Stable Baselines3 Documentation

Release 0.8.0

Stable Baselines3 Contributors

Aug 03, 2020

USER GUIDE

Stable Baselines3 is a set of improved 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

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

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

CHAPTER

ONE

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

1.1 Installation

1.1.1 Prerequisites

Stable-Baselines3 requires python 3.6+.

Windows 10

We recommend using Anaconda for Windows users for easier installation of Python packages and required libraries. You need an environment with Python version 3.6 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 to 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]
```

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

```
pip install stable-baselines3
```

1.1.2 Bleeding-edge version

```
pip install 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=/root/code/stable-baselines3,type=bind stablebaselines/stable-
↪baselines3 bash -c 'cd /root/code/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=/root/code/stable-baselines3,type=bind stablebaselines/stable-baselines3-cpu␣
↪bash -c 'cd /root/code/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 allow 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 `/root/code/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

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 gym

from stable_baselines3 import A2C

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

model = A2C('MlpPolicy', env, verbose=1)
model.learn(total_timesteps=10000)

obs = env.reset()
for i in range(1000):
    action, _state = model.predict(obs, deterministic=True)
```

(continues on next page)

(continued from previous page)

```
    obs, reward, done, info = env.step(action)
    env.render()
    if done:
      obs = env.reset()
```

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

```
from stable_baselines3 import A2C

model = A2C('MlpPolicy', 'CartPole-v1').learn(10000)
```

1.3 Reinforcement Learning Tips and Tricks

The aim of this section is to help you doing 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.

1.3.1 General advice when using Reinforcement Learning

TL;DR

1. Read about RL and Stable Baselines3
2. Do quantitative experiments and hyperparameter tuning if needed
3. Evaluate the performance using a separate test environment
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 you read Stable Baselines3 (SB3) documentation and do the tutorial. It covers basic usage and guide 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 (compared to supervised learning where you have a fixed dataset for instance). This dependence can lead to vicious circle: if the agent collects poor quality data (e.g., trajectories with no rewards), then it will not improve and continue to amass 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 have quantitative results.

Good results in RL are generally dependent on finding appropriate hyperparameters. Recent algorithms (PPO, SAC, TD3) normally require little hyperparameter tuning, however, *don't expect the default ones to work* on any 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 SB) are usually *sample inefficient*. They require a lot of samples (sometimes millions of interactions) to learn something 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.

One last limitation of RL is the instability of training. That is to say, you can observe during training a huge drop in performance. This behavior is particularly present in `DDPG`, that's why its extension `TD3` tries to tackle that issue. Other method, like `TRPO` or `PPO` make use of a *trust region* to minimize that problem by avoiding too large update.

How to evaluate an RL algorithm?

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.

As some policy 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.

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

We suggest you reading Deep Reinforcement Learning that Matters for a good discussion about RL evaluation.

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, depending on your needs and problem, you may choose one or the other. 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` only supports discrete actions, where `SAC` is restricted to continuous actions.

The second difference that will help you choose 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 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. 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` and `TD3`. Please use the hyperparameters in the RL zoo for best results.

Continuous Actions - Multiprocessed

Take a look at PPO, TRPO 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

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

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

```
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:

```
env = YourEnv()
obs = env.reset()
n_steps = 10
for _ in range(n_steps):
    # Random action
    action = env.action_space.sample()
    obs, reward, done, info = env.step(action)
                if done:
                        obs = 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 be difficult to debug (cf attached image and issue #473).

```
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")
```

Another consequence of using a Gaussian 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

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*:

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.

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
- David Silver's course
- Lilian Weng's blog
- Berkeley's Deep RL Bootcamp
- Berkeley's Deep Reinforcement Learning course
- 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
A2C	✓	✓	✓	✓	✓
DDPG	✓				
DQN		✓			
PPO	✓	✓	✓	✓	✓
SAC	✓				
TD3	✓				

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

Actions `gym.spaces`:

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

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.

1.5.1 Reproducibility

Completely reproducible results are not guaranteed across Tensorflow 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

1.6 Examples

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

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`

```
import gym

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


# Create environment
env = gym.make('LunarLander-v2')

# Instantiate the agent
model = DQN('MlpPolicy', env, verbose=1)
# Train the agent
model.learn(total_timesteps=int(2e5))
# Save the agent
model.save("dqn_lunar")
del model  # delete trained model to demonstrate loading

# Load the trained agent
model = DQN.load("dqn_lunar")

# Evaluate the agent
mean_reward, std_reward = evaluate_policy(model, model.get_env(), n_eval_episodes=10)

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

1.6.3 Multiprocessing: Unleashing the Power of Vectorized Environments

Fig. 2: CartPole Environment

```
import gym
import numpy as np

from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import SubprocVecEnv
from stable_baselines3.common.cmd_util import make_vec_env
from stable_baselines3.common.utils import set_random_seed

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

    :param env_id: (str) the environment ID
    :param num_env: (int) the number of environments you wish to have in subprocesses
    :param seed: (int) the inital seed for RNG
    :param rank: (int) index of the subprocess
    """
    def _init():
        env = gym.make(env_id)
        env.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
    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:
    # env = make_vec_env(env_id, n_envs=num_cpu, seed=0)

    model = PPO('MlpPolicy', env, verbose=1)
    model.learn(total_timesteps=25000)

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

1.6.4 Using Callback: 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 in Visdom) or save the best agent. If your callback returns False, training is aborted early.

```
import os

import 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
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: (int)
    :param log_dir: (str) Path to the folder where the model will be saved.
      It must contains the file created by the ``Monitor`` wrapper.
    :param verbose: (int)
    """
    def __init__(self, check_freq: int, log_dir: str, verbose=1):
        super(SaveOnBestTrainingRewardCallback, self).__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 > 0:
                print("Num timesteps: {}".format(self.num_timesteps))
```

(continues on next page)

(continued from previous page)

```
                print("Best mean reward: {:.2f} - Last mean reward per episode: {:.2f}
↪".format(self.best_mean_reward, mean_reward))

            # 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 > 0:
                  print("Saving new best model to {}".format(self.save_path))
                self.model.save(self.save_path)

        return True

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

# 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()
```

1.6.5 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.

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

# There already exists an environment generator
```

(continues on next page)

(continued from previous page)

```
# that will make and wrap atari environments correctly.
# Here we are also multi-worker training (n_envs=4 => 4 environments)
env = make_atari_env('PongNoFrameskip-v4', n_envs=4, seed=0)
# Frame-stacking with 4 frames
env = VecFrameStack(env, n_stack=4)

model = A2C('CnnPolicy', env, verbose=1)
model.learn(total_timesteps=25000)

obs = env.reset()
while True:
    action, _states = model.predict(obs)
    obs, rewards, dones, info = env.step(action)
    env.render()
```

1.6.6 PyBullet: Normalizing input features

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

Note: you need to install pybullet with `pip install pybullet`

```
import gym
import pybullet_envs

from stable_baselines3.common.vec_env import DummyVecEnv, VecNormalize
from stable_baselines3 import PPO

env = DummyVecEnv([lambda: gym.make("HalfCheetahBulletEnv-v0")])
# Automatically normalize the input features and reward
env = VecNormalize(env, norm_obs=True, norm_reward=True,
                   clip_obs=10.)

model = PPO('MlpPolicy', env)
model.learn(total_timesteps=2000)

# Don't forget to save the VecNormalize statistics when saving the agent
log_dir = "/tmp/"
model.save(log_dir + "ppo_halfcheetah")
stats_path = os.path.join(log_dir, "vec_normalize.pkl")
env.save(stats_path)

# To demonstrate loading
del model, env

# Load the agent
model = PPO.load(log_dir + "ppo_halfcheetah")

# Load the saved statistics
env = DummyVecEnv([lambda: gym.make("HalfCheetahBulletEnv-v0")])
```

(continues on next page)

(continued from previous page)

```
env = VecNormalize.load(stats_path, env)
#  do not update them at test time
env.training = False
# reward normalization is not needed at test time
env.norm_reward = False
```

1.6.7 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 gym
from stable_baselines3.common.vec_env import VecVideoRecorder, DummyVecEnv

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

env = DummyVecEnv([lambda: gym.make(env_id)])

obs = env.reset()

# Record the video starting at the first step
env = VecVideoRecorder(env, video_folder,
                       record_video_trigger=lambda x: x == 0, video_length=video_
↪length,
                       name_prefix="random-agent-{}".format(env_id))

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

1.6.8 Bonus: Make a GIF of a Trained Agent

Note: For Atari games, you need to use a screen recorder such as Kazam. And then convert the video using ffmpeg

```
import imageio
import numpy as np

from stable_baselines3 import A2C

model = A2C("MlpPolicy", "LunarLander-v2").learn(100000)

images = []
obs = model.env.reset()
```

(continues on next page)

(continued from previous page)

```
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 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

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

An abstract asynchronous, vectorized environment.

Parameters

- **num_envs** – (int) the number of environments
- **observation_space** – (Gym Space) the observation space
- **action_space** – (Gym Space) the action space

abstract close()

Clean up the environment's resources.

abstract 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** – (list,int) Indices of envs whose method to call
- **method_args** – (tuple) Any positional arguments to provide in the call
- **method_kwargs** – (dict) Any keyword arguments to provide in the call

Returns (list) List of items returned by the environment's method call

abstract 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** – (list,int) Indices of envs to get attribute from

Returns (list) List of values of 'attr_name' in all environments

get_images() → Sequence[numpy.ndarray]

Return RGB images from each environment

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 (str or None) name of module whose attribute is being shadowed, if any.

render(*mode: str = 'human'*)

Gym environment rendering

Parameters **mode** – the rendering type

abstract 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 ([int] or [float]) observation

abstract seed(*seed: Optional[int] = None*) → List[Union[None, int]]
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.

Parameters **seed** – (Optional[int]) The random seed. May be None for completely random seeding.

Returns (List[Union[None, int]]) 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.

abstract 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** – (obj) Value to assign to *attr_name*
- **indices** – (list,int) Indices of envs to assign value

Returns (NoneType)

step(*actions*)
Step the environments with the given action

Parameters **actions** – ([int] or [float]) the action

Returns ([int] or [float], [float], [bool], dict) observation, reward, done, information

abstract 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.

abstract step_wait()
Wait for the step taken with step_async().

Returns ([int] or [float], [float], [bool], dict) observation, reward, done, information

1.7.2 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** – ([Gym Environment]) the list of environments to vectorize

close()
Clean up the environment's resources.

env_method(*method_name*, **method_args*, *indices=None*, ***method_kwargs*)
Call instance methods of vectorized environments.

get_attr(*attr_name*, *indices=None*)
Return attribute from vectorized environment (see base class).

get_images() → Sequence[numpy.ndarray]
Return RGB images from each environment

render(*mode: str = 'human'*)
Gym environment rendering. If there are multiple environments then they are tiled together in one image via `BaseVecEnv.render()`. Otherwise (if `self.num_envs == 1`), we pass the render call directly to the underlying environment.

Therefore, some arguments such as `mode` will have values that are valid only when `num_envs == 1`.

Parameters **mode** – The rendering type.

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 ([int] or [float]) observation

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.

Parameters **seed** – (Optional[int]) The random seed. May be None for completely random seeding.

Returns (List[Union[None, int]]) 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.

set_attr(*attr_name*, *value*, *indices=None*)
Set attribute inside vectorized environments (see base class).

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.

step_wait()
Wait for the step taken with step_async().

Returns ([int] or [float], [float], [bool], dict) observation, reward, done, information

1.7.3 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** – ([Gym Environment]) Environments to run in subprocesses
- **start_method** – (str) 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.

env_method(*method_name*, **method_args*, *indices=None*, ***method_kwargs*)
Call instance methods of vectorized environments.

get_attr(*attr_name*, *indices=None*)
Return attribute from vectorized environment (see base class).

get_images() → Sequence[numpy.ndarray]
Return RGB images from each environment

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 ([int] or [float]) observation

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.

Parameters **seed** – (Optional[int]) The random seed. May be None for completely random seeding.

Returns (List[Union[None, int]]) 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.

set_attr(*attr_name*, *value*, *indices=None*)
Set attribute inside vectorized environments (see base class).

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.

step_wait()
Wait for the step taken with step_async().

Returns ([int] or [float], [float], [bool], dict) observation, reward, done, information

1.7.4 Wrappers

VecFrameStack

class stable_baselines3.common.vec_env.**VecFrameStack**(*venv: stable_baselines3.common.vec_env.base_vec_env.VecEnv*, *n_stack: int*)

Frame stacking wrapper for vectorized environment

Parameters

- **venv** – the vectorized environment to wrap
- **n_stack** – Number of frames to stack

close()
Clean up the environment's resources.

reset()
Reset all environments

step_wait()
Wait for the step taken with step_async().

Returns ([int] or [float], [float], [bool], dict) observation, reward, done, information

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*)

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

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

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

static load(*load_path: str*, *venv: stable_baselines3.common.vec_env.base_vec_env.VecEnv*) → stable_baselines3.common.vec_env.vec_normalize.VecNormalize
Loads a saved VecNormalize object.

Parameters

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

Returns (VecNormalize)

normalize_obs(*obs*)
Normalize observations using this VecNormalize's observations statistics. Calling this method does not update statistics.

normalize_reward(*reward*)
Normalize rewards using this VecNormalize's rewards statistics. Calling this method does not update statistics.

reset()
Reset all environments

save(*save_path: str*) → None
Save current VecNormalize object with all running statistics and settings (e.g. clip_obs)

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

set_venv(*venv*)
Sets the vector environment to wrap to venv.

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

Parameters **venv** – (VecEnv)

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

where 'news' is a boolean vector indicating whether each element is new.

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.

Parameters

- **venv** – (VecEnv or VecEnvWrapper)
- **video_folder** – (str) Where to save videos
- **record_video_trigger** – (func) 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()
Clean up the environment's resources.

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 ([int] or [float]) observation

step_wait()
Wait for the step taken with step_async().

Returns ([int] or [float], [float], [bool], dict) observation, reward, done, information

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 or not to raise a ValueError, instead of a UserWarning
- **warn_once** – (bool) Whether or not to only warn once.
- **check_inf** – (bool) Whether or not to check for +inf or -inf as well

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 ([int] or [float]) observation

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.

step_wait()
Wait for the step taken with step_async().

Returns ([int] or [float], [float], [bool], dict) observation, reward, done, information

VecTransposeImage

class stable_baselines3.common.vec_env.**VecTransposeImage**(*venv: stable_baselines3.common.vec_env.base_vec_env.Vec*

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

Parameters **venv** – (VecEnv)

close() → None
Clean up the environment's resources.

reset() → numpy.ndarray
Reset all environments

step_wait() → GymStepReturn
Wait for the step taken with step_async().

Returns ([int] or [float], [float], [bool], dict) observation, reward, done, information

static transpose_image(*image: numpy.ndarray*) → numpy.ndarray
Transpose an image or batch of images (re-order channels).

Parameters **image** – (np.ndarray)

Returns (np.ndarray)

static transpose_space(*observation_space: gym.spaces.box.Box*) → gym.spaces.box.Box
Transpose an observation space (re-order channels).

Parameters **observation_space** – (spaces.Box)

Returns (spaces.Box)

1.8 Using Custom Environments

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

Note: If you are using images as input, the input values must be in [0, 255] as the observation is normalized (dividing by 255 to have values in [0, 1]) when using CNN policies.

```
import gym
from gym import spaces

class CustomEnv(gym.Env):
  """Custom Environment that follows gym interface"""
  metadata = {'render.modes': ['human']}

  def __init__(self, arg1, arg2, ...):
    super(CustomEnv, self).__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:
    self.observation_space = spaces.Box(low=0, high=255,
                                        shape=(HEIGHT, WIDTH, N_CHANNELS), dtype=np.
↪uint8)
```

(continues on next page)

(continued from previous page)

```
  def step(self, action):
    ...
    return observation, reward, done, info
  def reset(self):
    ...
    return observation  # reward, done, info can't be included
  def render(self, mode='human'):
    ...
  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, 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)
```

We have created a colab notebook for a concrete example of creating a custom environment.

You can also find a complete guide online on creating a custom Gym environment.

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).

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.

1.9 Custom Policy Network

Stable Baselines3 provides policy networks for images (CnnPolicies) and other type of input features (MlpPolicies).

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

```
import gym
import torch as th

from stable_baselines3 import PPO

# Custom MLP policy of two layers of size 32 each with Relu activation function
policy_kwargs = dict(activation_fn=th.nn.ReLU, net_arch=[32, 32])
# Create the agent
model = PPO("MlpPolicy", "CartPole-v1", policy_kwargs=policy_kwargs, verbose=1)
# Retrieve the environment
env = model.get_env()
```

(continues on next page)

(continued from previous page)

```
# Train the agent
model.learn(total_timesteps=100000)
# Save the agent
model.save("ppo-cartpole")

del model
# the policy_kwargs are automatically loaded
model = PPO.load("ppo-cartpole")
```

You can also easily define a custom architecture for the policy (or value) network:

Note: Defining a custom policy class is equivalent to passing `policy_kwargs`. However, it lets you name the policy and so usually makes the code clearer. `policy_kwargs` is particularly useful when doing hyperparameter search.

The `net_arch` parameter of `A2C` and `PPO` policies allows to specify the amount and size of the hidden layers and how many of them are shared between the policy network and the value network. It is assumed to be a list with the following structure:

1. An arbitrary length (zero allowed) number of integers each specifying the number of units in a shared layer. If the number of ints is zero, there will be no shared layers.
2. An optional dict, to specify the following non-shared layers for the value network and the policy network. It is formatted like `dict(vf=[<value layer sizes>], pi=[<policy layer sizes>])`. If it is missing any of the keys (pi or vf), no non-shared layers (empty list) is assumed.

In short: `[<shared layers>, dict(vf=[<non-shared value network layers>], pi=[<non-shared policy network layers>])]`.

1.9.1 Examples

Two shared layers of size 128: `net_arch=[128, 128]`

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

Value network deeper than policy network, first layer shared: `net_arch=[128, dict(vf=[256, 256])]`

```
          obs
           |
         <128>
   /               \
action             <256>
                     |
                   <256>
                     |
                   value
```

Initially shared then diverging: `[128, dict(vf=[256], pi=[16])]`

```
          obs
           |
         <128>
   /               \
 <16>              <256>
   |                 |
action            value
```

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

TODO

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).

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


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

    :param verbose: (int) Verbosity level 0: not output 1: info 2: debug
    """
    def __init__(self, verbose=0):
        super(CustomCallback, self).__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 = None  # type: Union[gym.Env, VecEnv, None]
        # Number of time the callback was called
        # self.n_calls = 0  # type: int
        # self.num_timesteps = 0  # type: int
        # local and global variables
        # self.locals = None  # type: Dict[str, Any]
        # self.globals = None  # type: Dict[str, Any]
        # The logger object, used to report things in the terminal
        # self.logger = None  # stable_baselines3.common.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:
        """
```

(continues on next page)

(continued from previous page)

```
        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: (bool) 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 to take 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: (Optional[BaseCallback]) Callback that will be called
        when an event is triggered.
    :param verbose: (int)
    """
    def __init__(self, callback: Optional[BaseCallback] = None, verbose: int = 0):
        super(EventCallback, self).__init__(verbose=verbose)
        self.callback = callback
        # Give access to the parent
        if callback is not None:
            self.callback.parent = self
    ...

    def _on_event(self) -> bool:
        if self.callback is not None:
            return self.callback()
        return True
```

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*)
- stopping the training early based on a reward threshold (*StopTrainingOnRewardThreshold*)

CheckpointCallback

Callback for saving a model every `save_freq` steps, you must specify a log folder (`save_path`) and optionally a prefix for the checkpoints (`rl_model` by default).

```
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')

model = SAC('MlpPolicy', 'Pendulum-v0')
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 a child callback via the `callback_on_new_best` argument. It will be triggered each time there is a new best model.

```
import gym

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

# Separate evaluation env
eval_env = gym.make('Pendulum-v0')
# 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-v0')
model.learn(5000, callback=eval_callback)
```

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 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-v0')
```

(continues on next page)

(continued from previous page)

```
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])

model = SAC('MlpPolicy', 'Pendulum-v0')
# 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 gym

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

# Separate evaluation env
eval_env = gym.make('Pendulum-v0')
# 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-v0', 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 `PPO1` and `TRPO` work (they rely on MPI), `n_steps` is a lower bound between two events.

```
import 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-v0', verbose=1)
```

(continues on next page)

(continued from previous page)

```
model.learn(int(2e4), callback=event_callback)
```

class stable_baselines3.common.callbacks.**BaseCallback**(*verbose: int = 0*)
Base class for callback.

Parameters **verbose** – (int)

init_callback(*model: BaseAlgorithm*) → None
Initialize the callback by saving references to the RL model and the training environment for convenience.

on_step() → 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.

Returns (bool) If the callback returns False, training is aborted early.

class stable_baselines3.common.callbacks.**CallbackList**(*callbacks: List[*stable_baselines3.common.callbacks.BaseCallbac
Class for chaining callbacks.

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

class stable_baselines3.common.callbacks.**CheckpointCallback**(*save_freq: int*, *save_path: str*, *name_prefix='rl_model'*, *verbose=0*)
Callback for saving a model every `save_freq` steps

Parameters

- **save_freq** – (int)
- **save_path** – (str) Path to the folder where the model will be saved.
- **name_prefix** – (str) Common prefix to the saved models

class stable_baselines3.common.callbacks.**ConvertCallback**(*callback*, *verbose=0*)
Convert functional callback (old-style) to object.

Parameters

- **callback** – (callable)
- **verbose** – (int)

class stable_baselines3.common.callbacks.**EvalCallback**(*eval_env: Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv*, *callback_on_new_best: Optional[*stable_baselines3.common.callbacks.BaseCallb *= None*, *n_eval_episodes: int = 5*, *eval_freq: int = 10000*, *log_path: str = None*, *best_model_save_path: str = None*, *deterministic: bool = True*, *render: bool = False*, *verbose: int = 1*)
Callback for evaluating an agent.

Parameters

- **eval_env** – (Union[gym.Env, VecEnv]) The environment used for initialization
- **callback_on_new_best** – (Optional[BaseCallback]) Callback to trigger when there is a new best model according to the `mean_reward`
- **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) Path to a folder where the evaluations (`evaluations.npz`) will be saved. It will be updated at each evaluation.
- **best_model_save_path** – (str) 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.
- **deterministic** – (bool) Whether to render or not the environment during evaluation
- **render** – (bool) Whether to render or not the environment during evaluation
- **verbose** – (int)

class stable_baselines3.common.callbacks.**EventCallback**(*callback: Optional[*stable_baselines3.common.callbacks.BaseCall = *None*, *verbose: int = 0*)

Base class for triggering callback on event.

Parameters

- **callback** – (Optional[BaseCallback]) Callback that will be called when an event is triggered.
- **verbose** – (int)

init_callback(*model: BaseAlgorithm*) → None

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

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

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.**StopTrainingOnRewardThreshold**(*reward_threshold: float*, *verbose: int = 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)

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:

```
from stable_baselines3 import A2C

model = A2C('MlpPolicy', 'CartPole-v1', verbose=1, tensorboard_log="./a2c_cartpole_
↪tensorboard/")
model.learn(total_timesteps=10000)
```

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

```
from stable_baselines3 import A2C

model = A2C('MlpPolicy', 'CartPole-v1', verbose=1, tensorboard_log="./a2c_cartpole_
↪tensorboard/")
model.learn(total_timesteps=10000, 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
model.learn(total_timesteps=10000, tb_log_name="second_run", reset_num_
↪timesteps=False)
model.learn(total_timesteps=10000, tb_log_name="third_run", reset_num_timesteps=False)
```

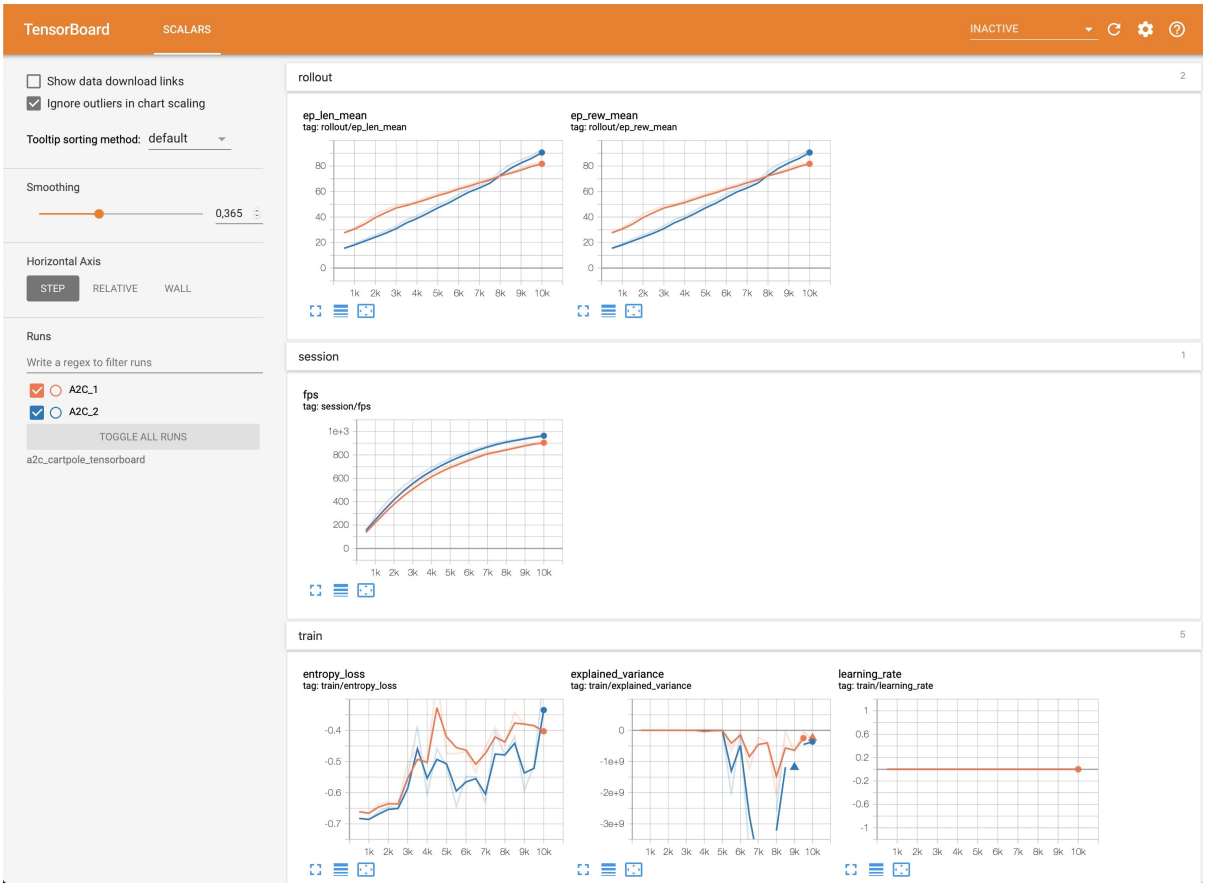
Once the learn function is called, you can monitor the RL agent during or after the training, with the following bash command:

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

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:

```
import numpy as np

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

model = SAC("MlpPolicy", "Pendulum-v0", tensorboard_log="/tmp/sac/", verbose=1)


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

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

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

(continues on next page)

(continued from previous page)

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

1.12 RL Baselines3 Zoo

RL Baselines3 Zoo. is a collection of pre-trained Reinforcement Learning agents using Stable-Baselines3. It also provides basic scripts for training, evaluating agents, tuning hyperparameters and recording videos.

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!

1.12.1 Installation

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

2. Install dependencies

```
apt-get install swig cmake ffmpeg
pip install -r requirements.txt
```

1.12.2 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 train.py --algo algo_name --env env_id
```

For example (with evaluation and checkpoints):

```
python train.py --algo ppo2 --env CartPole-v1 --eval-freq 10000 --save-freq 50000
```

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

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

1.12.3 Enjoy a Trained Agent

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

```
python enjoy.py --algo algo_name --env env_id
```

For example, enjoy A2C on Breakout during 5000 timesteps:

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

1.12.4 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 train.py --algo ppo --env MountainCar-v0 -n 50000 -optimize --n-trials 1000 --
↪n-jobs 2 \
  --sampler random --pruner median
```

1.12.5 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.

1.13 Migrating from Stable-Baselines

This is a guide to migrate from Stable-Baselines to Stable-Baselines3.

It also references the main changes.

TODO

1.14 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.

1.14.1 How and why?

The issue arises then 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 ($\sqrt{-1}$, $\inf \times 1$, $\text{NaN} \bmod 1$, ...) return NaN
- **Division by zero:**
  - if the operand is not zero ($1/0$, $-2/0$, ...) 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.

1.14.2 Anomaly detection with PyTorch

To enable NaN detection in PyTorch you can do

```
import torch as th
th.autograd.set_detect_anomaly(True)
```

1.14.3 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.

```
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:

```
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)
```

1.14.4 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 gym
from gym 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:
```

(continues on next page)

(continued from previous page)

```
            obs = randf
        return [obs], 0.0, False, {}

    def reset(self):
        return [0.0]

    def render(self, mode='human', close=False):
        pass

# Create environment
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.
```

1.14.5 RL Model hyperparameters

Depending on your hyperparameters, NaN can occurs 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 hyperparameters 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).

1.14.6 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

1.15 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.

1.15.1 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

1.15.2 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).

1.15.3 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, we make use of an additional wrapper `VecTransposeImage` because PyTorch uses the "channel-first" convention.

1.15.4 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.

1.15.5 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")

1.15.6 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

1.15.7 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 ;) Abstract base classes for RL algorithms.

1.16 Base RL Class

Common interface for all the RL algorithms

class stable_baselines3.common.base_class.**BaseAlgorithm**(*policy: Type[stable_baselines3.common.policies.BasePolic env: Optional[Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecE str]], policy_base: Type[stable_baselines3.common.policies.BasePolic learning_rate: Union[float, Callable], policy_kwargs: Dict[str, Any] = None, tensorboard_log: Optional[str] = None, verbose: int = 0, device: Union[torch.device, str] = 'auto', support_multi_env: bool = False, create_eval_env: bool = False, monitor_wrapper: bool = True, seed: Optional[int] = None, use_sde: bool = False, sde_sample_freq: int = - 1*)

The base of RL algorithms

Parameters

- **policy** – (Type[BasePolicy]) Policy object
- **env** – (Union[GymEnv, str, None]) The environment to learn from (if registered in Gym, can be str. Can be None for loading trained models)
- **policy_base** – (Type[BasePolicy]) The base policy used by this method
- **learning_rate** – (float or callable) learning rate for the optimizer, it can be a function of the current progress remaining (from 1 to 0)
- **policy_kwargs** – (Dict[str, Any]) Additional arguments to be passed to the policy on creation
- **tensorboard_log** – (str) the log location for tensorboard (if None, no logging)
- **verbose** – (int) The verbosity level: 0 none, 1 training information, 2 debug
- **device** – (Union[th.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)
- **create_eval_env** – (bool) Whether to create a second environment that will be used for evaluating the agent periodically. (Only available when passing string for the environment)
- **monitor_wrapper** – (bool) When creating an environment, whether to wrap it or not in a Monitor wrapper.
- **seed** – (Optional[int]) 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)

excluded_save_params() → List[str]
Returns the names of the parameters that should be excluded by default when saving the model.

Returns ([str]) List of parameters that should be excluded from save

get_env() → Optional[stable_baselines3.common.vec_env.base_vec_env.VecEnv]
Returns the current environment (can be None if not defined).

Returns (Optional[VecEnv]) The current environment

get_torch_variables() → Tuple[List[str], List[str]]
Get the name of the torch variables that will be saved. `th.save` and `th.load` will be used with the right device instead of the default pickling strategy.

Returns (Tuple[List[str], List[str]]) name of the variables with state dicts to save, name of additional torch tensors,

get_vec_normalize_env() → Optional[stable_baselines3.common.vec_env.vec_normalize.VecNormalize]
Return the `VecNormalize` wrapper of the training env if it exists. :return: Optional[VecNormalize] The `VecNormalize` env.

abstract learn(*total_timesteps: int*, *callback: Union[None, Callable, List[stable_baselines3.common.callbacks.BaseCallback], stable_baselines3.common.callbacks.BaseCallback] = None*, *log_interval: int = 100*, *tb_log_name: str = 'run'*, *eval_env: Optional[Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv]] = None*, *eval_freq: int = - 1*, *n_eval_episodes: int = 5*, *eval_log_path: Optional[str] = None*, *reset_num_timesteps: bool = True*) → *stable_baselines3.common.base_class.BaseAlgorithm*

Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples (env steps) to train on
- **callback** – (MaybeCallback) callback(s) called at every step with state of the algorithm.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for TensorBoard logging
- **eval_env** – (gym.Env) Environment that will be used to evaluate the agent
- **eval_freq** – (int) Evaluate the agent every `eval_freq` timesteps (this may vary a little)
- **n_eval_episodes** – (int) Number of episode to evaluate the agent
- **eval_log_path** – (Optional[str]) Path to a folder where the evaluations will be saved
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseAlgorithm) the trained model

classmethod load(*load_path: str*, *env: Optional[Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv]] = None*, ***kwargs*) → *stable_baselines3.common.base_class.BaseAlgorithm*

Load the model from a zip-file

Parameters

- **load_path** – the location of the saved data
- **env** – 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
- **kwargs** – extra arguments to change the model when loading

predict(*observation: numpy.ndarray*, *state: Optional[numpy.ndarray] = None*, *mask: Optional[numpy.ndarray] = None*, *deterministic: bool = False*) → Tuple[numpy.ndarray, Optional[numpy.ndarray]]

Get the model's action(s) from an observation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (Optional[np.ndarray]) The last states (can be None, used in recurrent policies)
- **mask** – (Optional[np.ndarray]) The last masks (can be None, used in recurrent policies)
- **deterministic** – (bool) Whether or not to return deterministic actions.

Returns (Tuple[np.ndarray, Optional[np.ndarray]]) the model's action and the next state (used in recurrent policies)

save(*path: Union[str, pathlib.Path, io.BufferedIOBase]*, *exclude: Optional[Iterable[str]] = None*, *include: Optional[Iterable[str]] = None*) → None
Save all the attributes of the object and the model parameters in a zip-file.

Parameters

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

set_env(*env: Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv]*) → None
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** – The environment for learning a policy

set_random_seed(*seed: Optional[int] = None*) → None
Set the seed of the pseudo-random generators (python, numpy, pytorch, gym, action_space)

Parameters **seed** – (int)

1.16.1 Base Off-Policy Class

The base RL algorithm for Off-Policy algorithm (ex: SAC/TD3)

class stable_baselines3.common.off_policy_algorithm.**OffPolicyAlgorithm**(*policy: Type[stable_baselines3.comm*
env: Union[gym.core.Env, stable_baselines3.common.vec_ str],
policy_base: Type[stable_baselines3.comm
learning_rate: Union[float, Callable],
buffer_size: int = 1000000,
learning_starts: int = 100,
batch_size: int = 256,
tau: float = 0.005,
gamma: float = 0.99,
train_freq: int = 1,
gradient_steps: int = 1,
n_episodes_rollout: int = -1,
action_noise: Optional[stable_baselines3.com = None,
optimize_memory_usage: bool = False,
pol-

The base for Off-Policy algorithms (ex: SAC/TD3)

Parameters

- **policy** – Policy object
- **env** – The environment to learn from (if registered in Gym, can be str. Can be None for loading trained models)
- **policy_base** – The base policy used by this method
- **learning_rate** – (float or callable) 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) Update the model every `train_freq` steps. Set to *-1* to disable.
- **gradient_steps** – (int) How many gradient steps to do after each rollout (see `train_freq` and `n_episodes_rollout`) Set to `-1` means to do as many gradient steps as steps done in the environment during the rollout.
- **n_episodes_rollout** – (int) Update the model every `n_episodes_rollout` episodes. Note that this cannot be used at the same time as `train_freq`. Set to *-1* to disable.
- **action_noise** – (ActionNoise) the action noise type (None by default), this can help for hard exploration problem. Cf common.noise for the different action noise type.
- **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** – Additional arguments to be passed to the policy on creation
- **tensorboard_log** – (str) the log location for tensorboard (if None, no logging)
- **verbose** – The verbosity level: 0 none, 1 training information, 2 debug
- **device** – 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** – Whether the algorithm supports training with multiple environments (as in A2C)
- **create_eval_env** – Whether to create a second environment that will be used for evaluating the agent periodically. (Only available when passing string for the environment)
- **monitor_wrapper** – When creating an environment, whether to wrap it or not in a Monitor wrapper.
- **seed** – Seed for the pseudo random generators
- **use_sde** – Whether to use State Dependent Exploration (SDE) instead of action noise exploration (default: False)

- **sde_sample_freq** – 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

collect_rollouts(*env: stable_baselines3.common.vec_env.base_vec_env.VecEnv*, *callback:* stable_baselines3.common.callbacks.BaseCallback, *n_episodes: int = 1*, *n_steps: int = - 1*, *action_noise: Optional[*stable_baselines3.common.noise.ActionNoise*] = None*, *learning_starts: int = 0*, *replay_buffer: Optional[stable_baselines3.common.buffers.ReplayBuffer] = None*, *log_interval: Optional[int] = None*) → stable_baselines3.common.type_aliases.RolloutReturn

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)
- **n_episodes** – (int) Number of episodes to use to collect rollout data You can also specify a `n_steps` instead
- **n_steps** – (int) Number of steps to use to collect rollout data You can also specify a `n_episodes` instead.
- **action_noise** – (Optional[ActionNoise]) Action noise that will be used for exploration Required for deterministic policy (e.g. TD3). This can also be used in addition to the stochastic policy for SAC.
- **learning_starts** – (int) Number of steps before learning for the warm-up phase.
- **replay_buffer** – (ReplayBuffer)
- **log_interval** – (int) Log data every `log_interval` episodes

Returns (RolloutReturn)

learn(*total_timesteps: int*, *callback: Union[None, Callable, List[stable_baselines3.common.callbacks.BaseCallback], stable_baselines3.common.callbacks.BaseCallback] = None*, *log_interval: int = 4*, *eval_env: Optional[Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv]] = None*, *eval_freq: int = - 1*, *n_eval_episodes: int = 5*, *tb_log_name: str = 'run'*, *eval_log_path: Optional[str] = None*, *reset_num_timesteps: bool = True*) → *stable_baselines3.common.off_policy_algorithm.OffPolicyAlgorithm*

Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples (env steps) to train on
- **callback** – (MaybeCallback) callback(s) called at every step with state of the algorithm.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for TensorBoard logging
- **eval_env** – (gym.Env) Environment that will be used to evaluate the agent
- **eval_freq** – (int) Evaluate the agent every `eval_freq` timesteps (this may vary a little)

- **n_eval_episodes** – (int) Number of episode to evaluate the agent
- **eval_log_path** – (Optional[str]) Path to a folder where the evaluations will be saved
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseAlgorithm) the trained model

load_replay_buffer(*path: Union[str, pathlib.Path, io.BufferedIOBase]*) → None
Load a replay buffer from a pickle file.

Parameters **path** – (Union[str, pathlib.Path, io.BufferedIOBase]) Path to the pickled replay buffer.

save_replay_buffer(*path: Union[str, pathlib.Path, io.BufferedIOBase]*) → None
Save the replay buffer as a pickle file.

Parameters **path** – (Union[str,pathlib.Path, io.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.

train(*gradient_steps: int*, *batch_size: int*) → None
Sample the replay buffer and do the updates (gradient descent and update target networks)

1.16.2 Base On-Policy Class

The base RL algorithm for On-Policy algorithm (ex: A2C/PPO)

class stable_baselines3.common.on_policy_algorithm.**OnPolicyAlgorithm**(*policy: Union[str, Type[stable_baselines3.common*, *env: Union[gym.core.Env, stable_baselines3.common.vec_env*, *str]*, *learning_rate: Union[float, Callable]*, *n_steps: int*, *gamma: float*, *gae_lambda: float*, *ent_coef: float*, *vf_coef: float*, *max_grad_norm: float*, *use_sde: bool*, *sde_sample_freq: int*, *tensorboard_log: Optional[str] = None*, *create_eval_env: bool = False*, *monitor_wrapper: bool = True*, *policy_kwargs: Optional[Dict[str, Any]] = None*, *verbose: int = 0*, *seed: Optional[int] = None*, *device: Union[torch.device, str] =*

The base for On-Policy algorithms (ex: A2C/PPO).

Parameters

- **policy** – (ActorCriticPolicy or str) The policy model to use (MlpPolicy, CnnPolicy, ...)
- **env** – (Gym environment or str) The environment to learn from (if registered in Gym, can be str)
- **learning_rate** – (float or callable) 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)
- **tensorboard_log** – (str) the log location for tensorboard (if None, no logging)
- **create_eval_env** – (bool) Whether to create a second environment that will be used for evaluating the agent periodically. (Only available when passing string for the environment)
- **monitor_wrapper** – When creating an environment, whether to wrap it or not in a Monitor wrapper.
- **policy_kwargs** – (dict) additional arguments to be passed to the policy on creation
- **verbose** – (int) the verbosity level: 0 no output, 1 info, 2 debug
- **seed** – (int) Seed for the pseudo random generators
- **device** – (str or th.device) 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: stable_baselines3.common.vec_env.base_vec_env.VecEnv*, *callback:* stable_baselines3.common.callbacks.BaseCallback, *rollout_buffer: stable_baselines3.common.buffers.RolloutBuffer*, *n_rollout_steps: int*) → bool

Collect rollouts using the current policy and fill a *RolloutBuffer*.

Parameters

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

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

get_torch_variables() → Tuple[List[str], List[str]]
cf base class

learn(*total_timesteps: int*, *callback: Union[None, Callable, List[stable_baselines3.common.callbacks.BaseCallback], stable_baselines3.common.callbacks.BaseCallback] = None*, *log_interval: int = 1*, *eval_env: Optional[Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv]] = None*, *eval_freq: int = - 1*, *n_eval_episodes: int = 5*, *tb_log_name: str = 'OnPolicyAlgorithm'*, *eval_log_path: Optional[str] = None*, *reset_num_timesteps: bool = True*) → *stable_baselines3.common.on_policy_algorithm.OnPolicyAlgorithm*
Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples (env steps) to train on
- **callback** – (MaybeCallback) callback(s) called at every step with state of the algorithm.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for TensorBoard logging
- **eval_env** – (gym.Env) Environment that will be used to evaluate the agent
- **eval_freq** – (int) Evaluate the agent every `eval_freq` timesteps (this may vary a little)
- **n_eval_episodes** – (int) Number of episode to evaluate the agent
- **eval_log_path** – (Optional[str]) Path to a folder where the evaluations will be saved
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseAlgorithm) the trained model

train() → None
Consume current rollout data and update policy parameters. Implemented by individual algorithms.

1.17 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))`. Read more here.

1.17.1 Notes

- Original paper: https://arxiv.org/abs/1602.01783
- OpenAI blog post: https://openai.com/blog/baselines-acktr-a2c/

1.17.2 Can I use?

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

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

1.17.3 Example

Train a A2C agent on `CartPole-v1` using 4 environments.

```
import gym

from stable_baselines3 import A2C
from stable_baselines3.a2c import MlpPolicy
from stable_baselines3.common.cmd_util import make_vec_env

# Parallel environments
env = make_vec_env('CartPole-v1', n_envs=4)

model = A2C(MlpPolicy, 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 = env.reset()
while True:
    action, _states = model.predict(obs)
    obs, rewards, dones, info = env.step(action)
    env.render()
```

1.17.4 Parameters

class stable_baselines3.a2c.**A2C**(*policy: Union[str, Type[stable_baselines3.common.policies.ActorCriticPolicy]], env: Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv, str], learning_rate: Union[float, Callable] = 0.0007, n_steps: int = 5, gamma: float = 0.99, gae_lambda: float = 1.0, ent_coef: float = 0.0, vf_coef: float = 0.5, max_grad_norm: float = 0.5, rms_prop_eps: float = 1e-05, use_rms_prop: bool = True, use_sde: bool = False, sde_sample_freq: int = - 1, normalize_advantage: bool = False, tensorboard_log: Optional[str] = None, create_eval_env: bool = False, policy_kwargs: Optional[Dict[str, Any]] = None, verbose: int = 0, seed: Optional[int] = None, device: Union[torch.device, str] = 'auto', _init_setup_model: bool = 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 or str) The policy model to use (MlpPolicy, CnnPolicy, ...)
- **env** – (Gym environment or str) The environment to learn from (if registered in Gym, can be str)
- **learning_rate** – (float or callable) The learning rate, it can be a function
- **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)
- **normalize_advantage** – (bool) Whether to normalize or not the advantage
- **tensorboard_log** – (str) the log location for tensorboard (if None, no logging)
- **create_eval_env** – (bool) Whether to create a second environment that will be used for evaluating the agent periodically. (Only available when passing string for the environment)
- **policy_kwargs** – (dict) additional arguments to be passed to the policy on creation

- **verbose** – (int) the verbosity level: 0 no output, 1 info, 2 debug
- **seed** – (int) Seed for the pseudo random generators
- **device** – (str or th.device) 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: stable_baselines3.common.vec_env.base_vec_env.VecEnv*, *callback:* stable_baselines3.common.callbacks.BaseCallback, *rollout_buffer: stable_baselines3.common.buffers.RolloutBuffer*, *n_rollout_steps: int*) → bool

Collect rollouts using the current policy and fill a *RolloutBuffer*.

Parameters

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

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

excluded_save_params() → List[str]

Returns the names of the parameters that should be excluded by default when saving the model.

Returns ([str]) List of parameters that should be excluded from save

get_env() → Optional[stable_baselines3.common.vec_env.base_vec_env.VecEnv]

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

Returns (Optional[VecEnv]) The current environment

get_torch_variables() → Tuple[List[str], List[str]]

cf base class

get_vec_normalize_env() → Optional[stable_baselines3.common.vec_env.vec_normalize.VecNormalize]

Return the `VecNormalize` wrapper of the training env if it exists. :return: Optional[VecNormalize] The `VecNormalize` env.

learn(*total_timesteps: int*, *callback: Union[None, Callable, List[stable_baselines3.common.callbacks.BaseCallback], stable_baselines3.common.callbacks.BaseCallback] = None*, *log_interval: int = 100*, *eval_env: Optional[Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv]] = None*, *eval_freq: int = - 1*, *n_eval_episodes: int = 5*, *tb_log_name: str = 'A2C'*, *eval_log_path: Optional[str] = None*, *reset_num_timesteps: bool = True*) → stable_baselines3.a2c.a2c.A2C

Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples (env steps) to train on
- **callback** – (MaybeCallback) callback(s) called at every step with state of the algorithm.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for TensorBoard logging
- **eval_env** – (gym.Env) Environment that will be used to evaluate the agent

- **eval_freq** – (int) Evaluate the agent every `eval_freq` timesteps (this may vary a little)
- **n_eval_episodes** – (int) Number of episode to evaluate the agent
- **eval_log_path** – (Optional[str]) Path to a folder where the evaluations will be saved
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseAlgorithm) the trained model

classmethod load(*load_path: str*, *env: Optional[Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv]] = None*, ***kwargs*) → *stable_baselines3.common.base_class.BaseAlgorithm*

Load the model from a zip-file

Parameters

- **load_path** – the location of the saved data
- **env** – 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
- **kwargs** – extra arguments to change the model when loading

predict(*observation: numpy.ndarray*, *state: Optional[numpy.ndarray] = None*, *mask: Optional[numpy.ndarray] = None*, *deterministic: bool = False*) → Tuple[numpy.ndarray, Optional[numpy.ndarray]]

Get the model's action(s) from an observation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (Optional[np.ndarray]) The last states (can be None, used in recurrent policies)
- **mask** – (Optional[np.ndarray]) The last masks (can be None, used in recurrent policies)
- **deterministic** – (bool) Whether or not to return deterministic actions.

Returns (Tuple[np.ndarray, Optional[np.ndarray]]) the model's action and the next state (used in recurrent policies)

save(*path: Union[str, pathlib.Path, io.BufferedIOBase]*, *exclude: Optional[Iterable[str]] = None*, *include: Optional[Iterable[str]] = None*) → None

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

Parameters

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

set_env(*env: Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv]*) → None

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** – The environment for learning a policy

set_random_seed(*seed: Optional[int] = None*) → None

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

Parameters **seed** – (int)

train() → None
Update policy using the currently gathered rollout buffer (one gradient step over whole data).

1.18 DDPG

Deep Deterministic Policy Gradient (DDPG) combines the trick for DQN with the deterministic policy gradient, to obtain an algorithm for continuous actions.

Available Policies

MlpPolicy	alias of stable_baselines3.td3.policies.TD3Policy

1.18.1 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

Note: The default policy for DDPG uses a ReLU activation, to match the original paper, whereas most other algorithms' MlpPolicy uses a tanh activation. to match the original paper

1.18.2 Can I use?

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

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

1.18.3 Example

```
import gym
import numpy as np

from stable_baselines3 import DDPG
from stable_baselines3.common.noise import NormalActionNoise,␣
↪OrnsteinUhlenbeckActionNoise

env = gym.make('Pendulum-v0')

# 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")
env = model.get_env()

del model # remove to demonstrate saving and loading

model = DDPG.load("ddpg_pendulum")

obs = env.reset()
while True:
    action, _states = model.predict(obs)
    obs, rewards, dones, info = env.step(action)
    env.render()
```

1.18.4 Parameters

class stable_baselines3.ddpg.**DDPG**(*policy: Union[str, Type[stable_baselines3.td3.policies.TD3Policy]], env: Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv, str], learning_rate: Union[float, Callable] = 0.001, buffer_size: int = 1000000, learning_starts: int = 100, batch_size: int = 100, tau: float = 0.005, gamma: float = 0.99, train_freq: int = - 1, gradient_steps: int = - 1, n_episodes_rollout: int = 1, action_noise: Optional[stable_baselines3.common.noise.ActionNoise] = None, optimize_memory_usage: bool = False, tensorboard_log: Optional[str] = None, create_eval_env: bool = False, policy_kwargs: Dict[str, Any] = None, verbose: int = 0, seed: Optional[int] = None, device: Union[torch.device, str] = 'auto', _init_setup_model: bool = 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** – (DDPGPolicy or str) The policy model to use (MlpPolicy, CnnPolicy, . . .)

- **env** – (GymEnv or str) The environment to learn from (if registered in Gym, can be str)
- **learning_rate** – (float or callable) learning rate for adam optimizer, the same learning rate will be used for all networks (Q-Values, Actor and Value function) it can be a function of the current progress remaining (from 1 to 0)
- **buffer_size** – (int) size of the replay buffer
- **learning_starts** – (int) how many steps of the model to collect transitions for before learning starts
- **batch_size** – (int) Minibatch size for each gradient update
- **tau** – (float) the soft update coefficient ("Polyak update", between 0 and 1)
- **gamma** – (float) the discount factor
- **train_freq** – (int) Update the model every `train_freq` steps. Set to *-1* to disable.
- **gradient_steps** – (int) How many gradient steps to do after each rollout (see `train_freq` and `n_episodes_rollout`) Set to `-1` means to do as many gradient steps as steps done in the environment during the rollout.
- **n_episodes_rollout** – (int) Update the model every `n_episodes_rollout` episodes. Note that this cannot be used at the same time as `train_freq`. Set to *-1* to disable.
- **action_noise** – (ActionNoise) the action noise type (None by default), this can help for hard exploration problem. Cf common.noise for the different action noise type.
- **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
- **create_eval_env** – (bool) Whether to create a second environment that will be used for evaluating the agent periodically. (Only available when passing string for the environment)
- **policy_kwargs** – (dict) additional arguments to be passed to the policy on creation
- **verbose** – (int) the verbosity level: 0 no output, 1 info, 2 debug
- **seed** – (int) Seed for the pseudo random generators
- **device** – (str or th.device) 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: stable_baselines3.common.vec_env.base_vec_env.VecEnv*, *callback:* stable_baselines3.common.callbacks.BaseCallback, *n_episodes: int = 1*, *n_steps: int = - 1*, *action_noise: Optional[*stable_baselines3.common.noise.ActionNoise*] = None*, *learning_starts: int = 0*, *replay_buffer: Optional[stable_baselines3.common.buffers.ReplayBuffer] = None*, *log_interval: Optional[int] = None*) → stable_baselines3.common.type_aliases.RolloutReturn

Collect rollout using the current policy (and possibly fill the replay buffer)

Parameters

- **env** – (VecEnv) The training environment
- **n_episodes** – (int) Number of episodes to use to collect rollout data You can also specify a `n_steps` instead

- **n_steps** – (int) Number of steps to use to collect rollout data You can also specify a `n_episodes` instead.
- **action_noise** – (Optional[ActionNoise]) Action noise that will be used for exploration Required for deterministic policy (e.g. TD3). This can also be used in addition to the stochastic policy for SAC.
- **callback** – (BaseCallback) Callback that will be called at each step (and at the beginning and end of the rollout)
- **learning_starts** – (int) Number of steps before learning for the warm-up phase.
- **replay_buffer** – (ReplayBuffer)
- **log_interval** – (int) Log data every `log_interval` episodes

Returns (RolloutReturn)

excluded_save_params() → List[str]
Returns the names of the parameters that should be excluded by default when saving the model.

Returns (List[str]) List of parameters that should be excluded from save

get_env() → Optional[stable_baselines3.common.vec_env.base_vec_env.VecEnv]
Returns the current environment (can be None if not defined).

Returns (Optional[VecEnv]) The current environment

get_torch_variables() → Tuple[List[str], List[str]]
cf base class

get_vec_normalize_env() → Optional[stable_baselines3.common.vec_env.vec_normalize.VecNormalize]
Return the `VecNormalize` wrapper of the training env if it exists. :return: Optional[VecNormalize] The `VecNormalize` env.

learn(*total_timesteps: int*, *callback: Union[None, Callable, List[stable_baselines3.common.callbacks.BaseCallback], stable_baselines3.common.callbacks.BaseCallback] = None*, *log_interval: int = 4*, *eval_env: Optional[Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv]] = None*, *eval_freq: int = - 1*, *n_eval_episodes: int = 5*, *tb_log_name: str = 'DDPG'*, *eval_log_path: Optional[str] = None*, *reset_num_timesteps: bool = True*) → *stable_baselines3.common.off_policy_algorithm.OffPolicyAlgorithm*
Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples (env steps) to train on
- **callback** – (MaybeCallback) callback(s) called at every step with state of the algorithm.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for TensorBoard logging
- **eval_env** – (gym.Env) Environment that will be used to evaluate the agent
- **eval_freq** – (int) Evaluate the agent every `eval_freq` timesteps (this may vary a little)
- **n_eval_episodes** – (int) Number of episode to evaluate the agent
- **eval_log_path** – (Optional[str]) Path to a folder where the evaluations will be saved
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseAlgorithm) the trained model

classmethod load(*load_path: str*, *env: Optional[Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv]] = None*, ***kwargs*) → *stable_baselines3.common.base_class.BaseAlgorithm*

Load the model from a zip-file

Parameters

- **load_path** – the location of the saved data
- **env** – 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
- **kwargs** – extra arguments to change the model when loading

load_replay_buffer(*path: Union[str, pathlib.Path, io.BufferedIOBase]*) → None

Load a replay buffer from a pickle file.

Parameters **path** – (Union[str, pathlib.Path, io.BufferedIOBase]) Path to the pickled replay buffer.

predict(*observation: numpy.ndarray*, *state: Optional[numpy.ndarray] = None*, *mask: Optional[numpy.ndarray] = None*, *deterministic: bool = False*) → Tuple[numpy.ndarray, Optional[numpy.ndarray]]

Get the model's action(s) from an observation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (Optional[np.ndarray]) The last states (can be None, used in recurrent policies)
- **mask** – (Optional[np.ndarray]) The last masks (can be None, used in recurrent policies)
- **deterministic** – (bool) Whether or not to return deterministic actions.

Returns (Tuple[np.ndarray, Optional[np.ndarray]]) the model's action and the next state (used in recurrent policies)

save(*path: Union[str, pathlib.Path, io.BufferedIOBase]*, *exclude: Optional[Iterable[str]] = None*, *include: Optional[Iterable[str]] = None*) → None

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

Parameters

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

save_replay_buffer(*path: Union[str, pathlib.Path, io.BufferedIOBase]*) → None

Save the replay buffer as a pickle file.

Parameters **path** – (Union[str,pathlib.Path, io.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.

set_env(*env: Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv]*) → None

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** – The environment for learning a policy

set_random_seed(*seed: Optional[int] = None*) → None
Set the seed of the pseudo-random generators (python, numpy, pytorch, gym, action_space)

Parameters **seed** – (int)

train(*gradient_steps: int*, *batch_size: int = 100*) → None
Sample the replay buffer and do the updates (gradient descent and update target networks)

1.18.5 DDPG Policies

`stable_baselines3.ddpg.MlpPolicy`
alias of `stable_baselines3.td3.policies.TD3Policy`

1.19 DQN

Deep Q Network (DQN)

Available Policies

MlpPolicy	alias of `stable_baselines3.dqn.policies.DQNPolicy`
CnnPolicy	Policy class for DQN when using images as input.

1.19.1 Notes

- Original paper: https://arxiv.org/abs/1312.5602
- Further reference: https://www.nature.com/articles/nature14236

Note: This implementation provides only vanilla Deep Q-Learning and has no extensions such as Double-DQN, Dueling-DQN and Prioritized Experience Replay.

1.19.2 Can I use?

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

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

1.19.3 Example

```
import gym
import numpy as np

from stable_baselines3 import DQN
from stable_baselines3.dqn import MlpPolicy

env = gym.make('Pendulum-v0')

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

del model # remove to demonstrate saving and loading

model = DQN.load("dqn_pendulum")

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

1.19.4 Parameters

class stable_baselines3.dqn.**DQN**(*policy: Union[str, Type[stable_baselines3.dqn.policies.DQNPolicy]]*, *env: Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv, str]*, *learning_rate: Union[float, Callable] = 0.0001*, *buffer_size: int = 1000000*, *learning_starts: int = 50000*, *batch_size: Optional[int] = 32*, *tau: float = 1.0*, *gamma: float = 0.99*, *train_freq: int = 4*, *gradient_steps: int = 1*, *n_episodes_rollout: int = - 1*, *optimize_memory_usage: bool = False*, *target_update_interval: int = 10000*, *exploration_fraction: float = 0.1*, *exploration_initial_eps: float = 1.0*, *exploration_final_eps: float = 0.05*, *max_grad_norm: float = 10*, *tensorboard_log: Optional[str] = None*, *create_eval_env: bool = False*, *policy_kwargs: Optional[Dict[str, Any]] = None*, *verbose: int = 0*, *seed: Optional[int] = None*, *device: Union[torch.device, str] = 'auto'*, *_init_setup_model: bool = 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 or str) The policy model to use (MlpPolicy, CnnPolicy, . . .)
- **env** – (Gym environment or str) The environment to learn from (if registered in Gym, can be str)

- **learning_rate** – (float or callable) The learning rate, it can be a function of the current progress (from 1 to 0)
- **buffer_size** – (int) size of the replay buffer
- **learning_starts** – (int) how many steps of the model to collect transitions for before learning starts
- **batch_size** – (int) Minibatch size for each gradient update
- **tau** – (float) the soft update coefficient ("Polyak update", between 0 and 1) default 1 for hard update
- **gamma** – (float) the discount factor
- **train_freq** – (int) Update the model every `train_freq` steps. Set to *-1* to disable.
- **gradient_steps** – (int) How many gradient steps to do after each rollout (see `train_freq` and `n_episodes_rollout`) Set to `-1` means to do as many gradient steps as steps done in the environment during the rollout.
- **n_episodes_rollout** – (int) Update the model every `n_episodes_rollout` episodes. Note that this cannot be used at the same time as `train_freq`. Set to *-1* to disable.
- **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
- **tensorboard_log** – (str) the log location for tensorboard (if None, no logging)
- **create_eval_env** – (bool) Whether to create a second environment that will be used for evaluating the agent periodically. (Only available when passing string for the environment)
- **policy_kwargs** – (dict) additional arguments to be passed to the policy on creation
- **verbose** – (int) the verbosity level: 0 no output, 1 info, 2 debug
- **seed** – (int) Seed for the pseudo random generators
- **device** – (str or th.device) 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: stable_baselines3.common.vec_env.base_vec_env.VecEnv*, *callback:* stable_baselines3.common.callbacks.BaseCallback, *n_episodes: int = 1*, *n_steps: int = - 1*, *action_noise: Optional[*stable_baselines3.common.noise.ActionNoise*] = None*, *learning_starts: int = 0*, *replay_buffer: Optional[stable_baselines3.common.buffers.ReplayBuffer] = None*, *log_interval: Optional[int] = None*) → stable_baselines3.common.type_aliases.RolloutReturn

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)
- **n_episodes** – (int) Number of episodes to use to collect rollout data You can also specify a `n_steps` instead
- **n_steps** – (int) Number of steps to use to collect rollout data You can also specify a `n_episodes` instead.
- **action_noise** – (Optional[ActionNoise]) Action noise that will be used for exploration Required for deterministic policy (e.g. TD3). This can also be used in addition to the stochastic policy for SAC.
- **learning_starts** – (int) Number of steps before learning for the warm-up phase.
- **replay_buffer** – (ReplayBuffer)
- **log_interval** – (int) Log data every `log_interval` episodes

Returns (RolloutReturn)

excluded_save_params() → List[str]
Returns the names of the parameters that should be excluded by default when saving the model.

Returns (List[str]) List of parameters that should be excluded from save

get_env() → Optional[stable_baselines3.common.vec_env.base_vec_env.VecEnv]
Returns the current environment (can be None if not defined).

Returns (Optional[VecEnv]) The current environment

get_torch_variables() → Tuple[List[str], List[str]]
cf base class

get_vec_normalize_env() → Optional[stable_baselines3.common.vec_env.vec_normalize.VecNormalize]
Return the `VecNormalize` wrapper of the training env if it exists. :return: Optional[VecNormalize] The `VecNormalize` env.

learn(*total_timesteps: int*, *callback: Union[None, Callable, List[stable_baselines3.common.callbacks.BaseCallback], stable_baselines3.common.callbacks.BaseCallback] = None*, *log_interval: int = 4*, *eval_env: Optional[Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv]] = None*, *eval_freq: int = - 1*, *n_eval_episodes: int = 5*, *tb_log_name: str = 'DQN'*, *eval_log_path: Optional[str] = None*, *reset_num_timesteps: bool = True*) → *stable_baselines3.common.off_policy_algorithm.OffPolicyAlgorithm*
Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples (env steps) to train on
- **callback** – (MaybeCallback) callback(s) called at every step with state of the algorithm.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for TensorBoard logging
- **eval_env** – (gym.Env) Environment that will be used to evaluate the agent

- **eval_freq** – (int) Evaluate the agent every `eval_freq` timesteps (this may vary a little)
- **n_eval_episodes** – (int) Number of episode to evaluate the agent
- **eval_log_path** – (Optional[str]) Path to a folder where the evaluations will be saved
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseAlgorithm) the trained model

classmethod load(*load_path: str*, *env: Optional[Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv]] = None*, ***kwargs*) → *stable_baselines3.common.base_class.BaseAlgorithm*

Load the model from a zip-file

Parameters

- **load_path** – the location of the saved data
- **env** – 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
- **kwargs** – extra arguments to change the model when loading

load_replay_buffer(*path: Union[str, pathlib.Path, io.BufferedIOBase]*) → None

Load a replay buffer from a pickle file.

Parameters **path** – (Union[str, pathlib.Path, io.BufferedIOBase]) Path to the pickled replay buffer.

predict(*observation: numpy.ndarray*, *state: Optional[numpy.ndarray] = None*, *mask: Optional[numpy.ndarray] = None*, *deterministic: bool = False*) → Tuple[numpy.ndarray, Optional[numpy.ndarray]]

Overrides the base_class predict function to include epsilon-greedy exploration.

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (Optional[np.ndarray]) The last states (can be None, used in recurrent policies)
- **mask** – (Optional[np.ndarray]) The last masks (can be None, used in recurrent policies)
- **deterministic** – (bool) Whether or not to return deterministic actions.

Returns (Tuple[np.ndarray, Optional[np.ndarray]]) the model's action and the next state (used in recurrent policies)

save(*path: Union[str, pathlib.Path, io.BufferedIOBase]*, *exclude: Optional[Iterable[str]] = None*, *include: Optional[Iterable[str]] = None*) → None

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

Parameters

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

save_replay_buffer(*path: Union[str, pathlib.Path, io.BufferedIOBase]*) → None

Save the replay buffer as a pickle file.

Parameters **path** – (Union[str,pathlib.Path, io.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.

set_env(*env: Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv]*) → None
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** – The environment for learning a policy

set_random_seed(*seed: Optional[int] = None*) → None
Set the seed of the pseudo-random generators (python, numpy, pytorch, gym, action_space)

Parameters **seed** – (int)

train(*gradient_steps: int*, *batch_size: int = 100*) → None
Sample the replay buffer and do the updates (gradient descent and update target networks)

1.19.5 DQN Policies

stable_baselines3.dqn.**MlpPolicy**
alias of stable_baselines3.dqn.policies.DQNPolicy

class stable_baselines3.dqn.**CnnPolicy**(*observation_space: gym.spaces.space.Space*, *action_space: gym.spaces.space.Space*, *lr_schedule: Callable*, *net_arch: Optional[List[int]] = None*, *device: Union[torch.device, str] = 'auto'*, *activation_fn: Type[torch.nn.modules.module.Module] = <class 'torch.nn.modules.activation.ReLU'>*, *features_extractor_class: Type[stable_baselines3.common.torch_layers.BaseFeaturesExtractor] = <class 'stable_baselines3.common.torch_layers.NatureCNN'>*, *features_extractor_kwargs: Optional[Dict[str, Any]] = None*, *normalize_images: bool = True*, *optimizer_class: Type[torch.optim.optimizer.Optimizer] = <class 'torch.optim.adam.Adam'>*, *optimizer_kwargs: Optional[Dict[str, Any]] = None*)
Policy class for DQN when using images as input.

Parameters

- **observation_space** – (gym.spaces.Space) Observation space
- **action_space** – (gym.spaces.Space) Action space
- **lr_schedule** – (callable) Learning rate schedule (could be constant)
- **net_arch** – (Optional[List[int]]) The specification of the policy and value networks.
- **device** – (str or th.device) Device on which the code should run.
- **activation_fn** – (Type[nn.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[th.optim.Optimizer]) The optimizer to use, `th.optim.Adam` by default

- **optimizer_kwargs** – (Optional[Dict[str, Any]]) Additional keyword arguments, excluding the learning rate, to pass to the optimizer

1.20 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 form 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 .

1.20.1 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://blog.openai.com/openai-baselines-ppo/
- Spinning Up guide: https://spinningup.openai.com/en/latest/algorithms/ppo.html

1.20.2 Can I use?

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

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

1.20.3 Example

Train a PPO agent on `Pendulum-v0` using 4 environments.

```
import gym

from stable_baselines3 import PPO
from stable_baselines3.ppo import MlpPolicy
from stable_baselines3.common.cmd_util import make_vec_env

# Parallel environments
env = make_vec_env('CartPole-v1', n_envs=4)
```

(continues on next page)

(continued from previous page)

```
model = PPO(MlpPolicy, 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 = env.reset()
while True:
    action, _states = model.predict(obs)
    obs, rewards, dones, info = env.step(action)
    env.render()
```

1.20.4 Parameters

class stable_baselines3.ppo.**PPO**(*policy: Union[str, Type[stable_baselines3.common.policies.ActorCriticPolicy]], env: Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv, str], learning_rate: Union[float, Callable] = 0.0003, n_steps: int = 2048, batch_size: Optional[int] = 64, n_epochs: int = 10, gamma: float = 0.99, gae_lambda: float = 0.95, clip_range: float = 0.2, clip_range_vf: Optional[float] = None, ent_coef: float = 0.0, vf_coef: float = 0.5, max_grad_norm: float = 0.5, use_sde: bool = False, sde_sample_freq: int = - 1, target_kl: Optional[float] = None, tensorboard_log: Optional[str] = None, create_eval_env: bool = False, policy_kwargs: Optional[Dict[str, Any]] = None, verbose: int = 0, seed: Optional[int] = None, device: Union[torch.device, str] = 'auto', _init_setup_model: bool = 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 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 or str) The policy model to use (MlpPolicy, CnnPolicy, ...)
- **env** – (Gym environment or str) The environment to learn from (if registered in Gym, can be str)
- **learning_rate** – (float or callable) The learning rate, it can be a function of the current progress remaining (from 1 to 0)
- **n_steps** – (int) The number of steps to run for each environment per update (i.e. batch size is n_steps * n_env where n_env is number of environment copies running in parallel)
- **batch_size** – (int) 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 or callable) Clipping parameter, it can be a function of the current progress remaining (from 1 to 0).
- **clip_range_vf** – (float or callable) 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.
- **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)
- **target_kl** – (float) Limit the KL divergence between updates, because the clipping is not enough to prevent large update see issue #213 (cf https://github.com/hill-a/stable-baselines/issues/213) By default, there is no limit on the kl div.
- **tensorboard_log** – (str) the log location for tensorboard (if None, no logging)
- **create_eval_env** – (bool) Whether to create a second environment that will be used for evaluating the agent periodically. (Only available when passing string for the environment)
- **policy_kwargs** – (dict) additional arguments to be passed to the policy on creation
- **verbose** – (int) the verbosity level: 0 no output, 1 info, 2 debug
- **seed** – (int) Seed for the pseudo random generators
- **device** – (str or th.device) 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: stable_baselines3.common.vec_env.base_vec_env.VecEnv*, *callback:* stable_baselines3.common.callbacks.BaseCallback, *rollout_buffer: stable_baselines3.common.buffers.RolloutBuffer*, *n_rollout_steps: int*) → bool

Collect rollouts using the current policy and fill a *RolloutBuffer*.

Parameters

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

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

excluded_save_params() → List[str]

Returns the names of the parameters that should be excluded by default when saving the model.

Returns ([str]) List of parameters that should be excluded from save

get_env() → Optional[stable_baselines3.common.vec_env.base_vec_env.VecEnv]
Returns the current environment (can be None if not defined).

Returns (Optional[VecEnv]) The current environment

get_torch_variables() → Tuple[List[str], List[str]]
cf base class

get_vec_normalize_env() → Optional[stable_baselines3.common.vec_env.vec_normalize.VecNormalize]
Return the `VecNormalize` wrapper of the training env if it exists. :return: Optional[VecNormalize] The `VecNormalize` env.

learn(*total_timesteps: int*, *callback: Union[None, Callable, List[stable_baselines3.common.callbacks.BaseCallback], stable_baselines3.common.callbacks.BaseCallback] = None*, *log_interval: int = 1*, *eval_env: Optional[Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv]] = None*, *eval_freq: int = - 1*, *n_eval_episodes: int = 5*, *tb_log_name: str = 'PPO'*, *eval_log_path: Optional[str] = None*, *reset_num_timesteps: bool = True*) → stable_baselines3.ppo.ppo.PPO
Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples (env steps) to train on
- **callback** – (MaybeCallback) callback(s) called at every step with state of the algorithm.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for TensorBoard logging
- **eval_env** – (gym.Env) Environment that will be used to evaluate the agent
- **eval_freq** – (int) Evaluate the agent every `eval_freq` timesteps (this may vary a little)
- **n_eval_episodes** – (int) Number of episode to evaluate the agent
- **eval_log_path** – (Optional[str]) Path to a folder where the evaluations will be saved
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseAlgorithm) the trained model

classmethod load(*load_path: str*, *env: Optional[Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv]] = None*, ***kwargs*) → *stable_baselines3.common.base_class.BaseAlgorithm*
Load the model from a zip-file

Parameters

- **load_path** – the location of the saved data
- **env** – 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
- **kwargs** – extra arguments to change the model when loading

predict(*observation: numpy.ndarray*, *state: Optional[numpy.ndarray] = None*, *mask: Optional[numpy.ndarray] = None*, *deterministic: bool = False*) → Tuple[numpy.ndarray, Optional[numpy.ndarray]]
Get the model's action(s) from an observation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (Optional[np.ndarray]) The last states (can be None, used in recurrent policies)
- **mask** – (Optional[np.ndarray]) The last masks (can be None, used in recurrent policies)
- **deterministic** – (bool) Whether or not to return deterministic actions.

Returns (Tuple[np.ndarray, Optional[np.ndarray]]) the model's action and the next state (used in recurrent policies)

save(*path: Union[str, pathlib.Path, io.BufferedIOBase]*, *exclude: Optional[Iterable[str]] = None*, *include: Optional[Iterable[str]] = None*) → None
Save all the attributes of the object and the model parameters in a zip-file.

Parameters

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

set_env(*env: Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv]*) → None
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** – The environment for learning a policy

set_random_seed(*seed: Optional[int] = None*) → None
Set the seed of the pseudo-random generators (python, numpy, pytorch, gym, action_space)

Parameters **seed** – (int)

train() → None
Update policy using the currently gathered rollout buffer.

1.21 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 stable_baselines3.sac.policies.SACPolicy
CnnPolicy	Policy class (with both actor and critic) for SAC.

1.21.1 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: The default policies for SAC differ a bit from others MlpPolicy: it uses ReLU instead of tanh activation, to match the original paper

1.21.2 Can I use?

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

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

1.21.3 Example

```
import gym
import numpy as np

from stable_baselines3 import SAC
from stable_baselines3.sac import MlpPolicy

env = gym.make('Pendulum-v0')

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 = env.reset()
while True:
```

(continues on next page)

(continued from previous page)

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

1.21.4 Parameters

class stable_baselines3.sac.**SAC**(*policy: Union[str, Type[stable_baselines3.sac.policies.SACPolicy]], env: Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv, str], learning_rate: Union[float, Callable] = 0.0003, buffer_size: int = 1000000, learning_starts: int = 100, batch_size: int = 256, tau: float = 0.005, gamma: float = 0.99, train_freq: int = 1, gradient_steps: int = 1, n_episodes_rollout: int = - 1, action_noise: Optional[stable_baselines3.common.noise.ActionNoise] = None, optimize_memory_usage: bool = False, ent_coef: Union[str, float] = 'auto', target_update_interval: int = 1, target_entropy: Union[str, float] = 'auto', use_sde: bool = False, sde_sample_freq: int = - 1, use_sde_at_warmup: bool = False, tensorboard_log: Optional[str] = None, create_eval_env: bool = False, policy_kwargs: Dict[str, Any] = None, verbose: int = 0, seed: Optional[int] = None, device: Union[torch.device, str] = 'auto', _init_setup_model: bool = 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 or str) The policy model to use (MlpPolicy, CnnPolicy, . . .)
- **env** – (GymEnv or str) The environment to learn from (if registered in Gym, can be str)
- **learning_rate** – (float or callable) learning rate for adam optimizer, the same learning rate will be used for all networks (Q-Values, Actor and Value function) it can be a function of the current progress remaining (from 1 to 0)
- **buffer_size** – (int) size of the replay buffer
- **learning_starts** – (int) how many steps of the model to collect transitions for before learning starts
- **batch_size** – (int) Minibatch size for each gradient update
- **tau** – (float) the soft update coefficient ("Polyak update", between 0 and 1)
- **gamma** – (float) the discount factor
- **train_freq** – (int) Update the model every `train_freq` steps. Set to *-1* to disable.

- **gradient_steps** – (int) How many gradient steps to do after each rollout (see `train_freq` and `n_episodes_rollout`) Set to `-1` means to do as many gradient steps as steps done in the environment during the rollout.
- **n_episodes_rollout** – (int) Update the model every `n_episodes_rollout` episodes. Note that this cannot be used at the same time as `train_freq`. Set to *-1* to disable.
- **action_noise** – (ActionNoise) the action noise type (None by default), this can help for hard exploration problem. Cf common.noise for the different action noise type.
- **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 or 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 or 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)
- **create_eval_env** – (bool) Whether to create a second environment that will be used for evaluating the agent periodically. (Only available when passing string for the environment)
- **policy_kwargs** – (dict) additional arguments to be passed to the policy on creation
- **verbose** – (int) the verbosity level: 0 no output, 1 info, 2 debug
- **seed** – (int) Seed for the pseudo random generators
- **device** – (str or th.device) 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: stable_baselines3.common.vec_env.base_vec_env.VecEnv*, *callback:* stable_baselines3.common.callbacks.BaseCallback, *n_episodes: int = 1*, *n_steps: int = - 1*, *action_noise: Optional[*stable_baselines3.common.noise.ActionNoise*]* *= None*, *learning_starts: int = 0*, *replay_buffer: Optional[stable_baselines3.common.buffers.ReplayBuffer]* *= None*, *log_interval: Optional[int] = None*) → stable_baselines3.common.type_aliases.RolloutReturn

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)

- **n_episodes** – (int) Number of episodes to use to collect rollout data You can also specify a `n_steps` instead
- **n_steps** – (int) Number of steps to use to collect rollout data You can also specify a `n_episodes` instead.
- **action_noise** – (Optional[ActionNoise]) Action noise that will be used for exploration Required for deterministic policy (e.g. TD3). This can also be used in addition to the stochastic policy for SAC.
- **learning_starts** – (int) Number of steps before learning for the warm-up phase.
- **replay_buffer** – (ReplayBuffer)
- **log_interval** – (int) Log data every `log_interval` episodes

Returns (RolloutReturn)

excluded_save_params() → List[str]
Returns the names of the parameters that should be excluded by default when saving the model.

Returns (List[str]) List of parameters that should be excluded from save

get_env() → Optional[stable_baselines3.common.vec_env.base_vec_env.VecEnv]
Returns the current environment (can be None if not defined).

Returns (Optional[VecEnv]) The current environment

get_torch_variables() → Tuple[List[str], List[str]]
cf base class

get_vec_normalize_env() → Optional[stable_baselines3.common.vec_env.vec_normalize.VecNormalize]
Return the `VecNormalize` wrapper of the training env if it exists. :return: Optional[VecNormalize] The `VecNormalize` env.

learn(*total_timesteps: int*, *callback: Union[None, Callable, List[stable_baselines3.common.callbacks.BaseCallback], stable_baselines3.common.callbacks.BaseCallback] = None*, *log_interval: int = 4*, *eval_env: Optional[Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv]] = None*, *eval_freq: int = - 1*, *n_eval_episodes: int = 5*, *tb_log_name: str = 'SAC'*, *eval_log_path: Optional[str] = None*, *reset_num_timesteps: bool = True*) → *stable_baselines3.common.off_policy_algorithm.OffPolicyAlgorithm*
Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples (env steps) to train on
- **callback** – (MaybeCallback) callback(s) called at every step with state of the algorithm.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for TensorBoard logging
- **eval_env** – (gym.Env) Environment that will be used to evaluate the agent
- **eval_freq** – (int) Evaluate the agent every `eval_freq` timesteps (this may vary a little)
- **n_eval_episodes** – (int) Number of episode to evaluate the agent
- **eval_log_path** – (Optional[str]) Path to a folder where the evaluations will be saved
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseAlgorithm) the trained model

classmethod load(*load_path: str*, *env: Optional[Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv]] = None*, ***kwargs*) → *stable_baselines3.common.base_class.BaseAlgorithm*

Load the model from a zip-file

Parameters

- **load_path** – the location of the saved data
- **env** – 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
- **kwargs** – extra arguments to change the model when loading

load_replay_buffer(*path: Union[str, pathlib.Path, io.BufferedIOBase]*) → None

Load a replay buffer from a pickle file.

Parameters **path** – (Union[str, pathlib.Path, io.BufferedIOBase]) Path to the pickled replay buffer.

predict(*observation: numpy.ndarray*, *state: Optional[numpy.ndarray] = None*, *mask: Optional[numpy.ndarray] = None*, *deterministic: bool = False*) → Tuple[numpy.ndarray, Optional[numpy.ndarray]]

Get the model's action(s) from an observation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (Optional[np.ndarray]) The last states (can be None, used in recurrent policies)
- **mask** – (Optional[np.ndarray]) The last masks (can be None, used in recurrent policies)
- **deterministic** – (bool) Whether or not to return deterministic actions.

Returns (Tuple[np.ndarray, Optional[np.ndarray]]) the model's action and the next state (used in recurrent policies)

save(*path: Union[str, pathlib.Path, io.BufferedIOBase]*, *exclude: Optional[Iterable[str]] = None*, *include: Optional[Iterable[str]] = None*) → None

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

Parameters

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

save_replay_buffer(*path: Union[str, pathlib.Path, io.BufferedIOBase]*) → None

Save the replay buffer as a pickle file.

Parameters **path** – (Union[str,pathlib.Path, io.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.

set_env(*env: Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv]*) → None

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** – The environment for learning a policy

set_random_seed(*seed: Optional[int] = None*) → None
Set the seed of the pseudo-random generators (python, numpy, pytorch, gym, action_space)

Parameters **seed** – (int)

train(*gradient_steps: int*, *batch_size: int = 64*) → None
Sample the replay buffer and do the updates (gradient descent and update target networks)

1.21.5 SAC Policies

stable_baselines3.sac.**MlpPolicy**
alias of stable_baselines3.sac.policies.SACPolicy

1.22 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 stable_baselines3.td3.policies.TD3Policy

1.22.1 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

1.22.2 Can I use?

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

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

1.22.3 Example

```
import gym
import numpy as np

from stable_baselines3 import TD3
from stable_baselines3.td3.policies import MlpPolicy
from stable_baselines3.common.noise import NormalActionNoise,␣
↪OrnsteinUhlenbeckActionNoise

env = gym.make('Pendulum-v0')

# 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")
env = model.get_env()

del model # remove to demonstrate saving and loading

model = TD3.load("td3_pendulum")

obs = env.reset()
while True:
    action, _states = model.predict(obs)
    obs, rewards, dones, info = env.step(action)
    env.render()
```

1.22.4 Parameters

class stable_baselines3.td3.**TD3**(*policy: Union[str, Type[stable_baselines3.td3.policies.TD3Policy]], env: Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv, str], learning_rate: Union[float, Callable] = 0.001, buffer_size: int = 1000000, learning_starts: int = 100, batch_size: int = 100, tau: float = 0.005, gamma: float = 0.99, train_freq: int = - 1, gradient_steps: int = - 1, n_episodes_rollout: int = 1, action_noise: Optional[stable_baselines3.common.noise.ActionNoise] = None, optimize_memory_usage: bool = False, policy_delay: int = 2, target_policy_noise: float = 0.2, target_noise_clip: float = 0.5, use_sde: bool = False, sde_sample_freq: int = - 1, sde_max_grad_norm: float = 1, sde_ent_coef: float = 0.0, sde_log_std_scheduler: Optional[Callable] = None, use_sde_at_warmup: bool = False, tensorboard_log: Optional[str] = None, create_eval_env: bool = False, policy_kwargs: Dict[str, Any] = None, verbose: int = 0, seed: Optional[int] = None, device: Union[torch.device, str] = 'auto', _init_setup_model: bool = 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 or str) The policy model to use (MlpPolicy, CnnPolicy, . . .)
- **env** – (GymEnv or str) The environment to learn from (if registered in Gym, can be str)
- **learning_rate** – (float or callable) learning rate for adam optimizer, the same learning rate will be used for all networks (Q-Values, Actor and Value function) it can be a function of the current progress remaining (from 1 to 0)
- **buffer_size** – (int) size of the replay buffer
- **learning_starts** – (int) how many steps of the model to collect transitions for before learning starts
- **batch_size** – (int) Minibatch size for each gradient update
- **tau** – (float) the soft update coefficient ("Polyak update", between 0 and 1)
- **gamma** – (float) the discount factor
- **train_freq** – (int) Update the model every `train_freq` steps. Set to *-1* to disable.
- **gradient_steps** – (int) How many gradient steps to do after each rollout (see `train_freq` and `n_episodes_rollout`) Set to `-1` means to do as many gradient steps as steps done in the environment during the rollout.
- **n_episodes_rollout** – (int) Update the model every `n_episodes_rollout` episodes. Note that this cannot be used at the same time as `train_freq`. Set to *-1* to disable.
- **action_noise** – (ActionNoise) the action noise type (None by default), this can help for hard exploration problem. Cf common.noise for the different action noise type.

- **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.
- **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 SDE Default: -1 (only sample at the beginning of the rollout)
- **sde_max_grad_norm** – (float)
- **sde_ent_coef** – (float)
- **sde_log_std_scheduler** – (callable)
- **use_sde_at_warmup** – (bool) Whether to use SDE instead of uniform sampling during the warm up phase (before learning starts)
- **create_eval_env** – (bool) Whether to create a second environment that will be used for evaluating the agent periodically. (Only available when passing string for the environment)
- **policy_kwargs** – (dict) additional arguments to be passed to the policy on creation
- **verbose** – (int) the verbosity level: 0 no output, 1 info, 2 debug
- **seed** – (int) Seed for the pseudo random generators
- **device** – (str or th.device) 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: stable_baselines3.common.vec_env.base_vec_env.VecEnv*, *callback:* stable_baselines3.common.callbacks.BaseCallback, *n_episodes: int = 1*, *n_steps: int = -1*, *action_noise: Optional[*stable_baselines3.common.noise.ActionNoise*] = None*, *learning_starts: int = 0*, *replay_buffer: Optional[stable_baselines3.common.buffers.ReplayBuffer] = None*, *log_interval: Optional[int] = None*) → stable_baselines3.common.type_aliases.RolloutReturn

Collect rollout using the current policy (and possibly fill the replay buffer)

Parameters

- **env** – (VecEnv) The training environment
- **n_episodes** – (int) Number of episodes to use to collect rollout data You can also specify a `n_steps` instead
- **n_steps** – (int) Number of steps to use to collect rollout data You can also specify a `n_episodes` instead.
- **action_noise** – (Optional[ActionNoise]) Action noise that will be used for exploration Required for deterministic policy (e.g. TD3). This can also be used in addition to the stochastic policy for SAC.

- **callback** – (BaseCallback) Callback that will be called at each step (and at the beginning and end of the rollout)
- **learning_starts** – (int) Number of steps before learning for the warm-up phase.
- **replay_buffer** – (ReplayBuffer)
- **log_interval** – (int) Log data every `log_interval` episodes

Returns (RolloutReturn)

excluded_save_params() → List[str]
Returns the names of the parameters that should be excluded by default when saving the model.

Returns (List[str]) List of parameters that should be excluded from save

get_env() → Optional[stable_baselines3.common.vec_env.base_vec_env.VecEnv]
Returns the current environment (can be None if not defined).

Returns (Optional[VecEnv]) The current environment

get_torch_variables() → Tuple[List[str], List[str]]
cf base class

get_vec_normalize_env() → Optional[stable_baselines3.common.vec_env.vec_normalize.VecNormalize]
Return the `VecNormalize` wrapper of the training env if it exists. :return: Optional[VecNormalize] The `VecNormalize` env.

learn(*total_timesteps: int*, *callback: Union[None, Callable, List[stable_baselines3.common.callbacks.BaseCallback], stable_baselines3.common.callbacks.BaseCallback] = None*, *log_interval: int = 4*, *eval_env: Optional[Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv]] = None*, *eval_freq: int = - 1*, *n_eval_episodes: int = 5*, *tb_log_name: str = 'TD3'*, *eval_log_path: Optional[str] = None*, *reset_num_timesteps: bool = True*) → *stable_baselines3.common.off_policy_algorithm.OffPolicyAlgorithm*
Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples (env steps) to train on
- **callback** – (MaybeCallback) callback(s) called at every step with state of the algorithm.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for TensorBoard logging
- **eval_env** – (gym.Env) Environment that will be used to evaluate the agent
- **eval_freq** – (int) Evaluate the agent every `eval_freq` timesteps (this may vary a little)
- **n_eval_episodes** – (int) Number of episode to evaluate the agent
- **eval_log_path** – (Optional[str]) Path to a folder where the evaluations will be saved
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseAlgorithm) the trained model

classmethod load(*load_path: str*, *env: Optional[Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv]] = None*, ***kwargs*) → *stable_baselines3.common.base_class.BaseAlgorithm*
Load the model from a zip-file

Parameters

- **load_path** – the location of the saved data
- **env** – 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
- **kwargs** – extra arguments to change the model when loading

load_replay_buffer(*path: Union[str, pathlib.Path, io.BufferedIOBase]*) → None
Load a replay buffer from a pickle file.

Parameters **path** – (Union[str, pathlib.Path, io.BufferedIOBase]) Path to the pickled replay buffer.

predict(*observation: numpy.ndarray*, *state: Optional[numpy.ndarray] = None*, *mask: Optional[numpy.ndarray] = None*, *deterministic: bool = False*) → Tuple[numpy.ndarray, Optional[numpy.ndarray]]
Get the model's action(s) from an observation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (Optional[np.ndarray]) The last states (can be None, used in recurrent policies)
- **mask** – (Optional[np.ndarray]) The last masks (can be None, used in recurrent policies)
- **deterministic** – (bool) Whether or not to return deterministic actions.

Returns (Tuple[np.ndarray, Optional[np.ndarray]]) the model's action and the next state (used in recurrent policies)

save(*path: Union[str, pathlib.Path, io.BufferedIOBase]*, *exclude: Optional[Iterable[str]] = None*, *include: Optional[Iterable[str]] = None*) → None
Save all the attributes of the object and the model parameters in a zip-file.

Parameters

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

save_replay_buffer(*path: Union[str, pathlib.Path, io.BufferedIOBase]*) → None
Save the replay buffer as a pickle file.

Parameters **path** – (Union[str,pathlib.Path, io.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.

set_env(*env: Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecEnv]*) → None
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** – The environment for learning a policy

set_random_seed(*seed: Optional[int] = None*) → None
Set the seed of the pseudo-random generators (python, numpy, pytorch, gym, action_space)

Parameters **seed** – (int)

train(*gradient_steps: int*, *batch_size: int = 100*) → None
Sample the replay buffer and do the updates (gradient descent and update target networks)

1.22.5 TD3 Policies

stable_baselines3.td3.**MlpPolicy**
alias of stable_baselines3.td3.policies.TD3Policy

1.23 Atari Wrappers

class stable_baselines3.common.atari_wrappers.**AtariWrapper**(*env: gym.core.Env*, *noop_max: int = 30*, *frame_skip: int = 4*, *screen_size: int = 84*, *terminal_on_life_loss: bool = True*, *clip_reward: bool = True*)

Atari 2600 preprocessings

Specifically:

- NoopReset: 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}

Parameters

- **env** – (gym.Env) gym environment
- **noop_max** – (int): max number of no-ops
- **frame_skip** – (int): the frequency at which the agent experiences the game.
- **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.

class stable_baselines3.common.atari_wrappers.**ClipRewardEnv**(*env: gym.core.Env*)

reward(*reward: float*) → float
Bin reward to {+1, 0, -1} by its sign.

Parameters **reward** – (float)

Returns (float)

class stable_baselines3.common.atari_wrappers.**EpisodicLifeEnv**(*env: gym.core.Env*)

reset(***kwargs*) → numpy.ndarray
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 (np.ndarray) the first observation of the environment

step(*action: int*) → Tuple[Union[Tuple, Dict[str, Any], numpy.ndarray, int], float, bool, Dict]
Run one timestep of the environment's dynamics. When end of episode is reached, you are responsible for calling *reset()* to reset this environment's state.

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

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

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

class stable_baselines3.common.atari_wrappers.**FireResetEnv**(*env: gym.core.Env*)

reset(***kwargs*) → numpy.ndarray
Resets the state of the environment and returns an initial observation.

Returns: observation (object): the initial observation.

class stable_baselines3.common.atari_wrappers.**MaxAndSkipEnv**(*env: gym.core.Env*, *skip: int = 4*)

reset(***kwargs*)
Resets the state of the environment and returns an initial observation.

Returns: observation (object): the initial observation.

step(*action: int*) → Tuple[Union[Tuple, Dict[str, Any], numpy.ndarray, int], float, bool, Dict]
Step the environment with the given action Repeat action, sum reward, and max over last observations.

Parameters **action** – ([int] or [float]) the action

Returns ([int] or [float], [float], [bool], dict) observation, reward, done, information

class stable_baselines3.common.atari_wrappers.**NoopResetEnv**(*env: gym.core.Env*, *noop_max: int = 30*)

reset(***kwargs*) → numpy.ndarray
Resets the state of the environment and returns an initial observation.

Returns: observation (object): the initial observation.

class stable_baselines3.common.atari_wrappers.**WarpFrame**(*env: gym.core.Env*, *width: int = 84*, *height: int = 84*)

observation(*frame: numpy.ndarray*) → numpy.ndarray
returns the current observation from a frame

Parameters **frame** – (np.ndarray) environment frame

Returns (np.ndarray) the observation

1.24 Command Utils

stable_baselines3.common.cmd_util.**make_atari_env**(*env_id: Union[str, Type[gym.core.Env]], n_envs: int = 1, seed: Optional[int] = None, start_index: int = 0, monitor_dir: Optional[str] = None, wrapper_kwargs: Optional[Dict[str, Any]] = None, env_kwargs: Optional[Dict[str, Any]] = None, vec_env_cls: Optional[Union[stable_baselines3.common.vec_env.dummy_vec stable_baselines3.common.vec_env.subproc_vec_env.Subproc = None, vec_env_kwargs: Optional[Dict[str, Any]] = 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 or Type[gym.Env]) the environment ID or the environment class
- **n_envs** – (int) the number of environments you wish to have in parallel
- **seed** – (int) the initial seed for the random number generator
- **start_index** – (int) start rank index
- **monitor_dir** – (str) 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]) Optional keyword argument to pass to the `AtariWrapper`
- **env_kwargs** – (Dict[str, Any]) Optional keyword argument to pass to the env constructor
- **vec_env_cls** – (Type[VecEnv]) A custom `VecEnv` class constructor. Default: None.
- **vec_env_kwargs** – (Dict[str, Any]) Keyword arguments to pass to the `VecEnv` class constructor.

Returns (VecEnv) The wrapped environment

stable_baselines3.common.cmd_util.**make_vec_env**(*env_id: Union[str, Type[gym.core.Env]], n_envs: int = 1, seed: Optional[int] = None, start_index: int = 0, monitor_dir: Optional[str] = None, wrapper_class: Optional[Callable] = None, env_kwargs: Optional[Dict[str, Any]] = None, vec_env_cls: Optional[Union[stable_baselines3.common.vec_env.dummy_vec_en stable_baselines3.common.vec_env.subproc_vec_env.SubprocVec = None, vec_env_kwargs: Optional[Dict[str, Any]] = None*)

Create a wrapped, monitored `VecEnv`. By default it uses a `DummyVecEnv` which is usually faster than a `SubprocVecEnv`.

Parameters

- **env_id** – (str or Type[gym.Env]) the environment ID or the environment class

- **n_envs** – (int) the number of environments you wish to have in parallel
- **seed** – (int) the initial seed for the random number generator
- **start_index** – (int) start rank index
- **monitor_dir** – (str) 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** – (gym.Wrapper or callable) Additional wrapper to use on the environment. This can also be a function with single argument that wraps the environment in many things.
- **env_kwargs** – (dict) Optional keyword argument to pass to the env constructor
- **vec_env_cls** – (Type[VecEnv]) A custom `VecEnv` class constructor. Default: None.
- **vec_env_kwargs** – (dict) Keyword arguments to pass to the `VecEnv` class constructor.

Returns (VecEnv) The wrapped environment

1.25 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 to sample 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: int*)

Bernoulli distribution for MultiBinary action spaces.

Parameters **action_dim** – (int) Number of binary actions

actions_from_params(*action_logits: torch.Tensor*, *deterministic: bool = False*) → torch.Tensor
Returns samples from the probability distribution given its parameters.

Returns (th.Tensor) actions

entropy() → torch.Tensor
Returns Shannon's entropy of the probability

Returns (Optional[th.Tensor]) the entropy, or None if no analytical form is known

log_prob(*actions: torch.Tensor*) → torch.Tensor
Returns the log likelihood

Parameters **x** – (th.Tensor) the taken action

Returns (th.Tensor) The log likelihood of the distribution

log_prob_from_params(*action_logits: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]
Returns samples and the associated log probabilities from the probability distribution given its parameters.

Returns (th.Tuple[th.Tensor, th.Tensor]) actions and log prob

mode() → torch.Tensor
Returns the most likely action (deterministic output) from the probability distribution

Returns (th.Tensor) the stochastic action

proba_distribution(*action_logits: torch.Tensor*) → *stable_baselines3.common.distributions.BernoulliDistribution*
Set parameters of the distribution.

Returns (Distribution) self

proba_distribution_net(*latent_dim: int*) → torch.nn.modules.module.Module
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 (nn.Linear)

sample() → torch.Tensor
Returns a sample from the probability distribution

Returns (th.Tensor) the stochastic action

class stable_baselines3.common.distributions.**CategoricalDistribution**(*action_dim: int*)
Categorical distribution for discrete actions.

Parameters **action_dim** – (int) Number of discrete actions

actions_from_params(*action_logits: torch.Tensor*, *deterministic: bool = False*) → torch.Tensor
Returns samples from the probability distribution given its parameters.

Returns (th.Tensor) actions

entropy() → torch.Tensor
Returns Shannon's entropy of the probability

Returns (Optional[th.Tensor]) the entropy, or None if no analytical form is known

log_prob(*actions: torch.Tensor*) → torch.Tensor
Returns the log likelihood

Parameters **x** – (th.Tensor) the taken action

Returns (th.Tensor) The log likelihood of the distribution

log_prob_from_params(*action_logits: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]
Returns samples and the associated log probabilities from the probability distribution given its parameters.

Returns (th.Tuple[th.Tensor, th.Tensor]) actions and log prob

mode() → torch.Tensor
Returns the most likely action (deterministic output) from the probability distribution

Returns (th.Tensor) the stochastic action

proba_distribution(*action_logits: torch.Tensor*) → *stable_baselines3.common.distributions.CategoricalDistribution*
Set parameters of the distribution.

Returns (Distribution) self

proba_distribution_net(*latent_dim: int*) → torch.nn.modules.module.Module
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 (nn.Linear)

sample() → torch.Tensor
Returns a sample from the probability distribution

Returns (th.Tensor) the stochastic action

class stable_baselines3.common.distributions.**DiagGaussianDistribution**(*action_dim: int*)
Gaussian distribution with diagonal covariance matrix, for continuous actions.

Parameters **action_dim** – (int) Dimension of the action space.

actions_from_params(*mean_actions: torch.Tensor*, *log_std: torch.Tensor*, *deterministic: bool = False*) → torch.Tensor
Returns samples from the probability distribution given its parameters.

Returns (th.Tensor) actions

entropy() → torch.Tensor
Returns Shannon's entropy of the probability

Returns (Optional[th.Tensor]) the entropy, or None if no analytical form is known

log_prob(*actions: torch.Tensor*) → torch.Tensor
Get the log probabilities of actions according to the distribution. Note that you must first call the proba_distribution() method.

Parameters **actions** – (th.Tensor)

Returns (th.Tensor)

log_prob_from_params(*mean_actions: torch.Tensor*, *log_std: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]
Compute the log probability of taking an action given the distribution parameters.

Parameters

- **mean_actions** – (th.Tensor)
- **log_std** – (th.Tensor)

Returns (Tuple[th.Tensor, th.Tensor])

mode() → torch.Tensor
Returns the most likely action (deterministic output) from the probability distribution

Returns (th.Tensor) the stochastic action

proba_distribution(*mean_actions: torch.Tensor*, *log_std: torch.Tensor*) → *stable_baselines3.common.distributions.DiagGaussianDistribution*
Create the distribution given its parameters (mean, std)

Parameters

- **mean_actions** – (th.Tensor)
- **log_std** – (th.Tensor)

Returns (DiagGaussianDistribution)

proba_distribution_net(*latent_dim: int*, *log_std_init: float = 0.0*) → Tuple[torch.nn.modules.module.Module, torch.nn.parameter.Parameter]
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 (nn.Linear, nn.Parameter)

sample() → torch.Tensor
Returns a sample from the probability distribution

Returns (th.Tensor) the stochastic action

class stable_baselines3.common.distributions.**Distribution**
Abstract base class for distributions.

abstract actions_from_params(**args*, ***kwargs*) → torch.Tensor
Returns samples from the probability distribution given its parameters.

Returns (th.Tensor) actions

abstract entropy() → Optional[torch.Tensor]
Returns Shannon's entropy of the probability

Returns (Optional[th.Tensor]) the entropy, or None if no analytical form is known

get_actions(*deterministic: bool = False*) → torch.Tensor
Return actions according to the probability distribution.

Parameters **deterministic** – (bool)

Returns (th.Tensor)

abstract log_prob(*x: torch.Tensor*) → torch.Tensor
Returns the log likelihood

Parameters **x** – (th.Tensor) the taken action

Returns (th.Tensor) The log likelihood of the distribution

abstract log_prob_from_params(**args*, ***kwargs*) → Tuple[torch.Tensor, torch.Tensor]
Returns samples and the associated log probabilities from the probability distribution given its parameters.

Returns (th.Tuple[th.Tensor, th.Tensor]) actions and log prob

abstract mode() → torch.Tensor
Returns the most likely action (deterministic output) from the probability distribution

Returns (th.Tensor) the stochastic action

abstract proba_distribution(**args*, ***kwargs*) → *stable_baselines3.common.distributions.Distribution*
Set parameters of the distribution.

Returns (Distribution) self

abstract 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.

abstract sample() → torch.Tensor
Returns a sample from the probability distribution

Returns (th.Tensor) the stochastic action

class stable_baselines3.common.distributions.**MultiCategoricalDistribution**(*action_dims: List[int]*)

MultiCategorical distribution for multi discrete actions.

Parameters **action_dims** – (List[int]) List of sizes of discrete action spaces

actions_from_params(*action_logits: torch.Tensor*, *deterministic: bool = False*) → torch.Tensor
Returns samples from the probability distribution given its parameters.

Returns (th.Tensor) actions

entropy() → torch.Tensor
Returns Shannon's entropy of the probability

Returns (Optional[th.Tensor]) the entropy, or None if no analytical form is known

log_prob(*actions: torch.Tensor*) → torch.Tensor
Returns the log likelihood

Parameters **x** – (th.Tensor) the taken action

Returns (th.Tensor) The log likelihood of the distribution

log_prob_from_params(*action_logits: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]
Returns samples and the associated log probabilities from the probability distribution given its parameters.

Returns (th.Tuple[th.Tensor, th.Tensor]) actions and log prob

mode() → torch.Tensor
Returns the most likely action (deterministic output) from the probability distribution

Returns (th.Tensor) the stochastic action

proba_distribution(*action_logits: torch.Tensor*) → *stable_baselines3.common.distributions.MultiCategoricalDistributio*
Set parameters of the distribution.

Returns (Distribution) self

proba_distribution_net(*latent_dim: int*) → torch.nn.modules.module.Module
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 (nn.Linear)

sample() → torch.Tensor
Returns a sample from the probability distribution

Returns (th.Tensor) the stochastic action

class stable_baselines3.common.distributions.**SquashedDiagGaussianDistribution**(*action_dim: int*, *epsilon: float = 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() → Optional[torch.Tensor]
Returns Shannon's entropy of the probability

Returns (Optional[th.Tensor]) the entropy, or None if no analytical form is known

log_prob(*actions: torch.Tensor*, *gaussian_actions: Optional[torch.Tensor] = None*) → torch.Tensor
Get the log probabilities of actions according to the distribution. Note that you must first call the `proba_distribution()` method.

Parameters **actions** – (th.Tensor)

Returns (th.Tensor)

log_prob_from_params(*mean_actions: torch.Tensor*, *log_std: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]
Compute the log probability of taking an action given the distribution parameters.

Parameters

- **mean_actions** – (th.Tensor)
- **log_std** – (th.Tensor)

Returns (Tuple[th.Tensor, th.Tensor])

mode() → torch.Tensor
Returns the most likely action (deterministic output) from the probability distribution

Returns (th.Tensor) the stochastic action

proba_distribution(*mean_actions: torch.Tensor*, *log_std: torch.Tensor*) → *stable_baselines3.common.distributions.SquashedDiagGaussianDistribution*
Create the distribution given its parameters (mean, std)

Parameters

- **mean_actions** – (th.Tensor)
- **log_std** – (th.Tensor)

Returns (DiagGaussianDistribution)

sample() → torch.Tensor
Returns a sample from the probability distribution

Returns (th.Tensor) the stochastic action

class stable_baselines3.common.distributions.**StateDependentNoiseDistribution**(*action_dim: int*, *full_std: bool = True*, *use_expln: bool = False*, *squash_output: bool = False*, *learn_features: bool = False*, *epsilon: float = 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: torch.Tensor*, *log_std: torch.Tensor*, *latent_sde: torch.Tensor*, *deterministic: bool = False*) → torch.Tensor

Returns samples from the probability distribution given its parameters.

Returns (th.Tensor) actions

entropy() → Optional[torch.Tensor]

Returns Shannon's entropy of the probability

Returns (Optional[th.Tensor]) the entropy, or None if no analytical form is known

get_std(*log_std: torch.Tensor*) → torch.Tensor

Get the standard deviation from the learned parameter (log of it by default). This ensures that the std is

positive.

Parameters **log_std** – (th.Tensor)

Returns (th.Tensor)

log_prob (*actions: torch.Tensor*) → torch.Tensor
Returns the log likelihood

Parameters **x** – (th.Tensor) the taken action

Returns (th.Tensor) The log likelihood of the distribution

log_prob_from_params (*mean_actions: torch.Tensor*, *log_std: torch.Tensor*, *latent_sde: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]
Returns samples and the associated log probabilities from the probability distribution given its parameters.

Returns (th.Tuple[th.Tensor, th.Tensor]) actions and log prob

mode () → torch.Tensor
Returns the most likely action (deterministic output) from the probability distribution

Returns (th.Tensor) the stochastic action

proba_distribution (*mean_actions: torch.Tensor*, *log_std: torch.Tensor*, *latent_sde: torch.Tensor*) → *stable_baselines3.common.distributions.StateDependentNoiseDistribution*
Create the distribution given its parameters (mean, std)

Parameters

- **mean_actions** – (th.Tensor)
- **log_std** – (th.Tensor)
- **latent_sde** – (th.Tensor)

Returns (StateDependentNoiseDistribution)

proba_distribution_net (*latent_dim: int*, *log_std_init: float = - 2.0*, *latent_sde_dim: Optional[int] = None*) → Tuple[torch.nn.modules.module.Module, torch.nn.parameter.Parameter]
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** – (Optional[int]) Dimension of the last layer of the feature extractor for gSDE. By default, it is shared with the policy network.

Returns (nn.Linear, nn.Parameter)

sample () → torch.Tensor
Returns a sample from the probability distribution

Returns (th.Tensor) the stochastic action

sample_weights (*log_std: torch.Tensor*, *batch_size: int = 1*) → None
Sample weights for the noise exploration matrix, using a centered Gaussian distribution.

Parameters

- **log_std** – (th.Tensor)

- **batch_size** – (int)

class stable_baselines3.common.distributions.**TanhBijector**(*epsilon: float = 1e-06*)

Bijective transformation of a probability distribution using a squashing function (tanh) TODO: use Pyro instead (https://pyro.ai/)

Parameters **epsilon** – (float) small value to avoid NaN due to numerical imprecision.

static atanh(*x: torch.Tensor*) → torch.Tensor

Inverse of Tanh

Taken from pyro: https://github.com/pyro-ppl/pyro 0.5 * torch.log((1 + x) / (1 - x))

static inverse(*y: torch.Tensor*) → torch.Tensor

Inverse tanh.

Parameters **y** – (th.Tensor)

Returns (th.Tensor)

stable_baselines3.common.distributions.**make_proba_distribution**(*action_space: gym.spaces.space.Space*, *use_sde: bool = False*, *dist_kwargs: Optional[Dict[str, Any]] = None*) → *stable_baselines3.common.distributions.Dis*

Return an instance of Distribution for the correct type of action space

Parameters

- **action_space** – (gym.spaces.Space) the input action space
- **use_sde** – (bool) Force the use of StateDependentNoiseDistribution instead of DiagGaussianDistribution
- **dist_kwargs** – (Optional[Dict[str, Any]]) Keyword arguments to pass to the probability distribution

Returns (Distribution) the appropriate Distribution object

stable_baselines3.common.distributions.**sum_independent_dims**(*tensor: torch.Tensor*) → torch.Tensor

Continuous actions are usually considered to be independent, so we can sum components of the `log_prob` or the entropy.

Parameters **tensor** – (th.Tensor) shape: (n_batch, n_actions) or (n_batch,)

Returns (th.Tensor) shape: (n_batch,)

1.26 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*)

Runs policy for `n_eval_episodes` episodes and returns average reward. This is made to work only with one env.

Parameters

- **model** – (BaseAlgorithm) The RL agent you want to evaluate.
- **env** – (gym.Env or VecEnv) The gym environment. In the case of a `VecEnv` this must contain only one 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) callback function to do additional checks, called after each step.
- **reward_threshold** – (float) 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 reward per episode will be returned instead of the mean.

Returns (float, float) Mean reward per episode, std of reward per episode returns ([float], [int]) when `return_episode_rewards` is True

1.27 Gym Environment Checker

stable_baselines3.common.env_checker.**check_env**(*env: gym.core.Env*, *warn: bool = True*, *skip_render_check: bool = True*) → None

Check that an environment follows Gym API. This is particularly useful when using a custom environment. Please take a look at https://github.com/openai/gym/blob/master/gym/core.py for more information about the API.

It also optionally check that the environment is compatible with Stable-Baselines.

Parameters

- **env** – (gym.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)

1.28 Monitor Wrapper

class stable_baselines3.common.monitor.**Monitor**(*env: gym.core.Env*, *filename: Optional[str] = None*, *allow_early_resets: bool = True*, *reset_keywords: Tuple[str, ...] = ()*, *info_keywords: Tuple[str, ...] = ()*)

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

Parameters

- **env** – (gym.Env) The environment
- **filename** – (Optional[str]) 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()

close()

Closes the environment

get_episode_lengths() → List[int]

Returns the number of timesteps of all the episodes

Returns ([int])

get_episode_rewards() → List[float]

Returns the rewards of all the episodes

Returns ([float])

get_episode_times() → List[float]

Returns the runtime in seconds of all the episodes

Returns ([float])

get_total_steps() → int

Returns the total number of timesteps

Returns (int)

reset(***kwargs*) → numpy.ndarray

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 (np.ndarray) the first observation of the environment

step(*action: numpy.ndarray*) → Tuple[numpy.ndarray, float, bool, Dict[Any, Any]]

Step the environment with the given action

Parameters **action** – (np.ndarray) the action

Returns (Tuple[np.ndarray, float, bool, Dict[Any, Any]]) observation, reward, done, information

stable_baselines3.common.monitor.**get_monitor_files**(*path: str*) → List[str]

get all the monitor files in the given path

Parameters **path** – (str) the logging folder

Returns ([str]) the log files

stable_baselines3.common.monitor.**load_results**(*path: str*) → pandas.core.frame.DataFrame

Load all Monitor logs from a given directory path matching `*monitor.csv`

Parameters **path** – (str) the directory path containing the log file(s)

Returns (pandas.DataFrame) the logged data

1.29 Logger

class stable_baselines3.common.logger.**CSVOutputFormat**(*filename: str*)

close() → None
closes the file

write(*key_values: Dict[str, Any]*, *key_excluded: Dict[str, Union[str, Tuple[str, ...]]]*, *step: int = 0*) → None
Write a dictionary to file

Parameters

- **key_values** – (dict)
- **key_excluded** – (dict)
- **step** – (int)

class stable_baselines3.common.logger.**HumanOutputFormat**(*filename_or_file: Union[str, TextIO]*)

close() → None
closes the file

write(*key_values: Dict*, *key_excluded: Dict*, *step: int = 0*) → None
Write a dictionary to file

Parameters

- **key_values** – (dict)
- **key_excluded** – (dict)
- **step** – (int)

write_sequence(*sequence: List*) → None
write_sequence an array to file

Parameters **sequence** – (list)

class stable_baselines3.common.logger.**JSONOutputFormat**(*filename: str*)

close() → None
closes the file

write(*key_values: Dict[str, Any]*, *key_excluded: Dict[str, Union[str, Tuple[str, ...]]]*, *step: int = 0*) → None
Write a dictionary to file

Parameters

- **key_values** – (dict)
- **key_excluded** – (dict)
- **step** – (int)

class stable_baselines3.common.logger.**KVWriter**
Key Value writer

close() → None
Close owned resources

write(*key_values: Dict[str, Any]*, *key_excluded: Dict[str, Union[str, Tuple[str, ...]]]*, *step: int = 0*) → None
Write a dictionary to file

Parameters

- **key_values** – (dict)
- **key_excluded** – (dict)
- **step** – (int)

class stable_baselines3.common.logger.**SeqWriter**
sequence writer

write_sequence(*sequence: List*)
write_sequence an array to file

Parameters **sequence** – (list)

class stable_baselines3.common.logger.**TensorBoardOutputFormat**(*folder: str*)

close() → None
closes the file

write(*key_values: Dict[str, Any]*, *key_excluded: Dict[str, Union[str, Tuple[str, ...]]]*, *step: int = 0*) → None
Write a dictionary to file

Parameters

- **key_values** – (dict)
- **key_excluded** – (dict)
- **step** – (int)

stable_baselines3.common.logger.**configure**(*folder: Optional[str] = None*, *format_strings: Optional[List[str]] = None*) → None
configure the current logger

Parameters

- **folder** – (Optional[str]) the save location (if None, $SB3_LOGDIR, if still None, tempdir/baselines-[date & time])
- **format_strings** – (Optional[List[str]]) the output logging format (if None, $SB3_LOG_FORMAT, if still None, ['stdout', 'log', 'csv'])

stable_baselines3.common.logger.**debug**(**args*) → None
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** – (list) log the arguments

stable_baselines3.common.logger.**dump**(*step: int = 0*) → None
Write all of the diagnostics from the current iteration

stable_baselines3.common.logger.**dump_tabular**(*step: int = 0*) → None
Write all of the diagnostics from the current iteration

stable_baselines3.common.logger.**error**(**args*) → None
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** – (list) log the arguments

stable_baselines3.common.logger.**get_dir**() → str
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 (str) the logging directory

stable_baselines3.common.logger.**get_level**() → int
Get logging threshold on current logger. :return: (int) the logging level (can be DEBUG=10, INFO=20, WARN=30, ERROR=40, DISABLED=50)

stable_baselines3.common.logger.**get_log_dict**() → Dict
get the key values logs

Returns (dict) the logged values

stable_baselines3.common.logger.**info**(**args*) → None
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** – (list) log the arguments

stable_baselines3.common.logger.**log**(**args*, *level: int = 20*) → None
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** – (list) log the arguments
- **level** – (int) the logging level (can be DEBUG=10, INFO=20, WARN=30, ERROR=40, DISABLED=50)

stable_baselines3.common.logger.**make_output_format**(*_format: str*, *log_dir: str*, *log_suffix: str = ''*) → *stable_baselines3.common.logger.KVWriter*
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 (KVWriter) the logger

stable_baselines3.common.logger.**read_csv**(*filename: str*) → pandas.core.frame.DataFrame
read a csv file using pandas

Parameters **filename** – (str) the file path to read

Returns (pandas.DataFrame) the data in the csv

stable_baselines3.common.logger.**read_json**(*filename: str*) → pandas.core.frame.DataFrame
read a json file using pandas

Parameters **filename** – (str) the file path to read

Returns (pandas.DataFrame) the data in the json

stable_baselines3.common.logger.**record**(*key: str*, *value: Any*, *exclude: Optional[Union[str, Tuple[str, ...]]] = None*) → 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** – (Any) save to log this key
- **value** – (Any) save to log this value
- **exclude** – (str or tuple) outputs to be excluded

stable_baselines3.common.logger.**record_dict**(*key_values: Dict[str, Any]*) → None
Log a dictionary of key-value pairs.

Parameters **key_values** – (dict) the list of keys and values to save to log

stable_baselines3.common.logger.**record_mean**(*key: str*, *value: Union[int, float]*, *exclude: Optional[Union[str, Tuple[str, ...]]] = None*) → None
The same as record(), but if called many times, values averaged.

Parameters

- **key** – (Any) save to log this key
- **value** – (Number) save to log this value
- **exclude** – (str or tuple) outputs to be excluded

stable_baselines3.common.logger.**record_tabular**(*key: str*, *value: Any*, *exclude: Optional[Union[str, Tuple[str, ...]]] = None*) → 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** – (Any) save to log this key
- **value** – (Any) save to log this value
- **exclude** – (str or tuple) outputs to be excluded

stable_baselines3.common.logger.**reset**() → None
reset the current logger

stable_baselines3.common.logger.**set_level**(*level: int*) → None
Set logging threshold on current logger.

Parameters **level** – (int) the logging level (can be DEBUG=10, INFO=20, WARN=30, ERROR=40, DISABLED=50)

stable_baselines3.common.logger.**warn**(**args*) → None
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** – (list) log the arguments

1.30 Action Noise

class stable_baselines3.common.noise.**ActionNoise**
The action noise base class

reset() → None
call end of episode reset for the noise

class stable_baselines3.common.noise.**NormalActionNoise**(*mean: numpy.ndarray*, *sigma: numpy.ndarray*)
A Gaussian action noise

Parameters

- **mean** – (np.ndarray) the mean value of the noise
- **sigma** – (np.ndarray) the scale of the noise (std here)

class stable_baselines3.common.noise.**OrnsteinUhlenbeckActionNoise**(*mean: numpy.ndarray*, *sigma: numpy.ndarray*, *theta: float = 0.15*, *dt: float = 0.01*, *initial_noise: Optional[numpy.ndarray] = None*)
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** – (np.ndarray) the mean of the noise
- **sigma** – (np.ndarray) the scale of the noise
- **theta** – (float) the rate of mean reversion
- **dt** – (float) the timestep for the noise
- **initial_noise** – (Optional[np.ndarray]) the initial value for the noise output, (if None: 0)

reset() → None
reset the Ornstein Uhlenbeck noise, to the initial position

class stable_baselines3.common.noise.**VectorizedActionNoise**(*base_noise: stable_baselines3.common.noise.ActionNoise*, *n_envs: int*)
A Vectorized action noise for parallel environments.

Parameters

- **base_noise** – ActionNoise The noise generator to use
- **n_envs** – (int) The number of parallel environments

reset (*indices: Optional[Iterable[int]] = None*) → None
Reset all the noise processes, or those listed in indices

Parameters **indices** – Optional[Iterable[int]] The indices to reset. Default: None. If the parameter is None, then all processes are reset to their initial position.

1.31 Utils

stable_baselines3.common.utils.**check_for_correct_spaces** (*env: Union[gym.core.Env, stable_baselines3.common.vec_env.base_vec_env.VecE observation_space: gym.spaces.space.Space, action_space: gym.spaces.space.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** – (GymEnv) Environment to check for valid spaces
- **observation_space** – (gym.spaces.Space) Observation space to check against
- **action_space** – (gym.spaces.Space) Action space to check against

stable_baselines3.common.utils.**configure_logger** (*verbose: int = 0, tensorboard_log: Optional[str] = None, tb_log_name: str = '', reset_num_timesteps: bool = True*) → None
Configure the logger's outputs.

Parameters

- **verbose** – (int) the verbosity level: 0 no output, 1 info, 2 debug
- **tensorboard_log** – (str) the log location for tensorboard (if None, no logging)
- **tb_log_name** – (str) tensorboard log

stable_baselines3.common.utils.**constant_fn** (*val: float*) → Callable
Create a function that returns a constant It is useful for learning rate schedule (to avoid code duplication)

Parameters **val** – (float)

Returns (Callable)

stable_baselines3.common.utils.**explained_variance** (*y_pred: numpy.ndarray, y_true: numpy.ndarray*) → numpy.ndarray
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** – (np.ndarray) the prediction

- **y_true** – (np.ndarray) the expected value

Returns (float) explained variance of ypred and y

stable_baselines3.common.utils.**get_device**(*device: Union[torch.device, str] = 'auto'*) → torch.device

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** – (Union[str, th.device]) One for 'auto', 'cuda', 'cpu'

Returns (th.device)

stable_baselines3.common.utils.**get_latest_run_id**(*log_path: Optional[str] = None*, *log_name: str = ''*) → int

Returns the latest run number for the given log name and log path, by finding the greatest number in the directories.

Returns (int) latest run number

stable_baselines3.common.utils.**get_linear_fn**(*start: float*, *end: float*, *end_fraction: float*) → Callable

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 (float) value to start with if `progress_remaining` = 1

Params end (float) value to end with if `progress_remaining` = 0

Params 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.

Returns (Callable)

stable_baselines3.common.utils.**get_schedule_fn**(*value_schedule: Union[Callable, float]*) → Callable

Transform (if needed) learning rate and clip range (for PPO) to callable.

Parameters **value_schedule** – (callable or float)

Returns (function)

stable_baselines3.common.utils.**is_vectorized_observation**(*observation: numpy.ndarray*, *observation_space: gym.spaces.space.Space*) → bool

For every observation type, detects and validates the shape, then returns whether or not the observation is vectorized.

Parameters

- **observation** – (np.ndarray) the input observation to validate
- **observation_space** – (gym.spaces) the observation space

Returns (bool) whether the given observation is vectorized or not

stable_baselines3.common.utils.**polyak_update**(*params: Iterable[torch.nn.parameter.Parameter]*, *target_params: Iterable[torch.nn.parameter.Parameter]*, *tau: float*) → None

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[th.nn.Parameter]) parameters to use to update the target params
- **target_params** – (Iterable[th.nn.Parameter]) parameters to update
- **tau** – (float) the soft update coefficient ("Polyak update", between 0 and 1)

stable_baselines3.common.utils.**safe_mean**(*arr: Union[numpy.ndarray, list, collections.deque]*) → numpy.ndarray

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** –

Returns

stable_baselines3.common.utils.**set_random_seed**(*seed: int*, *using_cuda: bool = False*) → None

Seed the different random generators :param seed: (int) :param using_cuda: (bool)

stable_baselines3.common.utils.**update_learning_rate**(*optimizer: torch.optim.optimizer.Optimizer*, *learning_rate: float*) → None

Update the learning rate for a given optimizer. Useful when doing linear schedule.

Parameters

- **optimizer** – (th.optim.Optimizer)
- **learning_rate** – (float)

1.32 Changelog

1.32.1 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 (@Partially-Typed)
- 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 feature 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)

1.32.2 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 feature 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

1.32.3 Pre-Release 0.6.0 (2020-06-01)

Tensorboard support, refactored logger

Breaking Changes:

- 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

1.32.4 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:

1.32.5 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

1.32.6 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

1.32.7 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

1.32.8 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

1.32.9 Maintainers

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

1.32.10 Contributors:

In random order...

Thanks to the maintainers of V2: @hill-a @enerijunior @AdamGleave @Miffyli

And all the contributors: @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 @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

1.33 Projects

This is a list of projects using stable-baselines3. Please tell us, if you want your project to appear on this page ;)

1.33.1 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

1.33.2 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

CHAPTER

TWO

CITING STABLE BASELINES3

To cite this project in publications:

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

CHAPTER

THREE

INDICES AND TABLES

- genindex
- search
- modindex

PYTHON MODULE INDEX

S

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W