over, or if allow_early_resets is True

Parameters
`kwargs` – Extra keywords saved for the next episode. only if defined by reset_keywords

Returns
the first observation of the environment

Return type
tuple[*ObsType*, dict[str, *Any*]]

`step`(*action*)

Step the environment with the given action

Parameters
`action` (*ActType*) – the action

Returns
observation, reward, terminated, truncated, information

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

`class stable_baselines3.common.monitor.ResultsWriter`(*filename=''*, *header=None*, *extra_keys=()*, *override_existing=True*)

A result writer that saves the data from the *Monitor* class

Parameters

- **filename** (*str*) – the location to save a log file. When it does not end in the string `"monitor.csv"`, this suffix will be appended to it

- **header** (*dict[str, float | str] | None*) – the header dictionary object of the saved csv
- **extra_keys** (*tuple[str, ...]*) – the extra information to log, typically is composed of `reset_keywords` and `info_keywords`
- **override_existing** (*bool*) – appends to file if `filename` exists, otherwise override existing files (default)

close()

Close the file handler

Return type
None

write_row(*epinfo*)

Write row of monitor data to csv log file.

Parameters
epinfo (*dict[str, float]*) – the information on episodic return, length, and time

Return type
None

stable_baselines3.common.monitor.**get_monitor_files**(*path*)

get all the monitor files in the given path

Parameters
path (*str*) – the logging folder

Returns
the log files

Return type
list[str]

stable_baselines3.common.monitor.**load_results**(*path*)

Load all Monitor logs from a given directory path matching *`monitor.csv`

Parameters
path (*str*) – the directory path containing the log file(s)

Returns
the logged data

Return type
DataFrame

1.37 Logger

To overwrite the default logger, you can pass one to the algorithm. Available formats are `["stdout", "csv", "log", "tensorboard", "json"]`.

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

```
from stable_baselines3 import A2C
from stable_baselines3.common.logger import configure

tmp_path = "/tmp/sb3_log/"
# set up logger
new_logger = configure(tmp_path, ["stdout", "csv", "tensorboard"])

model = A2C("MlpPolicy", "CartPole-v1", verbose=1)
# Set new logger
model.set_logger(new_logger)
model.learn(10000)
```

1.37.1 Explanation of logger output

You can find below short explanations of the values logged in Stable-Baselines3 (SB3). Depending on the algorithm used and of the wrappers/callbacks applied, SB3 only logs a subset of those keys during training.

Below you can find an example of the logger output when training a PPO agent:

```
-----------------------------------------
| eval/                   |             |
|    mean_ep_length       | 200         |
|    mean_reward          | -157        |
| rollout/                |             |
|    ep_len_mean          | 200         |
|    ep_rew_mean          | -227        |
| time/                   |             |
|    fps                  | 972         |
|    iterations           | 19          |
|    time_elapsed         | 80          |
|    total_timesteps      | 77824       |
| train/                  |             |
|    approx_kl            | 0.037781604 |
|    clip_fraction        | 0.243       |
|    clip_range           | 0.2         |
|    entropy_loss         | -1.06       |
|    explained_variance   | 0.999       |
|    learning_rate        | 0.001       |
|    loss                 | 0.245       |
|    n_updates            | 180         |
|    policy_gradient_loss | -0.00398    |
|    std                  | 0.205       |
|    value_loss           | 0.226       |
-----------------------------------------
```

eval/

All `eval/` values are computed by the `EvalCallback`.

- `mean_ep_length`: Mean episode length
- `mean_reward`: Mean episodic reward (during evaluation)
- `success_rate`: Mean success rate during evaluation (1.0 means 100% success), the environment info dict must contain an `is_success` key to compute that value

rollout/

- ep_len_mean: Mean episode length (averaged over stats_window_size episodes, 100 by default)
- ep_rew_mean: Mean episodic training reward (averaged over stats_window_size episodes, 100 by default), a Monitor wrapper is required to compute that value (automatically added by *make_vec_env*).
- exploration_rate: Current value of the exploration rate when using DQN, it corresponds to the fraction of actions taken randomly (epsilon of the "epsilon-greedy" exploration)
- success_rate: Mean success rate during training (averaged over stats_window_size episodes, 100 by default), you must pass an extra argument to the Monitor wrapper to log that value (info_keywords=("is_success",)) and provide info["is_success"]=True/False on the final step of the episode

time/

- episodes: Total number of episodes
- fps: Number of frames per seconds (includes time taken by gradient update)
- iterations: Number of iterations (data collection + policy update for A2C/PPO)
- time_elapsed: Time in seconds since the beginning of training
- total_timesteps: Total number of timesteps (steps in the environments)

train/

- actor_loss: Current value for the actor loss for off-policy algorithms
- approx_kl: approximate mean KL divergence between old and new policy (for PPO), it is an estimation of how much changes happened in the update
- clip_fraction: mean fraction of surrogate loss that was clipped (above clip_range threshold) for PPO.
- clip_range: Current value of the clipping factor for the surrogate loss of PPO
- critic_loss: Current value for the critic function loss for off-policy algorithms, usually error between value function output and TD(0), temporal difference estimate
- ent_coef: Current value of the entropy coefficient (when using SAC)
- ent_coef_loss: Current value of the entropy coefficient loss (when using SAC)
- entropy_loss: Mean value of the entropy loss (negative of the average policy entropy)
- explained_variance: Fraction of the return variance explained by the value function, see https://scikit-learn.org/stable/modules/model_evaluation.html#explained-variance-score (ev=0 => might as well have predicted zero, ev=1 => perfect prediction, ev<0 => worse than just predicting zero)
- learning_rate: Current learning rate value
- loss: Current total loss value
- n_updates: Number of gradient updates applied so far
- policy_gradient_loss: Current value of the policy gradient loss (its value does not have much meaning)
- value_loss: Current value for the value function loss for on-policy algorithms, usually error between value function output and Monte-Carlo estimate (or TD(lambda) estimate)
- std: Current standard deviation of the noise when using generalized State-Dependent Exploration (gSDE)

class stable_baselines3.common.logger.**CSVOutputFormat**(*filename*)

Log to a file, in a CSV format

Parameters

filename (*str*) – the file to write the log to

close()

closes the file

Return type

None

write(*key_values*, *key_excluded*, *step=0*)

Write a dictionary to file

Parameters

- **key_values** (*dict[str, Any]*)
- **key_excluded** (*dict[str, tuple[str, ...]]*)
- **step** (*int*)

Return type

None

class stable_baselines3.common.logger.**Figure**(*figure*, *close*)

Figure data class storing a matplotlib figure and whether to close the figure after logging it

Parameters

- **figure** (*Figure*) – figure to log
- **close** (*bool*) – if true, close the figure after logging it

exception stable_baselines3.common.logger.**FormatUnsupportedError**(*unsupported_formats*, *value_description*)

Custom error to display informative message when a value is not supported by some formats.

Parameters

- **unsupported_formats** (*Sequence[str]*) – A sequence of unsupported formats, for instance ["stdout"].
- **value_description** (*str*) – Description of the value that cannot be logged by this format.

class stable_baselines3.common.logger.**HParam**(*hparam_dict*, *metric_dict*)

Hyperparameter data class storing hyperparameters and metrics in dictionaries

Parameters

- **hparam_dict** (*Mapping[str, bool | str | float | None]*) – key-value pairs of hyperparameters to log
- **metric_dict** (*Mapping[str, float]*) – key-value pairs of metrics to log A non-empty metrics dict is required to display hyperparameters in the corresponding Tensorboard section.

class stable_baselines3.common.logger.**HumanOutputFormat**(*filename_or_file*, *max_length=36*)

A human-readable output format producing ASCII tables of key-value pairs.

Set attribute `max_length` to change the maximum length of keys and values to write to output (or specify it when calling `__init__`).

Parameters

- **filename_or_file** (*str | TextIO*) – the file to write the log to
- **max_length** (*int*) – the maximum length of keys and values to write to output. Outputs longer than this will be truncated. An error will be raised if multiple keys are truncated to the same value. The maximum output width will be `2*max_length + 7`. The default of 36 produces output no longer than 79 characters wide.

close()

closes the file

Return type

None

write(*key_values*, *key_excluded*, *step=0*)

Write a dictionary to file

Parameters

- **key_values** (*dict[str, Any]*)
- **key_excluded** (*dict[str, tuple[str, ...]]*)
- **step** (*int*)

Return type

None

write_sequence(*sequence*)

write_sequence an array to file

Parameters

sequence (*list[str]*)

Return type

None

class stable_baselines3.common.logger.**Image**(*image*, *dataformats*)

Image data class storing an image and data format

Parameters

- **image** (*Tensor | ndarray | str*) – image to log
- **dataformats** (*str*) – Image data format specification of the form NCHW, NHWC, CHW, HWC, HW, WH, etc. More info in add_image method doc at https://pytorch.org/docs/stable/tensorboard.html Gym envs normally use 'HWC' (channel last)

class stable_baselines3.common.logger.**JSONOutputFormat**(*filename*)

Log to a file, in the JSON format

Parameters

filename (*str*) – the file to write the log to

close()

closes the file

Return type

None

write(*key_values*, *key_excluded*, *step=0*)

Write a dictionary to file

Parameters

- **key_values** (*dict[str, Any]*)
- **key_excluded** (*dict[str, tuple[str, ...]]*)
- **step** (*int*)

Return type
None

class stable_baselines3.common.logger.**KVWriter**

Key Value writer

close()

Close owned resources

Return type
None

write(*key_values*, *key_excluded*, *step=0*)

Write a dictionary to file

Parameters

- **key_values** (*dict[str, Any]*)
- **key_excluded** (*dict[str, tuple[str, ...]]*)
- **step** (*int*)

Return type
None

class stable_baselines3.common.logger.**Logger**(*folder*, *output_formats*)

The logger class.

Parameters

- **folder** (*str | None*) – the logging location
- **output_formats** (*list[KVWriter]*) – the list of output formats

close()

closes the file

Return type
None

debug(**args*)

Write the sequence of args, with no separators, to the console and output files (if you've configured an output file). Using the DEBUG level.

Parameters
args – log the arguments

Return type
None

dump(*step=0*)

Write all of the diagnostics from the current iteration

Parameters
step (*int*)

Return type
None

`error(*args)`

Write the sequence of args, with no separators, to the console and output files (if you've configured an output file). Using the ERROR level.

Parameters
`args` – log the arguments

Return type
None

`get_dir()`

Get directory that log files are being written to. will be None if there is no output directory (i.e., if you didn't call start)

Returns
the logging directory

Return type
str | None

`info(*args)`

Write the sequence of args, with no separators, to the console and output files (if you've configured an output file). Using the INFO level.

Parameters
`args` – log the arguments

Return type
None

`log(*args, level=20)`

Write the sequence of args, with no separators, to the console and output files (if you've configured an output file).

level: int. (see logger.py docs) If the global logger level is higher than
the level argument here, don't print to stdout.

Parameters

- `args` – log the arguments
- `level` (*int*) – the logging level (can be DEBUG=10, INFO=20, WARN=30, ERROR=40, DISABLED=50)

Return type
None

`record(key, value, exclude=None)`

Log a value of some diagnostic Call this once for each diagnostic quantity, each iteration If called many times, last value will be used.

Parameters

- `key` (*str*) – save to log this key
- `value` (*Any*) – save to log this value
- `exclude` (*str | tuple[str, ...] | None*) – outputs to be excluded

Return type
None

`record_mean`(*key*, *value*, *exclude=None*)

The same as record(), but if called many times, values averaged.

Parameters

- **key** (*str*) – save to log this key
- **value** (*float | None*) – save to log this value
- **exclude** (*str | tuple[str, ...] | None*) – outputs to be excluded

Return type
None

`set_level`(*level*)

Set logging threshold on current logger.

Parameters
level (*int*) – the logging level (can be DEBUG=10, INFO=20, WARN=30, ERROR=40, DISABLED=50)

Return type
None

`static to_tuple`(*string_or_tuple*)

Helper function to convert str to tuple of str.

Parameters
string_or_tuple (*str | tuple[str, ...] | None*)

Return type
tuple[str, …]

`warn`(**args*)

Write the sequence of args, with no separators, to the console and output files (if you've configured an output file). Using the WARN level.

Parameters
args – log the arguments

Return type
None

`class stable_baselines3.common.logger.SeqWriter`

sequence writer

`write_sequence`(*sequence*)

write_sequence an array to file

Parameters
sequence (*list[str]*)

Return type
None

`class stable_baselines3.common.logger.TensorBoardOutputFormat`(*folder*)

Dumps key/value pairs into TensorBoard's numeric format.

Parameters
folder (*str*) – the folder to write the log to

`close()`

closes the file

Return type
None

`write`(*key_values*, *key_excluded*, *step=0*)

Write a dictionary to file

Parameters

- **key_values** (*dict[str, Any]*)
- **key_excluded** (*dict[str, tuple[str, ...]]*)
- **step** (*int*)

Return type
None

`class stable_baselines3.common.logger.Video`(*frames*, *fps*)

Video data class storing the video frames and the frame per seconds

Parameters

- **frames** (*Tensor*) – frames to create the video from
- **fps** (*float*) – frames per second

`stable_baselines3.common.logger.configure`(*folder=None*, *format_strings=None*)

Configure the current logger.

Parameters

- **folder** (*str | None*) – the save location (if None, $SB3_LOGDIR, if still None, tempdir/SB3-[date & time])
- **format_strings** (*list[str] | None*) – the output logging format (if None, $SB3_LOG_FORMAT, if still None, [‘stdout’, ‘log’, ‘csv’])

Returns
The logger object.

Return type
Logger

`stable_baselines3.common.logger.filter_excluded_keys`(*key_values*, *key_excluded*, *_format*)

Filters the keys specified by `key_exclude` for the specified format

Parameters

- **key_values** (*dict[str, Any]*) – log dictionary to be filtered
- **key_excluded** (*dict[str, tuple[str, ...]]*) – keys to be excluded per format
- **_format** (*str*) – format for which this filter is run

Returns
dict without the excluded keys

Return type
dict[str, *Any*]

`stable_baselines3.common.logger.`**`make_output_format`**`(`*_format*, *log_dir*, *log_suffix=''*`)`

return a logger for the requested format

Parameters

- **`_format`** (*`str`*) – the requested format to log to ('stdout', 'log', 'json' or 'csv' or 'tensorboard')
- **`log_dir`** (*`str`*) – the logging directory
- **`log_suffix`** (*`str`*) – the suffix for the log file

Returns

the logger

Return type

KVWriter

`stable_baselines3.common.logger.`**`read_csv`**`(`*filename*`)`

read a csv file using pandas

Parameters

`filename` (*`str`*) – the file path to read

Returns

the data in the csv

Return type

DataFrame

`stable_baselines3.common.logger.`**`read_json`**`(`*filename*`)`

read a json file using pandas

Parameters

`filename` (*`str`*) – the file path to read

Returns

the data in the json

Return type

DataFrame

1.38 Action Noise

`class` `stable_baselines3.common.noise.`**`ActionNoise`**

The action noise base class

`reset``()`

Call end of episode reset for the noise

Return type

None

`class` `stable_baselines3.common.noise.`**`NormalActionNoise`**`(`*mean*, *sigma*, *dtype=<class 'numpy.float32'>*`)`

A Gaussian action noise.

Parameters

- **`mean`** (*`ndarray`*) – Mean value of the noise

- **sigma** (*ndarray*) – Scale of the noise (std here)
- **dtype** (*dtype[Any] | None | type[Any] | _SupportsDType[dtype[Any]] | str | tuple[Any, int] | tuple[Any, SupportsIndex | Sequence[SupportsIndex]] | list[Any] | _DTypeDict | tuple[Any, Any]*) – Type of the output noise

class stable_baselines3.common.noise.**OrnsteinUhlenbeckActionNoise**(*mean*, *sigma*, *theta=0.15*, *dt=0.01*, *initial_noise=None*, *dtype=<class 'numpy.float32'>*)

An Ornstein Uhlenbeck action noise, this is designed to approximate Brownian motion with friction.

Based on http://math.stackexchange.com/questions/1287634/implementing-ornstein-uhlenbeck-in-matlab

Parameters

- **mean** (*ndarray*) – Mean of the noise
- **sigma** (*ndarray*) – Scale of the noise
- **theta** (*float*) – Rate of mean reversion
- **dt** (*float*) – Timestep for the noise
- **initial_noise** (*ndarray | None*) – Initial value for the noise output, (if None: 0)
- **dtype** (*dtype[Any] | None | type[Any] | _SupportsDType[dtype[Any]] | str | tuple[Any, int] | tuple[Any, SupportsIndex | Sequence[SupportsIndex]] | list[Any] | _DTypeDict | tuple[Any, Any]*) – Type of the output noise

reset()

reset the Ornstein Uhlenbeck noise, to the initial position

Return type
None

class stable_baselines3.common.noise.**VectorizedActionNoise**(*base_noise*, *n_envs*)

A Vectorized action noise for parallel environments.

Parameters

- **base_noise** (ActionNoise) – Noise generator to use
- **n_envs** (*int*) – Number of parallel environments

reset(*indices=None*)

Reset all the noise processes, or those listed in indices.

Parameters
indices (*Iterable[int] | None*) – The indices to reset. Default: None. If the parameter is None, then all processes are reset to their initial position.

Return type
None

1.39 Utils

`class stable_baselines3.common.utils.ConstantSchedule(val)`

Constant schedule that always returns the same value. Useful for fixed learning rates or clip ranges.

Parameters
`val` (*float*) – constant value

`class stable_baselines3.common.utils.FloatSchedule(value_schedule)`

Wrapper that ensures the output of a Schedule is cast to float. Can wrap either a constant value or an existing callable Schedule.

Parameters
`value_schedule` (*Callable[[float], float] | float*) – Constant value or callable schedule (e.g. LinearSchedule, ConstantSchedule)

`class stable_baselines3.common.utils.LinearSchedule(start, end, end_fraction)`

LinearSchedule interpolates linearly between start and end between `progress_remaining` = 1 and `progress_remaining` = `end_fraction`. This is used in DQN for linearly annealing the exploration fraction (epsilon for the epsilon-greedy strategy).

Parameters

- **start** (*float*) – value to start with if `progress_remaining` = 1
- **end** (*float*) – value to end with if `progress_remaining` = 0
- **end_fraction** (*float*) – fraction of `progress_remaining` where end is reached e.g 0.1 then end is reached after 10% of the complete training process.

`stable_baselines3.common.utils.check_for_correct_spaces(env, observation_space, action_space)`

Checks that the environment has same spaces as provided ones. Used by BaseAlgorithm to check if spaces match after loading the model with given env. Checked parameters: - observation_space - action_space

Parameters

- **env** (*Env | VecEnv*) – Environment to check for valid spaces
- **observation_space** (*Space*) – Observation space to check against
- **action_space** (*Space*) – Action space to check against

Return type
None

`stable_baselines3.common.utils.check_shape_equal(space1, space2)`

If the spaces are Box, check that they have the same shape.

If the spaces are Dict, it recursively checks the subspaces.

Parameters

- **space1** (*Space*) – Space
- **space2** (*Space*) – Other space

Return type
None

`stable_baselines3.common.utils.configure_logger(verbose=0, tensorboard_log=None, tb_log_name='', reset_num_timesteps=True)`

Configure the logger's outputs.

Parameters

- **verbose** (*int*) – Verbosity level: 0 for no output, 1 for the standard output to be part of the logger outputs
- **tensorboard_log** (*str | None*) – the log location for tensorboard (if None, no logging)
- **tb_log_name** (*str*) – tensorboard log
- **reset_num_timesteps** (*bool*) – Whether the `num_timesteps` attribute is reset or not. It allows to continue a previous learning curve (`reset_num_timesteps=False`) or start from t=0 (`reset_num_timesteps=True`, the default).

Returns
The logger object

Return type
Logger

`stable_baselines3.common.utils.`**`constant_fn`**`(`*val*`)`

Create a function that returns a constant It is useful for learning rate schedule (to avoid code duplication)

Parameters
val (*float*) – constant value

Returns
Constant schedule function.

Return type
Callable[[float], float]

`stable_baselines3.common.utils.`**`explained_variance`**`(`*y_pred*, *y_true*`)`

Computes fraction of variance that ypred explains about y. Returns 1 - Var[y-ypred] / Var[y]

interpretation:
ev=0 => might as well have predicted zero ev=1 => perfect prediction ev<0 => worse than just predicting zero

Parameters

- **y_pred** (*ndarray*) – the prediction
- **y_true** (*ndarray*) – the expected value

Returns
explained variance of ypred and y

Return type
float

`stable_baselines3.common.utils.`**`get_device`**`(`*device='auto'*`)`

Retrieve PyTorch device. It checks that the requested device is available first. For now, it supports only cpu and cuda. By default, it tries to use the gpu.

Parameters
device (*device | str*) – One for 'auto', 'cuda', 'cpu'

Returns
Supported Pytorch device

Return type
device

stable_baselines3.common.utils.**get_latest_run_id**(*log_path=''*, *log_name=''*)

Returns the latest run number for the given log name and log path, by finding the greatest number in the directories.

Parameters

- **log_path** (*str*) – Path to the log folder containing several runs.
- **log_name** (*str*) – Name of the experiment. Each run is stored in a folder named log_name_1, log_name_2, …

Returns
latest run number

Return type
int

stable_baselines3.common.utils.**get_linear_fn**(*start*, *end*, *end_fraction*)

Create a function that interpolates linearly between start and end between progress_remaining = 1 and progress_remaining = end_fraction. This is used in DQN for linearly annealing the exploration fraction (epsilon for the epsilon-greedy strategy).

Params start
value to start with if progress_remaining = 1

Params end
value to end with if progress_remaining = 0

Params end_fraction
fraction of progress_remaining where end is reached e.g 0.1 then end is reached after 10% of the complete training process.

Returns
Linear schedule function.

Parameters

- **start** (*float*)
- **end** (*float*)
- **end_fraction** (*float*)

Return type
Callable[[float], float]

stable_baselines3.common.utils.**get_parameters_by_name**(*model*, *included_names*)

Extract parameters from the state dict of model if the name contains one of the strings in included_names.

Parameters

- **model** (*Module*) – the model where the parameters come from.
- **included_names** (*Iterable[str]*) – substrings of names to include.

Returns
List of parameters values (Pytorch tensors) that matches the queried names.

Return type
list[*Tensor*]

stable_baselines3.common.utils.**get_schedule_fn**(*value_schedule*)

Transform (if needed) learning rate and clip range (for PPO) to callable.

Parameters
`value_schedule` (*Callable[[float], float] | float*) – Constant value of schedule function

Returns
Schedule function (can return constant value)

Return type
Callable[[float], float]

`stable_baselines3.common.utils.`**`get_system_info`**(*print_info=True*)

Retrieve system and python env info for the current system.

Parameters
`print_info` (*bool*) – Whether to print or not those infos

Returns
Dictionary summing up the version for each relevant package and a formatted string.

Return type
tuple[dict[str, str], str]

`stable_baselines3.common.utils.`**`is_vectorized_box_observation`**(*observation*, *observation_space*)

For box observation type, detects and validates the shape, then returns whether or not the observation is vectorized.

Parameters

- `observation` (*ndarray*) – the input observation to validate
- `observation_space` (*Box*) – the observation space

Returns
whether the given observation is vectorized or not

Return type
bool

`stable_baselines3.common.utils.`**`is_vectorized_dict_observation`**(*observation*, *observation_space*)

For dict observation type, detects and validates the shape, then returns whether or not the observation is vectorized.

Parameters

- `observation` (*ndarray*) – the input observation to validate
- `observation_space` (*Dict*) – the observation space

Returns
whether the given observation is vectorized or not

Return type
bool

`stable_baselines3.common.utils.`**`is_vectorized_discrete_observation`**(*observation*, *observation_space*)

For discrete observation type, detects and validates the shape, then returns whether or not the observation is vectorized.

Parameters

- `observation` (*int | ndarray*) – the input observation to validate
- `observation_space` (*Discrete*) – the observation space

Returns
whether the given observation is vectorized or not

Return type
bool

stable_baselines3.common.utils.**is_vectorized_multibinary_observation**(*observation*, *observation_space*)

For multibinary observation type, detects and validates the shape, then returns whether or not the observation is vectorized.

Parameters

- **observation** (*ndarray*) – the input observation to validate
- **observation_space** (*MultiBinary*) – the observation space

Returns
whether the given observation is vectorized or not

Return type
bool

stable_baselines3.common.utils.**is_vectorized_multidiscrete_observation**(*observation*, *observation_space*)

For multidiscrete observation type, detects and validates the shape, then returns whether or not the observation is vectorized.

Parameters

- **observation** (*ndarray*) – the input observation to validate
- **observation_space** (*MultiDiscrete*) – the observation space

Returns
whether the given observation is vectorized or not

Return type
bool

stable_baselines3.common.utils.**is_vectorized_observation**(*observation*, *observation_space*)

For every observation type, detects and validates the shape, then returns whether or not the observation is vectorized.

Parameters

- **observation** (*int | ndarray*) – the input observation to validate
- **observation_space** (*Space*) – the observation space

Returns
whether the given observation is vectorized or not

Return type
bool

stable_baselines3.common.utils.**obs_as_tensor**(*obs*, *device*)

Moves the observation to the given device.

Parameters

- **obs** (*ndarray | dict[str, ndarray]*)
- **device** (*device*) – PyTorch device

Returns
PyTorch tensor of the observation on a desired device.

Return type
Tensor | dict[str, *Tensor*]

stable_baselines3.common.utils.**polyak_update**(*params*, *target_params*, *tau*)

Perform a Polyak average update on `target_params` using `params`: target parameters are slowly updated towards the main parameters. `tau`, the soft update coefficient controls the interpolation: `tau=1` corresponds to copying the parameters to the target ones whereas nothing happens when `tau=0`. The Polyak update is done in place, with `no_grad`, and therefore does not create intermediate tensors, or a computation graph, reducing memory cost and improving performance. We scale the target params by `1-tau` (in-place), add the new weights, scaled by `tau` and store the result of the sum in the target params (in place). See https://github.com/DLR-RM/stable-baselines3/issues/93

Parameters

- **params** (*Iterable[Tensor]*) – parameters to use to update the target params
- **target_params** (*Iterable[Tensor]*) – parameters to update
- **tau** (*float*) – the soft update coefficient ("Polyak update", between 0 and 1)

Return type
None

stable_baselines3.common.utils.**safe_mean**(*arr*)

Compute the mean of an array if there is at least one element. For empty array, return NaN. It is used for logging only.

Parameters
arr (*ndarray | list | deque*) – Numpy array or list of values

Returns

Return type
float

stable_baselines3.common.utils.**set_random_seed**(*seed*, *using_cuda=False*)

Seed the different random generators.

Parameters

- **seed** (*int*)
- **using_cuda** (*bool*)

Return type
None

stable_baselines3.common.utils.**should_collect_more_steps**(*train_freq*, *num_collected_steps*, *num_collected_episodes*)

Helper used in `collect_rollouts()` of off-policy algorithms to determine the termination condition.

Parameters

- **train_freq** (*TrainFreq*) – How much experience should be collected before updating the policy.
- **num_collected_steps** (*int*) – The number of already collected steps.
- **num_collected_episodes** (*int*) – The number of already collected episodes.

Returns
Whether to continue or not collecting experience by doing rollouts of the current policy.

Return type
bool

`stable_baselines3.common.utils.`**`update_learning_rate`**`(`*optimizer*, *learning_rate*`)`

Update the learning rate for a given optimizer. Useful when doing linear schedule.

Parameters

- **`optimizer`** (*`Optimizer`*) – Pytorch optimizer
- **`learning_rate`** (*`float`*) – New learning rate value

Return type
None

`stable_baselines3.common.utils.`**`zip_strict`**`(`**iterables*`)`

`zip()` function but enforces that iterables are of equal length. Raises `ValueError` if iterables not of equal length. Code inspired by Stackoverflow answer for question #32954486.

Parameters
`*iterables` (*`Iterable`*) – iterables to `zip()`

Return type
Iterable

1.40 Changelog

1.40.1 Release 2.6.1a1 (WIP)

Breaking Changes:

New Features:

Bug Fixes:

- Fixed docker GPU image (PyTorch GPU was not installed)
- Fixed segmentation faults caused by non-portable schedules during model loading (@akanto)

SB3-Contrib

RL Zoo

SBX (SB3 + Jax)

Deprecations:

- `get_schedule_fn()`, `get_linear_fn()`, `constant_fn()` are deprecated, please use `FloatSchedule()`, `LinearSchedule()`, `ConstantSchedule()` instead

Others:

Documentation:

- Clarify `evaluate_policy` documentation
- Added doc about training exceeding the *total_timesteps* parameter

1.40.2 Release 2.6.0 (2025-03-24)

New ``LogEveryNTimesteps`` callback and ``has_attr`` method, refactored hyperparameter optimization

Breaking Changes:

New Features:

- Added `has_attr` method for `VecEnv` to check if an attribute exists
- Added `LogEveryNTimesteps` callback to dump logs every N timesteps (note: you need to pass `log_interval=None` to avoid any interference)
- Added Gymnasium v1.1 support

Bug Fixes:

- *SubProcVecEnv* will now exit gracefully (without big traceback) when using *KeyboardInterrupt*

SB3-Contrib

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

RL Zoo

- Refactored hyperparameter optimization. The Optuna Journal storage backend is now supported (recommended default) and you can easily load tuned hyperparameter via the new `--trial-id` argument of `train.py`.
- Save the exact command line used to launch a training
- Added support for special vectorized env (e.g. Brax, IsaacSim) by allowing to override the `VecEnv` class use to instantiate the env in the `ExperimentManager`
- Allow to disable auto-logging by passing `--log-interval -2` (useful when logging things manually)
- Added Gymnasium v1.1 support
- Fixed use of old HF api in `get_hf_trained_models()`

SBX (SB3 + Jax)

- Updated PPO to support `net_arch`, and additional fixes
- Fixed entropy coeff wrongly logged for SAC and derivatives.
- Fixed PPO `predict()` for env that were not normalized (action spaces with limits != [-1, 1])
- PPO now logs the standard deviation

Deprecations:

- `algo._dump_logs()` is deprecated in favor of `algo.dump_logs()` and will be removed in SB3 v2.7.0

Others:

- Updated black from v24 to v25
- Improved error messages when checking Box space equality (loading `VecNormalize`)
- Updated test to reflect how `set_wrapper_attr` should be used now

Documentation:

- Clarify the use of Gym wrappers with `make_vec_env` in the section on Vectorized Environments (@pstahlhofen)
- Updated callback doc for `EveryNTimesteps`
- Added doc on how to set env attributes via `VecEnv` calls
- Added ONNX export example for `MultiInputPolicy` (@darkopetrovic)

1.40.3 Release 2.5.0 (2025-01-27)

New algorithm: SimBa in SBX, NumPy 2.0 support

Breaking Changes:

- Increased minimum required version of PyTorch to 2.3.0
- Removed support for Python 3.8

New Features:

- Added support for NumPy v2.0: `VecNormalize` now cast normalized rewards to float32, updated bit flipping env to avoid overflow issues too
- Added official support for Python 3.12

SBX (SB3 + Jax)

- Added SimBa Policy: Simplicity Bias for Scaling Up Parameters in DRL
- Added support for parameter resets

Others:

- Updated Dockerfile

Documentation:

- Added Decisions and Dragons to resources. (@jmacglashan)
- Updated PyBullet example, now compatible with Gymnasium
- Added link to policies for `policy_kwargs` parameter (@kplers)
- Add FootstepNet Envs to the project page (@cgaspard3333)
- Added FRASA to the project page (@MarcDcls)
- Fixed atari example (@chrisgao99)
- Add a note about `Discrete` action spaces with `start!=0`
- Update doc for massively parallel simulators (Isaac Lab, Brax, …)
- Add dm_control example

1.40.4 Release 2.4.1 (2024-12-20)

Bug Fixes:

- Fixed a bug introduced in v2.4.0 where the `VecVideoRecorder` would override videos

1.40.5 Release 2.4.0 (2024-11-18)

New algorithm: CrossQ in SB3 Contrib, Gymnasium v1.0 support

> **Note**
>
> DQN (and QR-DQN) models saved with SB3 < 2.4.0 will show a warning about truncation of optimizer state when loaded with SB3 >= 2.4.0. To suppress the warning, simply save the model again. You can find more info in PR #1963

> **Warning**
>
> Stable-Baselines3 (SB3) v2.4.0 will be the last one supporting Python 3.8 (end of life in October 2024) and PyTorch < 2.3. We highly recommended you to upgrade to Python >= 3.9 and PyTorch >= 2.3 (compatible with NumPy v2).

Breaking Changes:

- Increased minimum required version of Gymnasium to 0.29.1

New Features:

- Added support for `pre_linear_modules` and `post_linear_modules` in `create_mlp` (useful for adding normalization layers, like in DroQ or CrossQ)
- Enabled np.ndarray logging for TensorBoardOutputFormat as histogram (see GH#1634) (@iwishwasaneagle)
- Updated env checker to warn users when using multi-dim array to define *MultiDiscrete* spaces
- Added support for Gymnasium v1.0

Bug Fixes:

- Fixed memory leak when loading learner from storage, `set_parameters()` does not try to load the object data anymore and only loads the PyTorch parameters (@peteole)
- Cast type in compute gae method to avoid error when using torch compile (@amjames)
- `CallbackList` now sets the `.parent` attribute of child callbacks to its own `.parent`. (will-maclean)
- Fixed error when loading a model that has `net_arch` manually set to `None` (@jak3122)
- Set requirement numpy<2.0 until PyTorch is compatible (https://github.com/pytorch/pytorch/issues/107302)
- Updated DQN optimizer input to only include q_network parameters, removing the target_q_network ones (@corentinlger)
- Fixed `test_buffers.py::test_device` which was not actually checking the device of tensors (@rhaps0dy)

SB3-Contrib

- Added `CrossQ` algorithm, from “Batch Normalization in Deep Reinforcement Learning” paper (@danielpalen)
- Added `BatchRenorm` PyTorch layer used in `CrossQ` (@danielpalen)
- Updated QR-DQN optimizer input to only include quantile_net parameters (@corentinlger)
- Fixed loading QRDQN changes *target_update_interval* (@jak3122)

RL Zoo

- Updated defaults hyperparameters for TQC/SAC for Swimmer-v4 (decrease gamma for more consistent results)

SBX (SB3 + Jax)

- Added CNN support for DQN
- Bug fix for SAC and related algorithms, optimize log of ent coeff to be consistent with SB3

Deprecations:

Others:

- Fixed various typos (@cschindlbeck)
- Remove unnecessary SDE noise resampling in PPO update (@brn-dev)
- Updated PyTorch version on CI to 2.3.1
- Added a warning to recommend using CPU with on policy algorithms (A2C/PPO) and `MlpPolicy`
- Switched to uv to download packages faster on GitHub CI
- Updated dependencies for read the doc
- Removed unnecessary `copy_obs_dict` method for `SubprocVecEnv`, remove the use of ordered dict and rename `flatten_obs` to `stack_obs`

Documentation:

- Updated PPO doc to recommend using CPU with `MlpPolicy`
- Clarified documentation about planned features and citing software
- Added a note about the fact we are optimizing log of ent coeff for SAC

1.40.6 Release 2.3.2 (2024-04-27)

Bug Fixes:

- Reverted `torch.load()` to be called `weights_only=False` as it caused loading issue with old version of PyTorch.

Documentation:

- Added ER-MRL to the project page (@corentinlger)
- Updated Tensorboard Logging Videos documentation (@NickLucche)

1.40.7 Release 2.3.1 (2024-04-22)

Bug Fixes:

- Cast return value of learning rate schedule to float, to avoid issue when loading model because of `weights_only=True` (@markscsmith)

Documentation:

- Updated SBX documentation (CrossQ and deprecated DroQ)
- Updated RL Tips and Tricks section

1.40.8 Release 2.3.0 (2024-03-31)

New defaults hyperparameters for DDPG, TD3 and DQN

⚠ **Warning**

Because of `weights_only=True`, this release breaks loading of policies when using PyTorch 1.13. Please upgrade to PyTorch >= 2.0 or upgrade SB3 version (we reverted the change in SB3 2.3.2)

Breaking Changes:

- The defaults hyperparameters of `TD3` and `DDPG` have been changed to be more consistent with `SAC`

```
# SB3 < 2.3.0 default hyperparameters
# model = TD3("MlpPolicy", env, train_freq=(1, "episode"), gradient_steps=-1, batch_
↪size=100)
# SB3 >= 2.3.0:
model = TD3("MlpPolicy", env, train_freq=1, gradient_steps=1, batch_size=256)
```

ℹ **Note**

Two inconsistencies remain: the default network architecture for `TD3/DDPG` is `[400, 300]` instead of `[256, 256]` for SAC (for backward compatibility reasons, see report on the influence of the network size) and the default learning rate is 1e-3 instead of 3e-4 for SAC (for performance reasons, see W&B report on the influence of the lr)

- The default `learning_starts` parameter of `DQN` 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 = DQN("MlpPolicy", env, learning_starts=50_000)
# SB3 >= 2.3.0:
model = DQN("MlpPolicy", env, learning_starts=100)
```

- For safety, `torch.load()` is now called with `weights_only=True` when loading torch tensors, policy `load()` still uses `weights_only=False` as gymnasium imports are required for it to work
- When using `huggingface_sb3`, you will now need to set `TRUST_REMOTE_CODE=True` when downloading models from the hub, as `pickle.load` is not safe.

New Features:

- Log success rate `rollout/success_rate` when available for on policy algorithms (@corentinlger)

Bug Fixes:

- Fixed `monitor_wrapper` argument that was not passed to the parent class, and dones argument that wasn't passed to `_update_into_buffer` (@corentinlger)

SB3-Contrib

- Added `rollout_buffer_class` and `rollout_buffer_kwargs` arguments to MaskablePPO
- 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
- Add some additional notes about `MaskablePPO` (evaluation and multi-process) (@icheered)

RL Zoo

- Updated defaults hyperparameters for TD3/DDPG to be more consistent with SAC
- Upgraded MuJoCo envs hyperparameters to v4 (pre-trained agents need to be updated)
- Added test dependencies to *setup.py* (@power-edge)
- Simplify dependencies of *requirements.txt* (remove duplicates from *setup.py*)

SBX (SB3 + Jax)

- Added support for `MultiDiscrete` and `MultiBinary` action spaces to PPO
- Added support for large values for gradient_steps to SAC, TD3, and TQC
- Fix `train()` signature and update type hints
- Fix replay buffer device at load time
- Added flatten layer
- Added `CrossQ`

Deprecations:

Others:

- Updated black from v23 to v24
- Updated ruff to >= v0.3.1
- Updated env checker for (multi)discrete spaces with non-zero start.

Documentation:

- Added a paragraph on modifying vectorized environment parameters via setters (@fracapuano)
- Updated callback code example
- Updated export to ONNX documentation, it is now much simpler to export SB3 models with newer ONNX Opset!
- Added video link to "Practical Tips for Reliable Reinforcement Learning" video
- Added `render_mode="human"` in the README example (@marekm4)
- Fixed docstring signature for sum_independent_dims (@stagoverflow)
- Updated docstring description for `log_interval` in the base class (@rushitnshah).

1.40.9 Release 2.2.1 (2023-11-17)

Support for options at reset, bug fixes and better error messages

> **Note**
>
> SB3 v2.2.0 was yanked after a breaking change was found in GH#1751. Please use SB3 v2.2.1 and not v2.2.0.

Breaking Changes:

- 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)

New Features:

- Improved error message of the `env_checker` for env wrongly detected as GoalEnv (`compute_reward()` is defined)
- Improved error message when mixing Gym API with VecEnv API (see GH#1694)
- Add support for setting `options` at reset with VecEnv via the `set_options()` method. Same as seeds logic, options are reset at the end of an episode (@ReHoss)
- Added `rollout_buffer_class` and `rollout_buffer_kwargs` arguments to on-policy algorithms (A2C and PPO)

Bug Fixes:

- Prevents using squash_output and not use_sde in ActorCritcPolicy (@PatrickHelm)
- Performs unscaling of actions in collect_rollout in OnPolicyAlgorithm (@PatrickHelm)
- Moves VectorizedActionNoise into `_setup_learn()` in OffPolicyAlgorithm (@PatrickHelm)
- Prevents out of bound error on Windows if no seed is passed (@PatrickHelm)
- Calls `callback.update_locals()` before `callback.on_rollout_end()` in OnPolicyAlgorithm (@PatrickHelm)
- Fixed replay buffer device after loading in OffPolicyAlgorithm (@PatrickHelm)
- Fixed `render_mode` which was not properly loaded when using `VecNormalize.load()`
- Fixed success reward dtype in `SimpleMultiObsEnv` (@NixGD)
- Fixed check_env for Sequence observation space (@corentinlger)
- Prevents instantiating BitFlippingEnv with conflicting observation spaces (@kylesayrs)
- Fixed ResourceWarning when loading and saving models (files were not closed), please note that only path are closed automatically, the behavior stay the same for tempfiles (they need to be closed manually), the behavior is now consistent when loading/saving replay buffer

SB3-Contrib

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

RL Zoo

- Removed *gym* dependency, the package is still required for some pretrained agents.
- Added *–eval-env-kwargs* to *train.py* (@Quentin18)
- Added *ppo_lstm* to hyperparams_opt.py (@technocrat13)
- Upgraded to *pybullet_envs_gymnasium>=0.4.0*
- Removed old hacks (for instance limiting offpolicy algorithms to one env at test time)
- Updated docker image, removed support for X server
- Replaced deprecated *optuna.suggest_uniform(. . .)* by *optuna.suggest_float(. . . , low=. . . , high=. . .)*

SBX (SB3 + Jax)

- Added `DDPG` and `TD3` algorithms

Deprecations:

Others:

- Fixed `stable_baselines3/common/callbacks.py` type hints
- Fixed `stable_baselines3/common/utils.py` type hints
- Fixed `stable_baselines3/common/vec_envs/vec_transpose.py` type hints
- Fixed `stable_baselines3/common/vec_env/vec_video_recorder.py` type hints
- Fixed `stable_baselines3/common/save_util.py` type hints
- Updated docker images to Ubuntu Jammy using micromamba 1.5
- Fixed `stable_baselines3/common/buffers.py` type hints
- Fixed `stable_baselines3/her/her_replay_buffer.py` type hints
- Buffers do no call an additional `.copy()` when storing new transitions
- Fixed `ActorCriticPolicy.extract_features()` signature by adding an optional `features_extractor` argument
- Update dependencies (accept newer Shimmy/Sphinx version and remove `sphinx_autodoc_typehints`)
- Fixed `stable_baselines3/common/off_policy_algorithm.py` type hints
- Fixed `stable_baselines3/common/distributions.py` type hints
- Fixed `stable_baselines3/common/vec_env/vec_normalize.py` type hints
- Fixed `stable_baselines3/common/vec_env/__init__.py` type hints
- Switched to PyTorch 2.1.0 in the CI (fixes type annotations)
- Fixed `stable_baselines3/common/policies.py` type hints
- Switched to `mypy` only for checking types
- Added tests to check consistency when saving/loading files

Documentation:

- Updated RL Tips and Tricks (include recommendation for evaluation, added links to DroQ, ARS and SBX).
- Fixed various typos and grammar mistakes
- Added PokemonRedExperiments to the project page
- Fixed an out-of-date command for installing Atari in examples

1.40.10 Release 2.1.0 (2023-08-17)

Float64 actions , Gymnasium 0.29 support and bug fixes

Breaking Changes:

- Removed Python 3.7 support
- SB3 now requires PyTorch >= 1.13

New Features:

- Added Python 3.11 support
- Added Gymnasium 0.29 support (@pseudo-rnd-thoughts)

SB3-Contrib

- Fixed MaskablePPO ignoring `stats_window_size` argument
- Added Python 3.11 support

RL Zoo

- Upgraded to Huggingface-SB3 >= 2.3
- Added Python 3.11 support

Bug Fixes:

- Relaxed check in logger, that was causing issue on Windows with colorama
- Fixed off-policy algorithms with continuous float64 actions (see #1145) (@tobirohrer)
- Fixed `env_checker.py` warning messages for out of bounds in complex observation spaces (@Gabo-Tor)

Deprecations:

Others:

- Updated GitHub issue templates
- Fix typo in gym patch error message (@lukashass)
- Refactor `test_spaces.py` tests

Documentation:

- Fixed callback example (@BertrandDecoster)
- Fixed policy network example (@kyle-he)
- Added mobile-env as new community project (@stefanbschneider)

- Added [DeepNetSlice](https://github.com/AlexPasqua/DeepNetSlice) to community projects (@AlexPasqua)

1.40.11 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.

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)
- The deprecated `online_sampling` argument of `HerReplayBuffer` was removed
- Removed deprecated `stack_observation_space` method of `StackedObservations`
- Renamed environment output observations in `evaluate_policy` to prevent shadowing the input observations during callbacks (@npit)
- Upgraded wrappers and custom environment to Gymnasium
- Refined the `HumanOutputFormat` file check: now it verifies if the object is an instance of `io.TextIOBase` instead of only checking for the presence of a `write` method.
- Because of new Gym API (0.26+), the random seed passed to `vec_env.seed(seed=seed)` will only be effective after then `env.reset()` call.

New Features:

- Added Gymnasium support (Gym 0.21 and 0.26 are supported via the `shimmy` package)

SB3-Contrib

- Fixed QRDQN update interval for multi envs

RL Zoo

- Gym 0.26+ patches to continue working with pybullet and TimeLimit wrapper
- Renamed *CarRacing-v1* to *CarRacing-v2* in hyperparameters
- Huggingface push to hub now accepts a *–n-timesteps* argument to adjust the length of the video
- Fixed *record_video* steps (before it was stepping in a closed env)
- Dropped Gym 0.21 support

Bug Fixes:

- Fixed `VecExtractDictObs` does not handle terminal observation (@WeberSamuel)
- Set NumPy version to `>=1.20` due to use of `numpy.typing` (@troiganto)
- Fixed loading DQN changes `target_update_interval` (@tobirohrer)
- Fixed env checker to properly reset the env before calling `step()` when checking for `Inf` and `NaN` (@lutogniew)

- Fixed HER `truncate_last_trajectory()` (@lbergmann1)
- Fixed HER desired and achieved goal order in reward computation (@JonathanKuelz)

Deprecations:

Others:

- Fixed `stable_baselines3/a2c/*.py` type hints
- Fixed `stable_baselines3/ppo/*.py` type hints
- Fixed `stable_baselines3/sac/*.py` type hints
- Fixed `stable_baselines3/td3/*.py` type hints
- Fixed `stable_baselines3/common/base_class.py` type hints
- Fixed `stable_baselines3/common/logger.py` type hints
- Fixed `stable_baselines3/common/envs/*.py` type hints
- Fixed `stable_baselines3/common/vec_env/vec_monitor|vec_extract_dict_obs|util.py` type hints
- Fixed `stable_baselines3/common/vec_env/base_vec_env.py` type hints
- Fixed `stable_baselines3/common/vec_env/vec_frame_stack.py` type hints
- Fixed `stable_baselines3/common/vec_env/dummy_vec_env.py` type hints
- Fixed `stable_baselines3/common/vec_env/subproc_vec_env.py` type hints
- Upgraded docker images to use mamba/micromamba and CUDA 11.7
- Updated env checker to reflect what subset of Gymnasium is supported and improve GoalEnv checks
- Improve type annotation of wrappers
- Tests envs are now checked too
- Added render test for `VecEnv` and `VecEnvWrapper`
- Update issue templates and env info saved with the model
- Changed `seed()` method return type from `List` to `Sequence`
- Updated env checker doc and requirements for tuple spaces/goal envs

Documentation:

- Added Deep RL Course link to the Deep RL Resources page
- Added documentation about `VecEnv` API vs Gym API
- Upgraded tutorials to Gymnasium API
- Make it more explicit when using `VecEnv` vs Gym env
- Added UAV_Navigation_DRL_AirSim to the project page (@heleidsn)
- Added `EvalCallback` example (@sidney-tio)
- Update custom env documentation
- Added *pink-noise-rl* to projects page
- Fix custom policy example, `ortho_init` was ignored

- Added SBX page

1.40.12 Release 1.8.0 (2023-04-07)

Multi-env HerReplayBuffer, Open RL Benchmark, Improved env checker

> ⚠ **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.

Breaking Changes:

- Removed shared layers in `mlp_extractor` (@AlexPasqua)
- Refactored `StackedObservations` (it now handles dict obs, `StackedDictObservations` was removed)
- You must now explicitly pass a `features_extractor` parameter when calling `extract_features()`
- Dropped offline sampling for `HerReplayBuffer`
- As `HerReplayBuffer` was refactored to support multiprocessing, previous replay buffer are incompatible with this new version
- `HerReplayBuffer` doesn't require a `max_episode_length` anymore

New Features:

- Added `repeat_action_probability` argument in `AtariWrapper`.
- Only use `NoopResetEnv` and `MaxAndSkipEnv` when needed in `AtariWrapper`
- Added support for dict/tuple observations spaces for `VecCheckNan`, the check is now active in the `env_checker()` (@DavyMorgan)
- Added multiprocessing support for `HerReplayBuffer`
- `HerReplayBuffer` now supports all datatypes supported by `ReplayBuffer`
- Provide more helpful failure messages when validating the `observation_space` of custom gym environments using `check_env` (@FieteO)
- Added `stats_window_size` argument to control smoothing in rollout logging (@jonasreiher)

SB3-Contrib

- Added warning about potential crashes caused by `check_env` in the `MaskablePPO` docs (@AlexPasqua)
- Fixed `sb3_contrib/qrdqn/*.py` type hints
- Removed shared layers in `mlp_extractor` (@AlexPasqua)

RL Zoo

- Open RL Benchmark
- Upgraded to new *HerReplayBuffer* implementation that supports multiple envs
- Removed *TimeFeatureWrapper* for Panda and Fetch envs, as the new replay buffer should handle timeout.

- Tuned hyperparameters for RecurrentPPO on Swimmer
- Documentation is now built using Sphinx and hosted on read the doc
- Removed *use_auth_token* for push to hub util
- Reverted from v3 to v2 for HumanoidStandup, Reacher, InvertedPendulum and InvertedDoublePendulum since they were not part of the mujoco refactoring (see https://github.com/openai/gym/pull/1304)
- Fixed *gym-minigrid* policy (from *MlpPolicy* to *MultiInputPolicy*)
- Replaced deprecated *optuna.suggest_loguniform(…)* by *optuna.suggest_float(…, log=True)*
- Switched to *ruff* and *pyproject.toml*
- Removed *online_sampling* and *max_episode_length* argument when using *HerReplayBuffer*

Bug Fixes:

- Fixed Atari wrapper that missed the reset condition (@luizapozzobon)
- Added the argument `dtype` (default to `float32`) to the noise for consistency with gym action (@sidney-tio)
- Fixed PPO train/n_updates metric not accounting for early stopping (@adamfrly)
- Fixed loading of normalized image-based environments
- Fixed `DictRolloutBuffer.add` with multidimensional action space (@younik)

Deprecations:

Others:

- Fixed `tests/test_tensorboard.py` type hint
- Fixed `tests/test_vec_normalize.py` type hint
- Fixed `stable_baselines3/common/monitor.py` type hint
- Added tests for StackedObservations
- Removed Gitlab CI file
- Moved from `setup.cg` to `pyproject.toml` configuration file
- Switched from `flake8` to `ruff`
- Upgraded AutoROM to latest version
- Fixed `stable_baselines3/dqn/*.py` type hints
- Added `extra_no_roms` option for package installation without Atari Roms

Documentation:

- Renamed `load_parameters` to `set_parameters` (@DavyMorgan)
- Clarified documentation about subproc multiprocessing for A2C (@Bonifatius94)
- Fixed typo in `A2C` docstring (@AlexPasqua)
- Renamed timesteps to episodes for `log_interval` description (@theSquaredError)
- Removed note about gif creation for Atari games (@harveybellini)
- Added information about default network architecture
- Update information about Gymnasium support

1.40.13 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.

Note

A2C and PPO saved with SB3 < 1.7.0 will show a warning about missing keys in the state dict when loaded with SB3 >= 1.7.0. To suppress the warning, simply save the model again. You can find more info in issue #1233

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
- Removed ret attributes in VecNormalize, please use returns instead
- VecNormalize now updates the observation space when normalizing images

New Features:

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

SB3-Contrib

- 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

RL Zoo

- Added support for python file for configuration
- Added monitor_kwargs parameter

Bug Fixes:

- Fixed `ProgressBarCallback` under-reporting (@dominicgkerr)
- Fixed return type of `evaluate_actions` in `ActorCritcPolicy` to reflect that entropy is an optional tensor (@Rocamonde)
- Fixed type annotation of `policy` in `BaseAlgorithm` and `OffPolicyAlgorithm`
- Allowed model trained with Python 3.7 to be loaded with Python 3.8+ without the `custom_objects` workaround
- Raise an error when the same gym environment instance is passed as separate environments when creating a vectorized environment with more than one environment. (@Rocamonde)
- Fix type annotation of `model` in `evaluate_policy`
- Fixed `Self` return type using `TypeVar`
- Fixed the env checker, the key was not passed when checking images from Dict observation space
- Fixed `normalize_images` which was not passed to parent class in some cases
- Fixed `load_from_vector` that was broken with newer PyTorch version when passing PyTorch tensor

Deprecations:

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

Others:

- Used issue forms instead of issue templates
- Updated the PR template to associate each PR with its peer in RL-Zoo3 and SB3-Contrib
- Fixed flake8 config to be compatible with flake8 6+
- Goal-conditioned environments are now characterized by the availability of the `compute_reward` method, rather than by their inheritance to `gym.GoalEnv`
- Replaced `CartPole-v0` by `CartPole-v1` is tests
- Fixed `tests/test_distributions.py` type hints
- Fixed `stable_baselines3/common/type_aliases.py` type hints
- Fixed `stable_baselines3/common/torch_layers.py` type hints
- Fixed `stable_baselines3/common/env_util.py` type hints
- Fixed `stable_baselines3/common/preprocessing.py` type hints
- Fixed `stable_baselines3/common/atari_wrappers.py` type hints
- Fixed `stable_baselines3/common/vec_env/vec_check_nan.py` type hints
- Exposed modules in `__init__.py` with the `__all__` attribute (@ZikangXiong)
- Upgraded GitHub CI/setup-python to v4 and checkout to v3
- Set tensors construction directly on the device (~8% speed boost on GPU)
- Monkey-patched `np.bool = bool` so gym 0.21 is compatible with NumPy 1.24+
- Standardized the use of `from gym import spaces`
- Modified `get_system_info` to avoid issue linked to copy-pasting on GitHub issue

Documentation:

- Updated Hugging Face Integration page (@simoninithomas)
- Changed `env` to `vec_env` when environment is vectorized
- Updated custom policy docs to better explain the `mlp_extractor`'s dimensions (@AlexPasqua)
- Updated custom policy documentation (@athatheo)
- Improved tensorboard callback doc
- Clarify doc when using image-like input
- Added RLeXplore to the project page (@yuanmingqi)

1.40.14 Release 1.6.2 (2022-10-10)

Progress bar in the learn() method, RL Zoo3 is now a package

Breaking Changes:

New Features:

- Added `progress_bar` argument in the `learn()` method, displayed using TQDM and rich packages
- Added progress bar callback
- The RL Zoo can now be installed as a package (`pip install rl_zoo3`)

SB3-Contrib

RL Zoo

- RL Zoo is now a python package and can be installed using `pip install rl_zoo3`

Bug Fixes:

- `self.num_timesteps` was initialized properly only after the first call to `on_step()` for callbacks
- Set importlib-metadata version to `~=4.13` to be compatible with `gym=0.21`

Deprecations:

- Added deprecation warning if parameters `eval_env`, `eval_freq` or `create_eval_env` are used (see #925) (@tobirohrer)

Others:

- Fixed type hint of the `env_id` parameter in `make_vec_env` and `make_atari_env` (@AlexPasqua)

Documentation:

- Extended docstring of the `wrapper_class` parameter in `make_vec_env` (@AlexPasqua)

1.40.15 Release 1.6.1 (2022-09-29)

Bug fix release

Breaking Changes:

- Switched minimum tensorboard version to 2.9.1

New Features:

- Support logging hyperparameters to tensorboard (@timothe-chaumont)
- Added checkpoints for replay buffer and `VecNormalize` statistics (@anand-bala)
- Added option for `Monitor` to append to existing file instead of overriding (@sidney-tio)
- The env checker now raises an error when using dict observation spaces and observation keys don't match observation space keys

SB3-Contrib

- Fixed the issue of wrongly passing policy arguments when using `CnnLstmPolicy` or `MultiInputLstmPolicy` with `RecurrentPPO` (@mlodel)

Bug Fixes:

- Fixed issue where `PPO` gives NaN if rollout buffer provides a batch of size 1 (@hughperkins)
- Fixed the issue that `predict` does not always return action as `np.ndarray` (@qgallouedec)
- Fixed division by zero error when computing FPS when a small number of time has elapsed in operating systems with low-precision timers.
- Added multidimensional action space support (@qgallouedec)
- Fixed missing verbose parameter passing in the `EvalCallback` constructor (@burakdmb)
- Fixed the issue that when updating the target network in DQN, SAC, TD3, the `running_mean` and `running_var` properties of batch norm layers are not updated (@honglu2875)
- Fixed incorrect type annotation of the replay_buffer_class argument in `common.OffPolicyAlgorithm` initializer, where an instance instead of a class was required (@Rocamonde)
- Fixed loading saved model with different number of environments
- Removed `forward()` abstract method declaration from `common.policies.BaseModel` (already defined in `torch.nn.Module`) to fix type errors in subclasses (@Rocamonde)
- Fixed the return type of `.load()` and `.learn()` methods in `BaseAlgorithm` so that they now use `TypeVar` (@Rocamonde)
- Fixed an issue where keys with different tags but the same key raised an error in `common.logger.HumanOutputFormat` (@Rocamonde and @AdamGleave)
- Set importlib-metadata version to *~=4.13*

Deprecations:

Others:

- Fixed `DictReplayBuffer.next_observations` typing (@qgallouedec)
- Added support for `device="auto"` in buffers and made it default (@qgallouedec)
- Updated `ResultsWriter` (used internally by `Monitor` wrapper) to automatically create missing directories when `filename` is a path (@dominicgkerr)

Documentation:

- Added an example of callback that logs hyperparameters to tensorboard. (@timothe-chaumont)
- Fixed typo in docstring "nature" -> "Nature" (@Melanol)
- Added info on split tensorboard logs into (@Melanol)
- Fixed typo in ppo doc (@francescoluciano)
- Fixed typo in install doc(@jlp-ue)
- Clarified and standardized verbosity documentation
- Added link to a GitHub issue in the custom policy documentation (@AlexPasqua)
- Update doc on exporting models (fixes and added torch jit)
- Fixed typos (@Akhilez)
- Standardized the use of `"` for string representation in documentation

1.40.16 Release 1.6.0 (2022-07-11)

Recurrent PPO (PPO LSTM), better defaults for learning from pixels with SAC/TD3

Breaking Changes:

- 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)
- SB3 now requires PyTorch >= 1.11
- Changed the default network architecture when using `CnnPolicy` or `MultiInputPolicy` with SAC or DDPG/TD3, `share_features_extractor` is now set to False by default and the `net_arch=[256, 256]` (instead of `net_arch=[]` that was before)

New Features:

SB3-Contrib

- Added Recurrent PPO (PPO LSTM). See https://github.com/Stable-Baselines-Team/stable-baselines3-contrib/pull/53

Bug Fixes:

- Fixed saving and loading large policies greater than 2GB (@jkterry1, @ycheng517)
- Fixed final goal selection strategy that did not sample the final achieved goal (@qgallouedec)
- Fixed a bug with special characters in the tensorboard log name (@quantitative-technologies)
- Fixed a bug in `DummyVecEnv`'s and `SubprocVecEnv`'s seeding function. None value was unchecked (@ScheiklP)
- Fixed a bug where `EvalCallback` would crash when trying to synchronize `VecNormalize` stats when observation normalization was disabled
- Added a check for unbounded actions
- Fixed issues due to newer version of protobuf (tensorboard) and sphinx
- Fix exception causes all over the codebase (@cool-RR)

- Prohibit simultaneous use of optimize_memory_usage and handle_timeout_termination due to a bug (@MWeltevrede)
- Fixed a bug in `kl_divergence` check that would fail when using numpy arrays with MultiCategorical distribution

Deprecations:

Others:

- Upgraded to Python 3.7+ syntax using `pyupgrade`
- Removed redundant double-check for nested observations from `BaseAlgorithm._wrap_env` (@TibiGG)

Documentation:

- Added link to gym doc and gym env checker
- Fix typo in PPO doc (@bcollazo)
- Added link to PPO ICLR blog post
- Added remark about breaking Markov assumption and timeout handling
- Added doc about MLFlow integration via custom logger (@git-thor)
- Updated Huggingface integration doc
- Added copy button for code snippets
- Added doc about EnvPool and Isaac Gym support

1.40.17 Release 1.5.0 (2022-03-25)

Bug fixes, early stopping callback

Breaking Changes:

- Switched minimum Gym version to 0.21.0

New Features:

- Added `StopTrainingOnNoModelImprovement` to callback collection (@caburu)
- Makes the length of keys and values in `HumanOutputFormat` configurable, depending on desired maximum width of output.
- Allow PPO to turn of advantage normalization (see PR #763) @vwxyzjn

SB3-Contrib

- coming soon: Cross Entropy Method, see https://github.com/Stable-Baselines-Team/stable-baselines3-contrib/pull/62

Bug Fixes:

- Fixed a bug in `VecMonitor`. The monitor did not consider the `info_keywords` during stepping (@ScheiklP)
- Fixed a bug in `HumanOutputFormat`. Distinct keys truncated to the same prefix would overwrite each others value, resulting in only one being output. This now raises an error (this should only affect a small fraction of use cases with very long keys.)

- Routing all the `nn.Module` calls through implicit rather than explicit forward as per pytorch guidelines (@manuel-delverme)
- Fixed a bug in `VecNormalize` where error occurs when `norm_obs` is set to False for environment with dictionary observation (@buoyancy99)
- Set default `env` argument to `None` in `HerReplayBuffer.sample` (@qgallouedec)
- Fix `batch_size` typing in `DQN` (@qgallouedec)
- Fixed sample normalization in `DictReplayBuffer` (@qgallouedec)

Deprecations:

Others:

- Fixed pytest warnings
- Removed parameter `remove_time_limit_termination` in off policy algorithms since it was dead code (@Gregwar)

Documentation:

- Added doc on Hugging Face integration (@simoninithomas)
- Added furuta pendulum project to project list (@armandpl)
- Fix indentation 2 spaces to 4 spaces in custom env documentation example (@Gautam-J)
- Update MlpExtractor docstring (@gianlucadecola)
- Added explanation of the logger output
- Update `Directly Accessing The Summary Writer` in tensorboard integration (@xy9485)

1.40.18 Release 1.4.0 (2022-01-18)

TRPO, ARS and multi env training for off-policy algorithms

Breaking Changes:

- Dropped python 3.6 support (as announced in previous release)
- Renamed `mask` argument of the `predict()` method to `episode_start` (used with RNN policies only)
- local variables `action`, `done` and `reward` were renamed to their plural form for offpolicy algorithms (`actions`, `dones`, `rewards`), this may affect custom callbacks.
- Removed `episode_reward` field from `RolloutReturn()` type

> ⚠ **Warning**
>
> An update to the `HER` algorithm is planned to support multi-env training and remove the max episode length constrain. (see PR #704) This will be a backward incompatible change (model trained with previous version of `HER` won't work with the new version).

New Features:

- Added `norm_obs_keys` param for `VecNormalize` wrapper to configure which observation keys to normalize (@kachayev)
- Added experimental support to train off-policy algorithms with multiple envs (note: `HerReplayBuffer` currently not supported)
- Handle timeout termination properly for on-policy algorithms (when using `TimeLimit`)
- Added `skip` option to `VecTransposeImage` to skip transforming the channel order when the heuristic is wrong
- Added `copy()` and `combine()` methods to `RunningMeanStd`

SB3-Contrib

- Added Trust Region Policy Optimization (TRPO) (@cyprienc)
- Added Augmented Random Search (ARS) (@sgillen)
- Coming soon: PPO LSTM, see https://github.com/Stable-Baselines-Team/stable-baselines3-contrib/pull/53

Bug Fixes:

- Fixed a bug where `set_env()` with `VecNormalize` would result in an error with off-policy algorithms (thanks @cleversonahum)
- FPS calculation is now performed based on number of steps performed during last `learn` call, even when `reset_num_timesteps` is set to `False` (@kachayev)
- Fixed evaluation script for recurrent policies (experimental feature in SB3 contrib)
- Fixed a bug where the observation would be incorrectly detected as non-vectorized instead of throwing an error
- The env checker now properly checks and warns about potential issues for continuous action spaces when the boundaries are too small or when the dtype is not float32
- Fixed a bug in `VecFrameStack` with channel first image envs, where the terminal observation would be wrongly created.

Deprecations:

Others:

- Added a warning in the env checker when not using `np.float32` for continuous actions
- Improved test coverage and error message when checking shape of observation
- Added `newline="\n"` when opening CSV monitor files so that each line ends with `\r\n` instead of `\r\r\n` on Windows while Linux environments are not affected (@hsuehch)
- Fixed `device` argument inconsistency (@qgallouedec)

Documentation:

- Add drivergym to projects page (@theDebugger811)
- Add highway-env to projects page (@eleurent)
- Add tactile-gym to projects page (@ac-93)
- Fix indentation in the RL tips page (@cove9988)
- Update GAE computation docstring
- Add documentation on exporting to TFLite/Coral

- Added JMLR paper and updated citation
- Added link to RL Tips and Tricks video
- Updated `BaseAlgorithm.load` docstring (@Demetrio92)
- Added a note on `load` behavior in the examples (@Demetrio92)
- Updated SB3 Contrib doc
- Fixed A2C and migration guide guidance on how to set epsilon with RMSpropTFLike (@thomasgubler)
- Fixed custom policy documentation (@IperGiove)
- Added doc on Weights & Biases integration

1.40.19 Release 1.3.0 (2021-10-23)

Bug fixes and improvements for the user

> ⚠ **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.

Breaking Changes:

- `sde_net_arch` argument in policies is deprecated and will be removed in a future version.
- `_get_latent` (`ActorCriticPolicy`) was removed
- All logging keys now use underscores instead of spaces (@timokau). Concretely this changes:
  - `time/total timesteps` to `time/total_timesteps` for off-policy algorithms (PPO and A2C) and the eval callback (on-policy algorithms already used the underscored version),
  - `rollout/exploration rate` to `rollout/exploration_rate` and
  - `rollout/success rate` to `rollout/success_rate`.

New Features:

- Added methods `get_distribution` and `predict_values` for `ActorCriticPolicy` for A2C/PPO/TRPO (@cyprienc)
- Added methods `forward_actor` and `forward_critic` for `MlpExtractor`
- Added `sb3.get_system_info()` helper function to gather version information relevant to SB3 (e.g., Python and PyTorch version)
- Saved models now store system information where agent was trained, and load functions have `print_system_info` parameter to help debugging load issues

Bug Fixes:

- Fixed `dtype` of observations for `SimpleMultiObsEnv`
- Allow *VecNormalize* to wrap discrete-observation environments to normalize reward when observation normalization is disabled
- Fixed a bug where `DQN` would throw an error when using `Discrete` observation and stochastic actions
- Fixed a bug where sub-classed observation spaces could not be used

- Added `force_reset` argument to `load()` and `set_env()` in order to be able to call `learn(reset_num_timesteps=False)` with a new environment

Deprecations:

Others:

- Cap gym max version to 0.19 to avoid issues with atari-py and other breaking changes
- Improved error message when using dict observation with the wrong policy
- Improved error message when using `EvalCallback` with two envs not wrapped the same way.
- Added additional infos about supported python version for PyPi in `setup.py`

Documentation:

- Add Rocket League Gym to list of supported projects (@AechPro)
- Added gym-electric-motor to project page (@wkirgsn)
- Added policy-distillation-baselines to project page (@CUN-bjy)
- Added ONNX export instructions (@batu)
- Update read the doc env (fixed `docutils` issue)
- Fix PPO environment name (@IljaAvadiev)
- Fix custom env doc and add env registration example
- Update algorithms from SB3 Contrib
- Use underscores for numeric literals in examples to improve clarity

1.40.20 Release 1.2.0 (2021-09-03)

Hotfix for VecNormalize, training/eval mode support

Breaking Changes:

- SB3 now requires PyTorch >= 1.8.1
- `VecNormalize` `ret` attribute was renamed to `returns`

New Features:

Bug Fixes:

- Hotfix for `VecNormalize` where the observation filter was not updated at reset (thanks @vwxyzjn)
- Fixed model predictions when using batch normalization and dropout layers by calling `train()` and `eval()` (@davidblom603)
- Fixed model training for DQN, TD3 and SAC so that their target nets always remain in evaluation mode (@ayeright)
- Passing `gradient_steps=0` to an off-policy algorithm will result in no gradient steps being taken (vs as many gradient steps as steps done in the environment during the rollout in previous versions)

Deprecations:

Others:

- Enabled Python 3.9 in GitHub CI
- Fixed type annotations
- Refactored `predict()` by moving the preprocessing to `obs_to_tensor()` method

Documentation:

- Updated multiprocessing example
- Added example of `VecEnvWrapper`
- Added a note about logging to tensorboard more often
- Added warning about simplicity of examples and link to RL zoo (@MihaiAnca13)

1.40.21 Release 1.1.0 (2021-07-01)

Dict observation support, timeout handling and refactored HER buffer

Breaking Changes:

- All customs environments (e.g. the `BitFlippingEnv` or `IdentityEnv`) were moved to `stable_baselines3.common.envs` folder
- Refactored `HER` which is now the `HerReplayBuffer` class that can be passed to any off-policy algorithm
- Handle timeout termination properly for off-policy algorithms (when using `TimeLimit`)
- Renamed `_last_dones` and `dones` to `_last_episode_starts` and `episode_starts` in `RolloutBuffer`.
- Removed `ObsDictWrapper` as `Dict` observation spaces are now supported

```
her_kwargs = dict(n_sampled_goal=2, goal_selection_strategy="future", online_
↪sampling=True)
# SB3 < 1.1.0
# model = HER("MlpPolicy", env, model_class=SAC, **her_kwargs)
# SB3 >= 1.1.0:
model = SAC("MultiInputPolicy", env, replay_buffer_class=HerReplayBuffer, replay_buffer_
↪kwargs=her_kwargs)
```

- Updated the KL Divergence estimator in the PPO algorithm to be positive definite and have lower variance (@09tangriro)
- Updated the KL Divergence check in the PPO algorithm to be before the gradient update step rather than after end of epoch (@09tangriro)
- Removed parameter `channels_last` from `is_image_space` as it can be inferred.
- The logger object is now an attribute `model.logger` that be set by the user using `model.set_logger()`
- Changed the signature of `logger.configure` and `utils.configure_logger`, they now return a `Logger` object
- Removed `Logger.CURRENT` and `Logger.DEFAULT`
- Moved `warn(), debug(), log(), info(), dump()` methods to the `Logger` class
- `.learn()` now throws an import error when the user tries to log to tensorboard but the package is not installed

New Features:

- Added support for single-level Dict observation space (@JadenTravnik)
- Added DictRolloutBuffer DictReplayBuffer to support dictionary observations (@JadenTravnik)
- Added StackedObservations and StackedDictObservations that are used within VecFrameStack
- Added simple 4x4 room Dict test environments
- HerReplayBuffer now supports VecNormalize when online_sampling=False
- Added VecMonitor and VecExtractDictObs wrappers to handle gym3-style vectorized environments (@vwxyzjn)
- Ignored the terminal observation if the it is not provided by the environment such as the gym3-style vectorized environments. (@vwxyzjn)
- Added policy_base as input to the OnPolicyAlgorithm for more flexibility (@09tangriro)
- Added support for image observation when using HER
- Added replay_buffer_class and replay_buffer_kwargs arguments to off-policy algorithms
- Added kl_divergence helper for Distribution classes (@09tangriro)
- Added support for vector environments with num_envs > 1 (@benblack769)
- Added wrapper_kwargs argument to make_vec_env (@amy12xx)

Bug Fixes:

- Fixed potential issue when calling off-policy algorithms with default arguments multiple times (the size of the replay buffer would be the same)
- Fixed loading of ent_coef for SAC and TQC, it was not optimized anymore (thanks @Atlis)
- Fixed saving of A2C and PPO policy when using gSDE (thanks @liusida)
- Fixed a bug where no output would be shown even if verbose>=1 after passing verbose=0 once
- Fixed observation buffers dtype in DictReplayBuffer (@c-rizz)
- Fixed EvalCallback tensorboard logs being logged with the incorrect timestep. They are now written with the timestep at which they were recorded. (@skandermoalla)

Deprecations:

Others:

- Added flake8-bugbear to tests dependencies to find likely bugs
- Updated env_checker to reflect support of dict observation spaces
- Added Code of Conduct
- Added tests for GAE and lambda return computation
- Updated distribution entropy test (thanks @09tangriro)
- Added sanity check batch_size > 1 in PPO to avoid NaN in advantage normalization

Documentation:

- Added gym pybullet drones project (@JacopoPan)
- Added link to SuperSuit in projects (@justinkterry)
- Fixed DQN example (thanks @ltbd78)

- Clarified channel-first/channel-last recommendation
- Update sphinx environment installation instructions (@tom-doerr)
- Clarified pip installation in Zsh (@tom-doerr)
- Clarified return computation for on-policy algorithms (TD(lambda) estimate was used)
- Added example for using `ProcgenEnv`
- Added note about advanced custom policy example for off-policy algorithms
- Fixed DQN unicode checkmarks
- Updated migration guide (@juancroldan)
- Pinned `docutils==0.16` to avoid issue with rtd theme
- Clarified callback `save_freq` definition
- Added doc on how to pass a custom logger
- Remove recurrent policies from `A2C` docs (@bstee615)

1.40.22 Release 1.0 (2021-03-15)

First Major Version

Breaking Changes:

- Removed `stable_baselines3.common.cmd_util` (already deprecated), please use `env_util` instead

⚠ **Warning**

A refactoring of the `HER` algorithm is planned together with support for dictionary observations (see PR #243 and #351) This will be a backward incompatible change (model trained with previous version of `HER` won't work with the new version).

New Features:

- Added support for `custom_objects` when loading models

Bug Fixes:

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

Documentation:

- Fixed examples
- Added new project using SB3: rl_reach (@PierreExeter)
- Added note about slow-down when switching to PyTorch
- Add a note on continual learning and resetting environment

Others:

- Updated RL-Zoo to reflect the fact that is it more than a collection of trained agents
- Added images to illustrate the training loop and custom policies (created with https://excalidraw.com/)
- Updated the custom policy section

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

Bug Fixes:

- Fixed a bug where `train_freq` was not properly converted when loading a saved model

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

Breaking Changes:

- `evaluate_policy` now returns rewards/episode lengths from a `Monitor` wrapper if one is present, this allows to return the unnormalized reward in the case of Atari games for instance.
- Renamed `common.vec_env.is_wrapped` to `common.vec_env.is_vecenv_wrapped` to avoid confusion with the new `is_wrapped()` helper
- Renamed `_get_data()` to `_get_constructor_parameters()` for policies (this affects independent saving/loading of policies)
- Removed `n_episodes_rollout` and merged it with `train_freq`, which now accepts a tuple `(frequency, unit)`:
- `replay_buffer` in `collect_rollout` is no more optional

```
# SB3 < 0.11.0
# model = SAC("MlpPolicy", env, n_episodes_rollout=1, train_freq=-1)
# SB3 >= 0.11.0:
model = SAC("MlpPolicy", env, train_freq=(1, "episode"))
```

New Features:

- Add support for `VecFrameStack` to stack on first or last observation dimension, along with automatic check for image spaces.
- `VecFrameStack` now has a `channels_order` argument to tell if observations should be stacked on the first or last observation dimension (originally always stacked on last).
- Added `common.env_util.is_wrapped` and `common.env_util.unwrap_wrapper` functions for checking/unwrapping an environment for specific wrapper.
- Added `env_is_wrapped()` method for `VecEnv` to check if its environments are wrapped with given Gym wrappers.
- Added `monitor_kwargs` parameter to `make_vec_env` and `make_atari_env`
- Wrap the environments automatically with a `Monitor` wrapper when possible.
- `EvalCallback` now logs the success rate when available (`is_success` must be present in the info dict)
- Added new wrappers to log images and matplotlib figures to tensorboard. (@zampanteymedio)
- Add support for text records to `Logger`. (@lorenz-h)

Bug Fixes:

- Fixed bug where code added VecTranspose on channel-first image environments (thanks @qxcv)
- Fixed `DQN` predict method when using single `gym.Env` with `deterministic=False`
- Fixed bug that the arguments order of `explained_variance()` in `ppo.py` and `a2c.py` is not correct (@thisray)
- Fixed bug where full `HerReplayBuffer` leads to an index error. (@megan-klaiber)

- Fixed bug where replay buffer could not be saved if it was too big (> 4 Gb) for python<3.8 (thanks @hn2)
- Added informative PPO construction error in edge-case scenario where n_steps * n_envs = 1 (size of rollout buffer), which otherwise causes downstream breaking errors in training (@decodyng)
- Fixed discrete observation space support when using multiple envs with A2C/PPO (thanks @ardabbour)
- Fixed a bug for TD3 delayed update (the update was off-by-one and not delayed when train_freq=1)
- Fixed numpy warning (replaced np.bool with bool)
- Fixed a bug where VecNormalize was not normalizing the terminal observation
- Fixed a bug where VecTranspose was not transposing the terminal observation
- Fixed a bug where the terminal observation stored in the replay buffer was not the right one for off-policy algorithms
- Fixed a bug where action_noise was not used when using HER (thanks @ShangqunYu)

Deprecations:

Others:

- Add more issue templates
- Add signatures to callable type annotations (@ernestum)
- Improve error message in NatureCNN
- Added checks for supported action spaces to improve clarity of error messages for the user
- Renamed variables in the train() method of SAC, TD3 and DQN to match SB3-Contrib.
- Updated docker base image to Ubuntu 18.04
- Set tensorboard min version to 2.2.0 (earlier version are apparently not working with PyTorch)
- Added warning for PPO when n_steps * n_envs is not a multiple of batch_size (last mini-batch truncated) (@decodyng)
- Removed some warnings in the tests

Documentation:

- Updated algorithm table
- Minor docstring improvements regarding rollout (@stheid)
- Fix migration doc for A2C (epsilon parameter)
- Fix clip_range docstring
- Fix duplicated parameter in EvalCallback docstring (thanks @tfederico)
- Added example of learning rate schedule
- Added SUMO-RL as example project (@LucasAlegre)
- Fix docstring of classes in atari_wrappers.py which were inside the constructor (@LucasAlegre)
- Added SB3-Contrib page
- Fix bug in the example code of DQN (@AptX395)
- Add example on how to access the tensorboard summary writer directly. (@lorenz-h)
- Updated migration guide

- Updated custom policy doc (separate policy architecture recommended)
- Added a note about OpenCV headless version
- Corrected typo on documentation (@mschweizer)
- Provide the environment when loading the model in the examples (@lorepieri8)

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

HER with online and offline sampling, bug fixes for features extraction

Breaking Changes:

- **Warning:** Renamed `common.cmd_util` to `common.env_util` for clarity (affects `make_vec_env` and `make_atari_env` functions)

New Features:

- Allow custom actor/critic network architectures using `net_arch=dict(qf=[400, 300], pi=[64, 64])` for off-policy algorithms (SAC, TD3, DDPG)
- Added Hindsight Experience Replay `HER`. (@megan-klaiber)
- `VecNormalize` now supports `gym.spaces.Dict` observation spaces
- Support logging videos to Tensorboard (@SwamyDev)
- Added `share_features_extractor` argument to `SAC` and `TD3` policies

Bug Fixes:

- Fix GAE computation for on-policy algorithms (off-by one for the last value) (thanks @Wovchena)
- Fixed potential issue when loading a different environment
- Fix ignoring the exclude parameter when recording logs using json, csv or log as logging format (@SwamyDev)
- Make `make_vec_env` support the `env_kwargs` argument when using an env ID str (@ManifoldFR)
- Fix model creation initializing CUDA even when *device="cpu"* is provided
- Fix `check_env` not checking if the env has a Dict actionspace before calling `_check_nan` (@wmmc88)
- Update the check for spaces unsupported by Stable Baselines 3 to include checks on the action space (@wmmc88)
- Fixed features extractor bug for target network where the same net was shared instead of being separate. This bug affects `SAC`, `DDPG` and `TD3` when using `CnnPolicy` (or custom features extractor)
- Fixed a bug when passing an environment when loading a saved model with a `CnnPolicy`, the passed env was not wrapped properly (the bug was introduced when implementing `HER` so it should not be present in previous versions)

Deprecations:

Others:

- Improved typing coverage
- Improved error messages for unsupported spaces
- Added `.vscode` to the gitignore

Documentation:

- Added first draft of migration guide
- Added intro to imitation library (@shwang)
- Enabled doc for `CnnPolicies`
- Added advanced saving and loading example
- Added base doc for exporting models
- Added example for getting and setting model parameters

1.40.26 Pre-Release 0.9.0 (2020-10-03)

Bug fixes, get/set parameters and improved docs

Breaking Changes:

- Removed `device` keyword argument of policies; use `policy.to(device)` instead. (@qxcv)
- Rename `BaseClass.get_torch_variables` -> `BaseClass._get_torch_save_params` and `BaseClass.excluded_save_params` -> `BaseClass._excluded_save_params`
- Renamed saved items `tensors` to `pytorch_variables` for clarity
- `make_atari_env`, `make_vec_env` and `set_random_seed` must be imported with (and not directly from `stable_baselines3.common`):

```
from stable_baselines3.common.cmd_util import make_atari_env, make_vec_env
from stable_baselines3.common.utils import set_random_seed
```

New Features:

- Added `unwrap_vec_wrapper()` to `common.vec_env` to extract `VecEnvWrapper` if needed
- Added `StopTrainingOnMaxEpisodes` to callback collection (@xicocaio)
- Added `device` keyword argument to `BaseAlgorithm.load()` (@liorcohen5)
- Callbacks have access to rollout collection locals as in SB2. (@PartiallyTyped)
- Added `get_parameters` and `set_parameters` for accessing/setting parameters of the agent
- Added actor/critic loss logging for TD3. (@mloo3)

Bug Fixes:

- Added `unwrap_vec_wrapper()` to `common.vec_env` to extract `VecEnvWrapper` if needed
- Fixed a bug where the environment was reset twice when using `evaluate_policy`
- Fix logging of `clip_fraction` in PPO (@diditforlulz273)
- Fixed a bug where cuda support was wrongly checked when passing the GPU index, e.g., `device="cuda:0"` (@liorcohen5)
- Fixed a bug when the random seed was not properly set on cuda when passing the GPU index

Deprecations:

Others:

- Improve typing coverage of the `VecEnv`
- Fix type annotation of `make_vec_env` (@ManifoldFR)
- Removed `AlreadySteppingError` and `NotSteppingError` that were not used
- Fixed typos in SAC and TD3
- Reorganized functions for clarity in `BaseClass` (save/load functions close to each other, private functions at top)
- Clarified docstrings on what is saved and loaded to/from files
- Simplified `save_to_zip_file` function by removing duplicate code
- Store library version along with the saved models
- DQN loss is now logged

Documentation:

- Added `StopTrainingOnMaxEpisodes` details and example (@xicocaio)
- Updated custom policy section (added custom features extractor example)
- Re-enable `sphinx_autodoc_typehints`
- Updated doc style for type hints and remove duplicated type hints

1.40.27 Pre-Release 0.8.0 (2020-08-03)

DQN, DDPG, bug fixes and performance matching for Atari games

Breaking Changes:

- `AtariWrapper` and other Atari wrappers were updated to match SB2 ones
- `save_replay_buffer` now receives as argument the file path instead of the folder path (@tirafesi)
- Refactored `Critic` class for `TD3` and `SAC`, it is now called `ContinuousCritic` and has an additional parameter `n_critics`
- `SAC` and `TD3` now accept an arbitrary number of critics (e.g. `policy_kwargs=dict(n_critics=3)`) instead of only 2 previously

New Features:

- Added `DQN` Algorithm (@Artemis-Skade)
- Buffer dtype is now set according to action and observation spaces for `ReplayBuffer`
- Added warning when allocation of a buffer may exceed the available memory of the system when `psutil` is available
- Saving models now automatically creates the necessary folders and raises appropriate warnings (@PartiallyTyped)
- Refactored opening paths for saving and loading to use strings, pathlib or io.BufferedIOBase (@PartiallyTyped)
- Added `DDPG` algorithm as a special case of `TD3`.
- Introduced `BaseModel` abstract parent for `BasePolicy`, which critics inherit from.

Bug Fixes:

- Fixed a bug in the `close()` method of `SubprocVecEnv`, causing wrappers further down in the wrapper stack to not be closed. (@NeoExtended)
- Fix target for updating q values in SAC: the entropy term was not conditioned by terminals states
- Use `cloudpickle.load` instead of `pickle.load` in `CloudpickleWrapper`. (@shwang)
- Fixed a bug with orthogonal initialization when *bias=False* in custom policy (@rk37)
- Fixed approximate entropy calculation in PPO and A2C. (@andyshih12)
- Fixed DQN target network sharing features extractor with the main network.
- Fixed storing correct `dones` in on-policy algorithm rollout collection. (@andyshih12)
- Fixed number of filters in final convolutional layer in NatureCNN to match original implementation.

Deprecations:

Others:

- Refactored off-policy algorithm to share the same `.learn()` method
- Split the `collect_rollout()` method for off-policy algorithms
- Added `_on_step()` for off-policy base class
- Optimized replay buffer size by removing the need of `next_observations` numpy array
- Optimized polyak updates (1.5-1.95 speedup) through inplace operations (@PartiallyTyped)
- Switch to `black` codestyle and added `make format`, `make check-codestyle` and `commit-checks`
- Ignored errors from newer pytype version
- Added a check when using `gSDE`
- Removed codacy dependency from Dockerfile
- Added `common.sb2_compat.RMSpropTFLike` optimizer, which corresponds closer to the implementation of RMSprop from Tensorflow.

Documentation:

- Updated notebook links
- Fixed a typo in the section of Enjoy a Trained Agent, in RL Baselines3 Zoo README. (@blurLake)
- Added Unity reacher to the projects page (@koulakis)
- Added PyBullet colab notebook
- Fixed typo in PPO example code (@joeljosephjin)
- Fixed typo in custom policy doc (@RaphaelWag)

1.40.28 Pre-Release 0.7.0 (2020-06-10)

Hotfix for PPO/A2C + gSDE, internal refactoring and bug fixes

Breaking Changes:

- render() method of VecEnvs now only accept one argument: mode
- Created new file common/torch_layers.py, similar to SB refactoring
  - Contains all PyTorch network layer definitions and features extractors: MlpExtractor, create_mlp, NatureCNN
- Renamed BaseRLModel to BaseAlgorithm (along with offpolicy and onpolicy variants)
- Moved on-policy and off-policy base algorithms to common/on_policy_algorithm.py and common/off_policy_algorithm.py, respectively.
- Moved PPOPolicy to ActorCriticPolicy in common/policies.py
- Moved PPO (algorithm class) into OnPolicyAlgorithm (common/on_policy_algorithm.py), to be shared with A2C
- Moved following functions from BaseAlgorithm:
  - _load_from_file to load_from_zip_file (save_util.py)
  - _save_to_file_zip to save_to_zip_file (save_util.py)
  - safe_mean to safe_mean (utils.py)
  - check_env to check_for_correct_spaces (utils.py. Renamed to avoid confusion with environment checker tools)
- Moved static function _is_vectorized_observation from common/policies.py to common/utils.py under name is_vectorized_observation.
- Removed {save,load}_running_average functions of VecNormalize in favor of load/save.
- Removed use_gae parameter from RolloutBuffer.compute_returns_and_advantage.

New Features:

Bug Fixes:

- Fixed render() method for VecEnvs
- Fixed seed() method for SubprocVecEnv
- Fixed loading on GPU for testing when using gSDE and deterministic=False
- Fixed register_policy to allow re-registering same policy for same sub-class (i.e. assign same value to same key).
- Fixed a bug where the gradient was passed when using gSDE with PPO/A2C, this does not affect SAC

Deprecations:

Others:

- Re-enable unsafe fork start method in the tests (was causing a deadlock with tensorflow)
- Added a test for seeding SubprocVecEnv and rendering
- Fixed reference in NatureCNN (pointed to older version with different network architecture)
- Fixed comments saying “CxWxH” instead of “CxHxW” (same style as in torch docs / commonly used)
- Added bit further comments on register/getting policies (“MlpPolicy”, “CnnPolicy”).
- Renamed progress (value from 1 in start of training to 0 in end) to progress_remaining.

- Added `policies.py` files for A2C/PPO, which define MlpPolicy/CnnPolicy (renamed ActorCriticPolicies).
- Added some missing tests for `VecNormalize`, `VecCheckNan` and `PPO`.

Documentation:

- Added a paragraph on "MlpPolicy"/"CnnPolicy" and policy naming scheme under "Developer Guide"
- Fixed second-level listing in changelog

1.40.29 Pre-Release 0.6.0 (2020-06-01)

Tensorboard support, refactored logger

Breaking Changes:

- Remove State-Dependent Exploration (SDE) support for `TD3`
- Methods were renamed in the logger:
  - `logkv` -> `record`, `writekvs` -> `write`, `writeseq` -> `write_sequence`,
  - `logkvs` -> `record_dict`, `dumpkvs` -> `dump`,
  - `getkvs` -> `get_log_dict`, `logkv_mean` -> `record_mean`,

New Features:

- Added env checker (Sync with Stable Baselines)
- Added `VecCheckNan` and `VecVideoRecorder` (Sync with Stable Baselines)
- Added determinism tests
- Added `cmd_util` and `atari_wrappers`
- Added support for `MultiDiscrete` and `MultiBinary` observation spaces (@rolandgvc)
- Added `MultiCategorical` and `Bernoulli` distributions for PPO/A2C (@rolandgvc)
- Added support for logging to tensorboard (@rolandgvc)
- Added `VectorizedActionNoise` for continuous vectorized environments (@PartiallyTyped)
- Log evaluation in the `EvalCallback` using the logger

Bug Fixes:

- Fixed a bug that prevented model trained on cpu to be loaded on gpu
- Fixed version number that had a new line included
- Fixed weird seg fault in docker image due to FakeImageEnv by reducing screen size
- Fixed `sde_sample_freq` that was not taken into account for SAC
- Pass logger module to `BaseCallback` otherwise they cannot write in the one used by the algorithms

Deprecations:

Others:

- Renamed to Stable-Baseline3
- Added Dockerfile

- Sync `VecEnvs` with Stable-Baselines
- Update requirement: `gym>=0.17`
- Added `.readthedoc.yml` file
- Added `flake8` and `make lint` command
- Added Github workflow
- Added warning when passing both `train_freq` and `n_episodes_rollout` to Off-Policy Algorithms

Documentation:

- Added most documentation (adapted from Stable-Baselines)
- Added link to CONTRIBUTING.md in the README (@kinalmehta)
- Added gSDE project and update docstrings accordingly
- Fix `TD3` example code block

1.40.30 Pre-Release 0.5.0 (2020-05-05)

CnnPolicy support for image observations, complete saving/loading for policies

Breaking Changes:

- Previous loading of policy weights is broken and replace by the new saving/loading for policy

New Features:

- Added `optimizer_class` and `optimizer_kwargs` to `policy_kwargs` in order to easily customizer optimizers
- Complete independent save/load for policies
- Add `CnnPolicy` and `VecTransposeImage` to support images as input

Bug Fixes:

- Fixed `reset_num_timesteps` behavior, so `env.reset()` is not called if `reset_num_timesteps=True`
- Fixed `squashed_output` that was not pass to policy constructor for `SAC` and `TD3` (would result in scaled actions for unscaled action spaces)

Deprecations:

Others:

- Cleanup rollout return
- Added `get_device` util to manage PyTorch devices
- Added type hints to logger + use f-strings

Documentation:

1.40.31 Pre-Release 0.4.0 (2020-02-14)

Proper pre-processing, independent save/load for policies

Breaking Changes:

- Removed CEMRL
- Model saved with previous versions cannot be loaded (because of the pre-preprocessing)

New Features:

- Add support for `Discrete` observation spaces
- Add saving/loading for policy weights, so the policy can be used without the model

Bug Fixes:

- Fix type hint for activation functions

Deprecations:

Others:

- Refactor handling of observation and action spaces
- Refactored features extraction to have proper preprocessing
- Refactored action distributions

1.40.32 Pre-Release 0.3.0 (2020-02-14)

Bug fixes, sync with Stable-Baselines, code cleanup

Breaking Changes:

- Removed default seed
- Bump dependencies (PyTorch and Gym)
- `predict()` now returns a tuple to match Stable-Baselines behavior

New Features:

- Better logging for `SAC` and `PPO`

Bug Fixes:

- Synced callbacks with Stable-Baselines
- Fixed colors in `results_plotter`
- Fix entropy computation (now summed over action dim)

Others:

- SAC with SDE now sample only one matrix
- Added `clip_mean` parameter to SAC policy
- Buffers now return `NamedTuple`
- More typing
- Add test for `expln`
- Renamed `learning_rate` to `lr_schedule`

- Add `version.txt`
- Add more tests for distribution

Documentation:

- Deactivated `sphinx_autodoc_typehints` extension

1.40.33 Pre-Release 0.2.0 (2020-02-14)

Python 3.6+ required, type checking, callbacks, doc build

Breaking Changes:

- Python 2 support was dropped, Stable Baselines3 now requires Python 3.6 or above
- Return type of `evaluation.evaluate_policy()` has been changed
- Refactored the replay buffer to avoid transformation between PyTorch and NumPy
- Created *OffPolicyRLModel* base class
- Remove deprecated JSON format for *Monitor*

New Features:

- Add `seed()` method to `VecEnv` class
- Add support for Callback (cf https://github.com/hill-a/stable-baselines/pull/644)
- Add methods for saving and loading replay buffer
- Add `extend()` method to the buffers
- Add `get_vec_normalize_env()` to `BaseRLModel` to retrieve `VecNormalize` wrapper when it exists
- Add `results_plotter` from Stable Baselines
- Improve `predict()` method to handle different type of observations (single, vectorized, …)

Bug Fixes:

- Fix loading model on CPU that were trained on GPU
- Fix `reset_num_timesteps` that was not used
- Fix entropy computation for squashed Gaussian (approximate it now)
- Fix seeding when using multiple environments (different seed per env)

Others:

- Add type check
- Converted all format string to f-strings
- Add test for `OrnsteinUhlenbeckActionNoise`
- Add type aliases in `common.type_aliases`

Documentation:

- fix documentation build

1.40.34 Pre-Release 0.1.0 (2020-01-20)

First Release: base algorithms and state-dependent exploration

New Features:

- Initial release of A2C, CEM-RL, PPO, SAC and TD3, working only with `Box` input space
- State-Dependent Exploration (SDE) for A2C, PPO, SAC and TD3

1.40.35 Maintainers

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

1.40.36 Contributors:

In random order…

Thanks to the maintainers of V2: @hill-a @ernestum @AdamGleave @Miffyli

And all the contributors: @taymuur @bjmuld @iambenzo @iandanforth @r7vme @brendenpetersen @huvar @abhiskk @JohannesAck @EliasHasle @mrakgr @Bleyddyn @antoine-galataud @junhyeokahn @AdamGleave @keshaviyengar @tperol @XMaster96 @kantneel @Pastafarianist @GerardMaggiolino @PatrickWalter214 @yutingsz @sc420 @Aaahh @billtubbs @Miffyli @dwiel @miguelrass @qxcv @jaberkow @eavelardev @ruifeng96150 @pedrohbtp @srivatsankrishnan @evilsocket @MarvineGothic @jdossgollin @stheid @SyllogismRXS @rusu24edward @jbulow @Antymon @seheevic @justinkterry @edbeeching @flodorner @KuKuXia @NeoExtended @PartiallyTyped @mmcenta @richardwu @kinalmehta @rolandgvc @tkelestemur @mloo3 @tirafesi @blurLake @koulakis @joeljosephjin @shwang @rk37 @andyshih12 @RaphaelWag @xicocaio @diditforlulz273 @liorcohen5 @ManifoldFR @mloo3 @SwamyDev @wmmc88 @megan-klaiber @thisray @tfederico @hn2 @LucasAlegre @AptX395 @zampanteymedio @fracapuano @JadenTravnik @decodyng @ardabbour @lorenz-h @mschweizer @lorepieri8 @vwxyzjn @ShangqunYu @PierreExeter @JacopoPan @ltbd78 @tom-doerr @Atlis @liusida @09tangriro @amy12xx @juancroldan @benblack769 @bstee615 @c-rizz @skandermoalla @MihaiAnca13 @davidblom603 @ayeright @cyprienc @wkirgsn @AechPro @CUN-bjy @batu @IljaAvadiev @timokau @kachayev @cleversonahum @eleurent @ac-93 @cove9988 @theDebugger811 @hsuehch @Demetrio92 @thomasgubler @IperGiove @ScheiklP @simoninithomas @armandpl @manuel-delverme @Gautam-J @gianlucadecola @buoyancy99 @caburu @xy9485 @Gregwar @ycheng517 @quantitative-technologies @bcollazo @git-thor @TibiGG @cool-RR @MWeltevrede @carlosluis @arjun-kg @tlpss @JonathanKuelz @Gabo-Tor @iwishiwasaneagle @Melanol @qgallouedec @francescoluciano @jlp-ue @burakdmb @timothe-chaumont @honglu2875 @anand-bala @hughperkins @sidney-tio @AlexPasqua @dominicgkerr @Akhilez @Rocamonde @tobirohrer @ZikangXiong @ReHoss @DavyMorgan @luizapozzobon @Bonifatius94 @theSquaredError @harveybellini @DavyMorgan @FieteO @jonasreiher @npit @WeberSamuel @troiganto @lutogniew @lbergmann1 @lukashass @BertrandDecoster @pseudo-rnd-thoughts @stefanbschneider @kyle-he @PatrickHelm @corentinlger @marekm4 @stagoverflow @rushitnshah @markscsmith @NickLucche @cschindlbeck @peteole @jak3122 @will-maclean @brn-dev @jmacglashan @kplers @MarcDcls @chrisgao99 @pstahlhofen @akanto

1.41 Projects

This is a list of projects using stable-baselines3. Please tell us, if you want your project to appear on this page ;)

1.41.1 DriverGym

An open-source Gym-compatible environment specifically tailored for developing RL algorithms for autonomous driving. DriverGym provides access to more than 1000 hours of expert logged data and also supports reactive and data-driven agent behavior. The performance of an RL policy can be easily validated using an extensive and flexible closed-loop evaluation protocol. We also provide behavior cloning baselines using supervised learning and RL, trained in DriverGym.

Authors: Parth Kothari, Christian Perone, Luca Bergamini, Alexandre Alahi, Peter Ondruska
Github: https://github.com/lyft/l5kit
Paper: https://arxiv.org/abs/2111.06889

1.41.2 RL Reach

A platform for running reproducible reinforcement learning experiments for customizable robotic reaching tasks. This self-contained and straightforward toolbox allows its users to quickly investigate and identify optimal training configurations.

Authors: Pierre Aumjaud, David McAuliffe, Francisco Javier Rodríguez Lera, Philip Cardiff
Github: https://github.com/PierreExeter/rl_reach
Paper: https://arxiv.org/abs/2102.04916

1.41.3 Generalized State Dependent Exploration for Deep Reinforcement Learning in Robotics

An exploration method to train RL agent directly on real robots. It was the starting point of Stable-Baselines3.

Author: Antonin Raffin, Freek Stulp
Github: https://github.com/DLR-RM/stable-baselines3/tree/sde
Paper: https://arxiv.org/abs/2005.05719

1.41.4 Furuta Pendulum Robot

Everything you need to build and train a rotary inverted pendulum, also known as a furuta pendulum! This makes use of gSDE listed above. The Github repository contains code, CAD files and a bill of materials for you to build the robot. You can watch a video overview of the project here.

Authors: Armand du Parc Locmaria, Pierre Fabre
Github: https://github.com/Armandpl/furuta

1.41.5 Reacher

A solution to the second project of the Udacity deep reinforcement learning course. It is an example of:

- wrapping single and multi-agent Unity environments to make them usable in Stable-Baselines3

- creating experimentation scripts which train and run A2C, PPO, TD3 and SAC models (a better choice for this one is https://github.com/DLR-RM/rl-baselines3-zoo)
- generating several pre-trained models which solve the reacher environment

Author: Marios Koulakis
Github: https://github.com/koulakis/reacher-deep-reinforcement-learning

1.41.6 SUMO-RL

A simple interface to instantiate RL environments with SUMO for Traffic Signal Control.

- Supports Multiagent RL
- Compatibility with gym.Env and popular RL libraries such as stable-baselines3 and RLlib
- Easy customization: state and reward definitions are easily modifiable

Author: Lucas Alegre
Github: https://github.com/LucasAlegre/sumo-rl

1.41.7 gym-pybullet-drones

PyBullet Gym environments for single and multi-agent reinforcement learning of quadcopter control.

- Physics-based simulation for the development and test of quadcopter control.
- Compatibility with `gym.Env`, RLlib's MultiAgentEnv.
- Learning and testing script templates for stable-baselines3 and RLlib.

Author: Jacopo Panerati
Github: https://github.com/utiasDSL/gym-pybullet-drones/
Paper: https://arxiv.org/abs/2103.02142

1.41.8 SuperSuit

SuperSuit contains easy to use wrappers for Gym (and multi-agent PettingZoo) environments to do all forms of common preprocessing (frame stacking, converting graphical observations to greyscale, max-and-skip for Atari, etc.). It also notably includes:

-Wrappers that apply lambda functions to observations, actions, or rewards with a single line of code. -All wrappers can be used natively on vector environments, wrappers exist to Gym environments to vectorized environments and concatenate multiple vector environments together -A wrapper is included that allows for using regular single agent RL libraries (e.g. stable baselines) to learn simple multi-agent PettingZoo environments, explained in this tutorial:

Author: Justin Terry
GitHub: https://github.com/PettingZoo-Team/SuperSuit
Tutorial on multi-agent support in stable baselines: https://towardsdatascience.com/multi-agent-deep-reinforcement-learning-in-15-lines-of-code-using-pettingzoo-e0b963c0820b

1.41.9 Rocket League Gym

A fully custom python API and C++ DLL to treat the popular game Rocket League like an OpenAI Gym environment.

- Dramatically increases the rate at which the game runs.
- Supports full configuration of initial states, observations, rewards, and terminal states.
- Supports multiple simultaneous game clients.
- Supports multi-agent training and self-play.
- Provides custom wrappers for easy use with stable-baselines3.

Authors: Lucas Emery, Matthew Allen
GitHub: https://github.com/lucas-emery/rocket-league-gym
Website: https://rlgym.github.io

1.41.10 gym-electric-motor

An OpenAI gym environment for the simulation and control of electric drive trains. Think of Matlab/Simulink for electric motors, inverters, and load profiles, but non-graphical and open-source in Python.

gym-electric-motor offers a rich interface for customization, including - plug-and-play of different control algorithms ranging from classical controllers (like field-oriented control) up to any RL agent you can find, - reward shaping, - load profiling, - finite-set or continuous-set control, - one-phase and three-phase motors such as induction machines and permanent magnet synchronous motors, among others.

SB3 is used as an example in one of many tutorials showcasing the easy usage of *gym-electric-motor*.

Author: Paderborn University, LEA department
GitHub: https://github.com/upb-lea/gym-electric-motor
SB3 Tutorial: Colab Link
Paper: JOSS, TNNLS, ArXiv

1.41.11 policy-distillation-baselines

A PyTorch implementation of Policy Distillation for control, which has well-trained teachers via Stable Baselines3.

- *policy-distillation-baselines* provides some good examples for policy distillation in various environment and using reliable algorithms.
- All well-trained models and algorithms are compatible with Stable Baselines3.

Authors: Junyeob Baek
GitHub: https://github.com/CUN-bjy/policy-distillation-baselines
Demo: link

1.41.12 highway-env

A minimalist environment for decision-making in Autonomous Driving.

Driving policies can be trained in different scenarios, and several notebooks using SB3 are provided as examples.

Author: Edouard Leurent
GitHub: https://github.com/eleurent/highway-env
Examples: Colab Links

1.41.13 tactile-gym

Suite of RL environments focused on using a simulated tactile sensor as the primary source of observations. Sim-to-Real results across 4 out of 5 proposed envs.

Author: Alex Church
GitHub: https://github.com/ac-93/tactile_gym
Paper: https://arxiv.org/abs/2106.08796
Website: tactile-gym website

1.41.14 RLeXplore

RLeXplore is a set of implementations of intrinsic reward driven-exploration approaches in reinforcement learning using PyTorch, which can be deployed in arbitrary algorithms in a plug-and-play manner. In particular, RLeXplore is designed to be well compatible with Stable-Baselines3, providing more stable exploration benchmarks.

- Support arbitrary RL algorithms;
- Highly modular and high expansibility;
- Keep up with the latest research progress.

Author: Mingqi Yuan
GitHub: https://github.com/yuanmingqi/rl-exploration-baselines

1.41.15 UAV_Navigation_DRL_AirSim

A platform for training UAV navigation policies in complex unknown environments.

- Based on AirSim and SB3.
- An Open AI Gym env is created including kinematic models for both multirotor and fixed-wing UAVs.
- Some UE4 environments are provided to train and test the navigation policy.

Try to train your own autonomous flight policy and even transfer it to real UAVs! Have fun ^_^!

Author: Lei He
Github: https://github.com/heleidsn/UAV_Navigation_DRL_AirSim

1.41.16 Pink Noise Exploration

A simple library for pink noise exploration with deterministic (DDPG / TD3) and stochastic (SAC) off-policy algorithms. Pink noise has been shown to work better than uncorrelated Gaussian noise (the default choice) and Ornstein-Uhlenbeck noise on a range of continuous control benchmark tasks. This library is designed to work with Stable Baselines3.

Authors: Onno Eberhard, Jakob Hollenstein, Cristina Pinneri, Georg Martius

Github: https://github.com/martius-lab/pink-noise-rl

Paper: https://openreview.net/forum?id=hQ9V5QN27eS (Oral at ICLR 2023)

1.41.17 mobile-env

An open, minimalist Gymnasium environment for autonomous coordination in wireless mobile networks. It allows simulating various scenarios with moving users in a cellular network with multiple base stations.

- Written in pure Python, easy to modify and extend, and can be installed directly via PyPI.
- Implements the standard Gymnasium interface such that it can be used with all common frameworks for reinforcement learning.
- There are examples for both single-agent and multi-agent RL using either *stable-baselines3* or Ray RLlib.

Authors: Stefan Schneider, Stefan Werner

Github: https://github.com/stefanbschneider/mobile-env

Paper: https://ris.uni-paderborn.de/download/30236/30237 (2022 IEEE/IFIP Network Operations and Management Symposium (NOMS))

1.41.18 DeepNetSlice

A Deep Reinforcement Learning Open-Source Toolkit for Network Slice Placement (NSP).

NSP is the problem of deciding which physical servers in a network should host the virtual network functions (VNFs) that make up a network slice, as well as managing the mapping of the virtual links between the VNFs onto the physical infrastructure. It is a complex optimization problem, as it involves considering the requirements of the network slice and the available resources on the physical network. The goal is generally to maximize the utilization of the physical resources while ensuring that the network slices meet their performance requirements.

The toolkit includes a customizable simulation environments, as well as some ready-to-use demos for training intelligent agents to perform network slice placement.

Author: Alex Pasquali

Github: https://github.com/AlexPasqua/DeepNetSlice

Paper: **under review** (citation instructions on the project's README.md) -> see this Master's Thesis for the moment: https://etd.adm.unipi.it/theses/available/etd-01182023-110038/unrestricted/Tesi_magistrale_Pasquali_Alex.pdf

1.41.19 PokemonRedExperiments

Playing Pokemon Red with Reinforcement Learning.

Author: Peter Whidden
Github: https://github.com/PWhiddy/PokemonRedExperiments
Video: https://www.youtube.com/watch?v=DcYLT37ImBY

1.41.20 Evolving Reservoirs for Meta Reinforcement Learning

Meta-RL framework to optimize reservoir-like neural structures (special kind of RNNs), and integrate them to RL agents to improve their training. It enables solving environments involving partial observability or locomotion (e.g MuJoCo), and optimizing reservoirs that can generalize to unseen tasks.

Authors: Corentin Léger, Gautier Hamon, Eleni Nisioti, Xavier Hinaut, Clément Moulin-Frier
Github: https://github.com/corentinlger/ER-MRL
Paper: https://arxiv.org/abs/2312.06695

1.41.21 FootstepNet Envs

These environments are dedicated to train efficient agents that can plan and forecast bipedal robot footsteps in order to go to a target location possibly avoiding obstacles. They are designed to be used with Reinforcement Learning (RL) algorithms.

Real world experiments were conducted during RoboCup competitions on the Sigmaban robot, a small-sized humanoid designed by the *Rhoban Team*.

Authors: Clément Gaspard, Grégoire Passault, Mélodie Daniel, Olivier Ly
Github: https://github.com/Rhoban/footstepnet_envs
Paper: https://arxiv.org/abs/2403.12589

1.41.22 FRASA: Fall Recovery And Stand up agent

A Deep Reinforcement Learning agent for a humanoid robot that learns to recover from falls and stand up.

The agent is trained using the MuJoCo physics engine. Real world experiments are conducted on the Sigmaban humanoid robot, a small-sized humanoid designed by the *Rhoban Team* to compete in the RoboCup Kidsize League. The results, detailled in the paper and the video, show that the agent is able to recover from various external disturbances and stand up in a few seconds.

Authors: Marc Duclusaud, Clément Gaspard, Grégoire Passault, Mélodie Daniel, Olivier Ly
Github: https://github.com/Rhoban/frasa
Paper: https://arxiv.org/abs/2410.08655
Video: https://www.youtube.com/watch?v=NL65XW0O0mk

CHAPTER

TWO

CITING STABLE BASELINES3

To cite this project in publications:

```
@article{stable-baselines3,
  author  = {Antonin Raffin and Ashley Hill and Adam Gleave and Anssi Kanervisto and Maximilian Ernestus and Noah Dormann},
  title   = {Stable-Baselines3: Reliable Reinforcement Learning Implementations},
  journal = {Journal of Machine Learning Research},
  year    = {2021},
  volume  = {22},
  number  = {268},
  pages   = {1-8},
  url     = {http://jmlr.org/papers/v22/20-1364.html}
}
```

Note: If you need to refer to a specific version of SB3, you can also use the Zenodo DOI.

CHAPTER THREE

CONTRIBUTING

To any interested in making the rl baselines better, there are still some improvements that need to be done. You can check issues in the repository.

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

CHAPTER FOUR

INDICES AND TABLES

- genindex
- search
- modindex

PYTHON MODULE INDEX

INDEX