Stable Baselines3 Documentation

Release 0.11.0a5

Stable Baselines3 Contributors

Jan 11, 2021
## 1 Main Features

1.1 Installation ......................................................... 3
1.2 Getting Started ................................................... 5
1.3 Reinforcement Learning Tips and Tricks ........................... 6
1.4 Reinforcement Learning Resources ................................ 11
1.5 RL Algorithms ..................................................... 11
1.6 Examples ........................................................... 12
1.7 Vectorized Environments ........................................... 23
1.8 Using Custom Environments ....................................... 33
1.9 Custom Policy Network ............................................ 35
1.10 Callbacks ........................................................... 39
1.11 Tensorboard Integration ............................................ 47
1.12 RL Baselines3 Zoo ................................................ 52
1.13 SB3 Contrib ........................................................ 54
1.14 Imitation Learning ................................................ 55
1.15 Migrating from Stable-Baselines ................................. 56
1.16 Dealing with NaNs and infs ..................................... 60
1.17 Developer Guide ................................................... 63
1.18 On saving and loading ............................................. 64
1.19 Exporting models ................................................ 65
1.20 Base RL Class ...................................................... 66
1.21 A2C ..................................................................... 76
1.22 DDPG .................................................................... 85
1.23 DQN ...................................................................... 93
1.24 HER ..................................................................... 101
1.25 PPO ..................................................................... 110
1.26 SAC ...................................................................... 119
1.27 TD3 ..................................................................... 128
1.28 Atari Wrappers ....................................................... 136
1.29 Environments Utils .................................................. 139
1.30 Probability Distributions .......................................... 141
1.31 Evaluation Helper ................................................... 150
1.32 Gym Environment Checker ........................................ 151
1.33 Monitor Wrapper .................................................... 151
1.34 Logger ................................................................... 153
1.35 Action Noise .......................................................... 158
1.36 Utils ..................................................................... 159
1.37 Changelog ............................................................. 162
1.38 Projects ............................................................... 175
Stable Baselines3 (SB3) is a set of reliable implementations of reinforcement learning algorithms in PyTorch. It is the next major version of Stable Baselines.

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


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
• The performance of each algorithm was tested (see Results section in their respective page)

1.1 Installation

1.1.1 Prerequisites

Stable-Baselines3 requires python 3.6+.

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 Baselines3 Documentation, Release 0.11.0a5

Stable Release

To install Stable Baselines3 with pip, execute:

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

This includes an optional dependencies like Tensorboard, OpenCV or \texttt{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.

```
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 \texttt{nvidia-docker}):

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

CPU only:

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

Build the Docker Images

Build GPU image (with \texttt{nvidia-docker}):

```
make docker-gpu
```

Build CPU image:

```
make docker-cpu
```

Chapter 1. Main Features
Note: if you are using a proxy, you need to pass extra params during build and do some tweaks:

```
```

### Run the images (CPU/GPU)

#### Run the nvidia-docker GPU image

```
```

Or, with the shell file:

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

#### Run the docker CPU image

```
```

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-baselines3`), 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:

```python
import gym

from stable_baselines3 import A2C

test = 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)
    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:

```python
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 (remember to check wrappers!)
4. For better performance, increase the training budget

Like any other subject, if you want to work with RL, you should first read about it (we have a dedicated resource page to get you started) to understand what you are using. We also recommend 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 Mimir 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?**

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

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

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

As some 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.

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. We notably provide QR-DQN in our contrib repo. DQN is usually slower to train (regarding wall clock time) but is the most sample efficient (because of its replay buffer).

**Discrete Actions - Multiprocessed**

You should give a try to PPO or A2C.

**Continuous Actions**

**Continuous Actions - Single Process**

Current State Of The Art (SOTA) algorithms are SAC, TD3 and TQC (available in our contrib repo). 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*
Goal Environment

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

Note: The number of workers is an important hyperparameters for experiments with HER

1.3.3 Tips and Tricks when creating a custom environment

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

Some basic advice:

- always normalize your observation space 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:

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

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

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

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

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Box</th>
<th>Discrete</th>
<th>MultiDiscrete</th>
<th>MultiBinary</th>
<th>Multi Processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2C</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DDPG</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DQN</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HER</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PPO</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SAC</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TD3</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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: More algorithms (like QR-DQN or TQC) are implemented in our contrib repo.

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
- Hindsight Experience Replay
- Advanced Saving and Loading

1.6.2 Basic Usage: Training, Saving, Loading

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

Fig. 1: Lunar Lander Environment

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

```python
import gym
from stable_baselines3 import DQN
from stable_baselines3.common.evaluation import evaluate_policy
```

(continues on next page)
# 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

```python
import gym
import numpy as np

from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import SubprocVecEnv
from stable_baselines3.common.env_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 initial seed for RNG
    :param rank: (int) index of the subprocess
    """
    def _init():
        env = gym.make(env_id)
```

(continues on next page)
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.

```python
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").
    """
    def __init__(self, check_freq=1, log_dir="logs"):
        self.check_freq = check_freq
        self.log_dir = log_dir
        self.save_path = os.path.join(log_dir, "trained_model")

    def _init(self):
        # Initialize callback

    def _on_step(self):
        if self.n_calls % self.check_freq == 0:
            # Check if the training reward is the best
            if self.mean_reward > self.best_reward:
                self.best_reward = self.mean_reward
                self.save_path = os.path.join(self.log_dir, "saved_model")
                self.model.save(self.save_path)

        return True
```

(continues on next page)
Stable Baselines3 Documentation, Release 0.11.0a5

:math:`\text{param check_freq: (int)}`
:math:`\text{param log_dir: (str) Path to the folder where the model will be saved. It must contains the file created by the ``Monitor`` wrapper.}`
:math:`\text{param verbose: (int)}`

```python
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(f'Num timesteps: {x[-1]}.
            print(f'Best mean reward: {mean_reward:.2f} - Last mean reward per episode: {mean_reward:.2f}

        # New best model, you could save the agent here
        if mean_reward > self.best_mean_reward:
            self.best_mean_reward = mean_reward
            # Example for saving best model
            if self.verbose > 0:
                print(f'Saving new best model to {self.save_path}')
            self.model.save(self.save_path)

    return True
```

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

# 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

(continues on next page)
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

![Trained A2C agent on Breakout](image1)

![Pong Environment](image2)

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.

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

# There already exists an environment generator
# that will make and wrap atari environments correctly.
# Here we are also multi-worker training (n_envs=4 => 4 environments)
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

dummy_env = DummyVecEnv([lambda: gym.make("HalfCheetahBulletEnv-v0")])
# Automatically normalize the input features and reward
env = VecNormalize(dummy_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")])
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 Hindsight Experience Replay (HER)

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

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

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

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

import gym
import highway_env
import numpy as np

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

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

# Create 4 artificial transitions per real transition
n_sampled_goal = 4

# SAC hyperparams:
model = HER(
    "MlpPolicy",
    env,
    SAC,
    n_sampled_goal=n_sampled_goal,
    goal_selection_strategy="future",
    # IMPORTANT: because the env is not wrapped with a TimeLimit wrapper
    # we have to manually specify the max number of steps per episode
    max_episode_length=100,
    verbose=1,
    buffer_size=int(1e6),
    learning_rate=1e-3,
    gamma=0.95,
    batch_size=256,
    online_sampling=True,
    policy_kwargs=dict(net_arch=[256, 256, 256]),
)

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

# Load saved model
model = HER.load("her_sac_highway", env=env)

obs = env.reset()

# Evaluate the agent
episode_reward = 0
for _ in range(100):
    action, _ = model.predict(obs, deterministic=True)
    obs, reward, done, info = env.step(action)
    env.render()
    episode_reward += reward
    if done or info.get("is_success", False):
        print("Reward:", episode_reward, "Success?", info.get("is_success", False))
        episode_reward = 0.0
        obs = env.reset()

1.6.8 Learning Rate Schedule

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

The RL Zoo already includes linear and constant schedules.

```python
from typing import Callable
from stable_baselines3 import PPO
```
def linear_schedule(initial_value: float) -> Callable[[float], float]:
    """
    Linear learning rate schedule.
    :param initial_value: Initial learning rate.
    :return: schedule that computes current learning rate depending on remaining progress
    """
    def func(progress_remaining: float) -> float:
        """
        Progress will decrease from 1 (beginning) to 0.
        :param progress_remaining:
        :return: current learning rate
        """
        return progress_remaining * initial_value
    return func

# Initial learning rate of 0.001
model = PPO("MlpPolicy", "CartPole-v1", learning_rate=linear_schedule(0.001),
            verbose=1)
model.learn(total_timesteps=20000)
# By default, `reset_num_timesteps` is True, in which case the learning rate schedule resets.
# progress_remaining = 1.0 - (num_timesteps / total_timesteps)
model.learn(total_timesteps=10000, reset_num_timesteps=True)

1.6.9 Advanced Saving and Loading

In this example, we show how to use some advanced features of Stable-Baselines3 (SB3): how to easily create a test environment to evaluate an agent periodically, use a policy independently from a model (and how to save it, load it) and save/load a replay buffer.

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

Stable-Baselines3 automatic creation of an environment for evaluation. For that, you only need to specify create_eval_env=True when passing the Gym ID of the environment while creating the agent. Behind the scene, SB3 uses an EvalCallback.

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

# Create the model, the training environment
# and the test environment (for evaluation)
model = SAC("MlpPolicy", "Pendulum-v0", verbose=1,
            learning_rate=1e-3, create_eval_env=True)
# Evaluate the model every 1000 steps on 5 test episodes
# and save the evaluation to the "logs/" folder
model.learn(6000, eval_freq=1000, n_eval_episodes=5, eval_log_path="./logs/")

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

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

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

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

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

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

# Retrieve the environment
env = model.get_env()

# Evaluate the policy
mean_reward, std_reward = evaluate_policy(policy, env, n_eval_episodes=10, deterministic=True)
print(f"mean_reward={mean_reward:.2f} +/- {std_reward}")

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

# Evaluate the loaded policy
mean_reward, std_reward = evaluate_policy(saved_policy, env, n_eval_episodes=10, deterministic=True)
print(f"mean_reward={mean_reward:.2f} +/- {std_reward}")
1.6.10 Accessing and modifying model parameters

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

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

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

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

```python
from typing import Dict
import gym
import numpy as np
import torch as th
from stable_baselines3 import A2C
from stable_baselines3.common.evaluation import evaluate_policy

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

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

# Use traditional actor-critic policy gradient updates to
# find good initial parameters
model.learn(total_timesteps=10000)

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

# population size of 50 individuals
pop_size = 50
```

(continues on next page)
# Keep top 10%

```python
n_elite = pop_size // 10
```

# Retrieve the environment

```python
env = model.get_env()
```

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

## 1.6.11 Record a Video

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

**Note:** It requires ffmpeg or avconv to be installed on the machine.

```python
import gym
from stable_baselines3.common.vec_env import VecVideoRecorder, DummyVecEnv
```

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

```python
env = DummyVecEnv([lambda: gym.make(env_id)])
obs = env.reset()
```

```python
# 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))
```

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

1.6.12 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

```python
import imageio
import numpy as np
from stable_baselines3 import A2C

model = A2C("MlpPolicy", "LunarLander-v2").learn(100000)
images = []
obs = model.env.reset()
img = model.env.render(mode='rgb_array')
for i in range(350):
    images.append(img)
    action, _ = model.predict(obs)
    obs, _, _, _ = model.env.step(action)
    img = model.env.render(mode='rgb_array')
imageio.mimsave('lander_a2c.gif', [np.array(img) for i, img in enumerate(images) if i % 2 == 0], fps=29)
```

### 1.7 Vectorized Environments

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Box</th>
<th>Discrete</th>
<th>Dict</th>
<th>Tuple</th>
<th>Multi Processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>DummyVecEnv</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SubprocVecEnv</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

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 (Space) – the observation space

• action_space (Space) – the action space

abstract close()

Clean up the environment’s resources.

Return type None

abstract env_is_wrapped(wrapper_class, indices=None)

Check if environments are wrapped with a given wrapper.

Parameters

• method_name – The name of the environment method to invoke.

• indices (Union[None, int, Iterable[int]]) – Indices of envs whose method to call

• method_args – Any positional arguments to provide in the call

• method_kwargs – Any keyword arguments to provide in the call

Return type List[bool]

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

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 (Union[None, int, Iterable[int]]) – Indices of envs whose method to call

• method_args – Any positional arguments to provide in the call

• method_kwargs – Any keyword arguments to provide in the call

Return type List[Any]
Returns 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**(Union[None, int, Iterable[int]]) – Indices of envs to get attribute from

Return type List[None]

Returns List of values of ‘attr_name’ in all environments

**get_images**( )

Return RGB images from each environment

Return type Sequence[ndarray]

**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

Return type Optional[None]

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

**render**(mode=’human’)

Gym environment rendering

Parameters **mode**(str) – the rendering type

Return type Optional[ndarray]

**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.

Return type Union[ndarray, Dict[str, ndarray], Tuple[ndarray, ...]]

Returns observation

**abstract 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.

Return type List[Optional[int]]

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

**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** *(Any)* – Value to assign to **attr_name**
• **indices** *(Union[None, int, Iterable[int]])* – Indices of envs to assign value

**Return type** None

**Returns**

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

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

**Return type** Tuple[Union[ndarray, Dict[str, ndarray], Tuple[ndarray, ...]],
ndarray, ndarray, List[Dict]]

**Returns** 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.

**Return type** None

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

**Return type** Tuple[Union[ndarray, Dict[str, ndarray], Tuple[ndarray, ...]],
ndarray, ndarray, List[Dict]]

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

**close** ()
Clean up the environment’s resources.

**Return type** None

**env_is_wrapped** *(wrapper_class, indices=None)*
Check if worker environments are wrapped with a given wrapper

**Return type** List[bool]

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

**Return type** List[Any]

**get_attr** *(attr_name, indices=None)*
Return attribute from vectorized environment (see base class).
Return type List[Optional[int]]

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

Return type None

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

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

Return type None

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

Return type Tuple[Union[ndarray, Dict[str, ndarray], Tuple[ndarray, ...]], ndarray, ndarray, List[Dict]]

Returns 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 Tensor-
Flow sessions or other non thread-safe libraries are used in the parent (see issue #217). However, compared
to ‘fork’ they incur a small start-up cost and have restrictions on global variables. With those methods,
users must wrap the code in an if __name__ == "__main__": block. For more information, see the
multiprocessing documentation.

Parameters

• env_fns (List[Callable[[], Env]]) – Environments to run in subprocesses
• start_method (Optional[str]) – method used to start the subprocesses. Must be
  one of the methods returned by multiprocessing.get_all_start_methods(). Defaults to ‘fork-
  server’ on available platforms, and ‘spawn’ otherwise.

close()

Clean up the environment’s resources.

Return type None

env_is_wrapped(wrapper_class, indices=None)

Check if worker environments are wrapped with a given wrapper

Return type List[bool]

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

Call instance methods of vectorized environments.

Return type List[Union]

get_attr(attr_name, indices=None)

Return attribute from vectorized environment (see base class).

Return type List[Union]

get_images()

Return RGB images from each environment

Return type Sequence[ndarray]

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.

Return type Union[ndarray, Dict[str, ndarray], Tuple[ndarray, ...]]

Returns 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.

Return type List[Optional[int]]

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

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

Return type None

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

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

Return type None

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

Return type Tuple[Union[ndarray, Dict[str, ndarray], Tuple[ndarray, ...]],
ndarray, ndarray, List[Dict]]

Returns observation, reward, done, information

1.7.4 Wrappers

VecFrameStack

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

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

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

Parameters

- `venv` (VecEnv) – the vectorized environment to wrap
- `n_stack` (int) – Number of frames to stack
- `channels_order` (Optional[str]) – If “first”, stack on first image dimension. If “last”, stack on last dimension. If None, automatically detect channel to stack over in case of image observation or default to “last” (default).

`close` ()
Clean up the environment’s resources.

Return type None

`reset` ()
Reset all environments

Return type ndarray

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

Return type Tuple[ndarray, ndarray, ndarray, List[Dict[str, Any]]]
Returns 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.

Return type Union[ndarray, Dict[str, ndarray]]

get_original_reward()

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

Return type ndarray

static load(load_path, venv)

Loads a saved VecNormalize object.

Parameters

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

Return type VecNormalize

Returns

normalize_obs(obs)

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

Return type Union[ndarray, Dict[str, ndarray]]

normalize_reward(reward)

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

Return type ndarray
**reset()**
Reset all environments :rtype: Union[ndarray, Dict[str, ndarray]] :return: first observation of the episode

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

Parameters

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

Return type None

**set_venv(venv)**
Sets the vector environment to wrap to venv. Also sets attributes derived from this such as num_env.

Parameters

- **venv**(VecEnv) –

Return type None

**step_wait()**
Apply sequence of actions to sequence of environments actions -> (observations, rewards, news) where ‘news’ is a boolean vector indicating whether each element is new.

Return type Tuple[Union[ndarray, Dict[str, ndarray], Tuple[ndarray, ...]], ndarray, ndarray, List[Dict]]

---

**VecVideoRecorder**

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

**close()**
Clean up the environment’s resources.

Return type None

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

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

Return type Union[ndarray, Dict[str, ndarray], Tuple[ndarray, ...]]
Returns observation

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

**Return type** Tuple[Union[ndarray, Dict[str, ndarray], Tuple[ndarray, ...]],
ndarray, ndarray, List[Dict]]

**Returns** 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.

**Return type** Union[ndarray, Dict[str, ndarray], Tuple[ndarray, ...]]

**Returns** 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.

**Return type** None

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

**Return type** Tuple[Union[ndarray, Dict[str, ndarray], Tuple[ndarray, ...]],
ndarray, ndarray, List[Dict]]

**Returns** observation, reward, done, information
VecTransposeImage

class stable_baselines3.common.vec_env.VecTransposeImage(venv)
    Re-order channels, from HxWxC to CxHxW. It is required for PyTorch convolution layers.

    Parameters venv (VecEnv) –

close()
    Clean up the environment’s resources.

    Return type None

reset()
    Reset all environments

    Return type ndarray

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

    Return type Tuple[Union[ndarray, Dict[str, ndarray], Tuple[ndarray, ...]],
                        ndarray, ndarray, List[Dict]]

    Returns observation, reward, done, information

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

    Parameters image (ndarray) –

    Return type ndarray

    Returns

static transpose_space(observation_space)
    Transpose an observation space (re-order channels).

    Parameters observation_space (Box) –

    Return type Box

    Returns

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] and np.uint8 as the observation is normalized (dividing by 255 to have values in [0, 1]) when using CNN policies. Images can be either channel-first or channel-last.

```python
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 (can be channel-first or channel-last):
    self.observation_space = spaces.Box(low=0, high=255,
                                          shape=(HEIGHT, WIDTH, N_CHANNELS), dtype=np.uint8)

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:

```python
# 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:

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

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

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).

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

1.9.1 Custom Policy Architecture

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

```python
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()
# 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.

1.9.2 Custom Feature Extractor

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

```python
import gym
import torch as th
import torch.nn as nn
from stable_baselines3 import PPO
from stable_baselines3.common.torch_layers import BaseFeaturesExtractor
```

(continues on next page)
class CustomCNN(BaseFeaturesExtractor):
    
    :param observation_space: (gym.Space)
    :param features_dim: (int) Number of features extracted. This corresponds to the number of unit for the last layer.
    
    def __init__(self, observation_space: gym.spaces.Box, features_dim: int = 256):
        super(CustomCNN, self).__init__(observation_space, features_dim)
        # We assume CxHxW images (channels first)
        # Re-ordering will be done by pre-preprocessing or wrapper
        n_input_channels = observation_space.shape[0]
        self.cnn = nn.Sequential(
            nn.Conv2d(n_input_channels, 32, kernel_size=8, stride=4, padding=0),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2, padding=0),
            nn.ReLU(),
            nn.Flatten(),
        )
        # Compute shape by doing one forward pass
        with th.no_grad():
            n_flatten = self.cnn(
                th.as_tensor(observation_space.sample()[None]).float()).shape[1]
        self.linear = nn.Sequential(nn.Linear(n_flatten, features_dim), nn.ReLU())

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

policy_kwargs = dict(
    features_extractor_class=CustomCNN,
    features_extractor_kwargs=dict(features_dim=128),
)

model = PPO("CnnPolicy", "BreakoutNoFrameskip-v4", policy_kwargs=policy_kwargs,
             verbose=1)
model.learn(1000)

1.9.3 On-Policy Algorithms

Shared Networks

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: \[
\langle \text{shared layers}\rangle, \text{dict(vf=}[\langle \text{non-shared value network layers}\rangle], \\
\text{pi=}[\langle \text{non-shared policy network layers}\rangle])\].
\]

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
```

Advanced Example

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

```python
from typing import Callable, Dict, List, Optional, Tuple, Type, Union
import gym
import torch as th
from torch import nn
from stable_baselines3 import PPO
from stable_baselines3.common.policies import ActorCriticPolicy

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

(continues on next page)
```python
:param feature_dim: dimension of the features extracted with the features_
→extractor (e.g. features from a CNN)
:param last_layer_dim_pi: (int) number of units for the last layer of the policy_
→network
:param last_layer_dim_vf: (int) number of units for the last layer of the value_
→network

```
activation_fn,
# Pass remaining arguments to base class
*args,
**kwargs,
)
# Disable orthogonal initialization
self.ortho_init = False

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

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

1.9.4 Off-Policy Algorithms

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

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

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

Note: Compared to their on-policy counterparts, no shared layers (other than the feature extractor) between the actor and the critic are allowed (to prevent issues with target networks).

```python
from stable_baselines3 import SAC

# Custom actor architecture with two layers of 64 units each
# Custom critic architecture with two layers of 400 and 300 units
policy_kwargs = dict(net_arch=dict(pi=[64, 64], qf=[400, 300]))
# Create the agent
model = SAC("MlpPolicy", "Pendulum-v0", policy_kwargs=policy_kwargs, verbose=1)
model.learn(5000)
```

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

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

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

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

(continues on next page)
.. 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.

```python
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:
```

(continues on next page)
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 (EventCallback, 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).

```python
from stable_baselines3 import SAC
from stable_baselines3.common.callbacks import CheckpointCallback
# Save a checkpoint every 1000 steps
checkpoint_callback = CheckpointCallback(save_freq=1000, save_path='./logs/',
                                        name_prefix='rl_model')
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.

```python
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.

```python
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')
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.

```python
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 \( n_{\text{steps}} \) timesteps.

**Note:** Because of the way PPO and TRPO work (they rely on MPI), \( n_{\text{steps}} \) is a lower bound between two events.

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

StopTrainingOnMaxEpisodes

Stop the training upon reaching the maximum number of episodes, regardless of the model’s \( \text{total\_timesteps} \) value. Also, it presumes that, for multiple environments, the desired behavior is that the agent trains on each env for \( \text{max\_episodes} \) and in total for \( \text{max\_episodes} \times n_{\text{envs}} \) episodes.

**Note:** For multiple environments, the agent will train for a total of \( \text{max\_episodes} \times n_{\text{envs}} \) episodes. However, it can’t be guaranteed that this training will occur for an exact number of \( \text{max\_episodes} \) per environment. Thus, there is an assumption that, on average, each environment ran for \( \text{max\_episodes} \).

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

callback_max_episodes = StopTrainingOnMaxEpisodes(max_episodes=5, verbose=1)
model = A2C('MlpPolicy', 'Pendulum-v0', verbose=1)
model.learn(int(2e4), callback=callback_max_episodes)
```

```python
class _stable_baselines3.common.callbacks.BaseCallback (verbose=0)
Base class for callback.

Parameters

**verbose** (int)–

**init\_callback** (*model*)-

Initialize the callback by saving references to the RL model and the training environment for convenience.
```
Return type None

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

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

Return type bool  
Returns If the callback returns False, training is aborted early.

update_child_locals(locals_)  
Update the references to the local variables on sub callbacks.

Parameters locals – the local variables during rollout collection

Return type None

update_locals(locals_)  
Update the references to the local variables.

Parameters locals – the local variables during rollout collection

Return type None

class stable_baselines3.common.callbacks.CallbackList(callbacks)  
Class for chaining callbacks.

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

update_child_locals(locals_)  
Update the references to the local variables.

Parameters locals – the local variables during rollout collection

Return type None

class stable_baselines3.common.callbacks.CheckpointCallback (save_freq,  
save_path,  
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
• verbose (int)–

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

Parameters

• callback (Callable[[Dict[str, Any], Dict[str, Any]], bool])–
• verbose (int)–
class stable_baselines3.common.callbacks.EvalCallback(eval_env, callback_on_new_best=None, n_eval_episodes=5, eval_freq=10000, log_path=None, best_model_save_path=None, deterministic=True, render=False, verbose=1, warn=True)

Callback for evaluating an agent.

Parameters

- **eval_env**(Union[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**(Optional[str]) – Path to a folder where the evaluations (evaluations.npz) will be saved. It will be updated at each evaluation.
- **best_model_save_path**(Optional[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.
- **render**(bool) – Whether to render or not the environment during evaluation
- **verbose**(int) –
- **warn**(bool) – Passed to evaluate_policy (warns if eval_env has not been wrapped with a Monitor wrapper)

update_child_locals(locals_)

Update the references to the local variables.

Parameters **locals** – the local variables during rollout collection

Return type None

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

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

Return type None

update_child_locals(locals_)

Update the references to the local variables.

Parameters **locals** – the local variables during rollout collection
Return type None

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

Trigger a callback every n_steps timesteps

Parameters

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

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

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

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

Parameters

• max_episodes (int) – Maximum number of episodes to stop training.
• verbose (int) – Select whether to print information about when training ended by reaching max_episodes

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

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

It must be used with the EvalCallback.

Parameters

• reward_threshold (float) – Minimum expected reward per episode to stop training.
• verbose (int) –

1.11 Tensorboard Integration

1.11.1 Basic Usage

To use Tensorboard with stable baseline3, you simply need to pass the location of the log folder to the RL agent:

```python
from stable_baselines3 import A2C

model = A2C('MlpPolicy', 'CartPole-v1', verbose=1, tensorboard_log='./a2c_cartpole_tensorboard/')
model.learn(total_timesteps=10000)
```

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

```python
from stable_baselines3 import A2C

model = A2C('MlpPolicy', 'CartPole-v1', verbose=1, tensorboard_log='./a2c_cartpole_tensorboard/')
model.learn(total_timesteps=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
```

(continues on next page)
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.

---

**Example Code:**

```python
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)
```
1.11.2 Logging More Values

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

```python
import numpy as np

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

model = SAC("MlpPolicy", "Pendulum-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

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

1.11.3 Logging Images

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

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

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

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

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

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

    def _on_step(self):
        image = self.training_env.render(mode="rgb_array")
        # "HWC" specify the dataformat of the image, here channel last
        return True

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

(continues on next page)
# See https://pytorch.org/docs/stable/tensorboard.html
# for supported formats
self.logger.record("trajectory/image", Image(image, "HWC"), exclude=\"stdout", \"log", \"json", \"csv\")
return True

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

1.11.4 Logging Figures/Plots

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

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

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

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

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

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

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

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

model.learn(50000, callback=FigureRecorderCallback())
```
1.11.5 Logging Videos

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

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

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

```python
from typing import Any, Dict
import gym
import torch as th
from stable_baselines3 import A2C
from stable_baselines3.common.callbacks import BaseCallback
from stable_baselines3.common.evaluation import evaluate_policy
from stable_baselines3.common.logger import Video

class VideoRecorderCallback(BaseCallback):
    def __init__(self, eval_env: gym.Env, render_freq: int, n_eval_episodes: int = 1, deterministic: bool = True):
        
        Records a video of an agent's trajectory traversing `eval_env` and logs it to TensorBoard
        :param eval_env: A gym environment from which the trajectory is recorded
        :param render_freq: Render the agent's trajectory every render_freq call of the callback.
        :param n_eval_episodes: Number of episodes to render
        :param deterministic: Whether to use deterministic or stochastic policy
        
        super().__init__
        self._eval_env = eval_env
        self._render_freq = render_freq
        self._n_eval_episodes = n_eval_episodes
        self._deterministic = deterministic

    def _on_step(self) -> bool:
        if self.n_calls % self._render_freq == 0:
            screens = []
            def grab_screens(_locals: Dict[str, Any], _globals: Dict[str, Any]) -> None:
                
                Renders the environment in its current state, recording the screen in the captured `screens` list
                
                :param _locals: A dictionary containing all local variables of the callback's scope
                :param _globals: A dictionary containing all global variables of the callback's scope
                
                screen = self._eval_env.render(mode="rgb_array")

            # PyTorch uses CxHxW vs HxWxC gym (and tensorflow) image convention
            
            # (continues on next page)
```

1.11. Tensorboard Integration
screens.append(screen.transpose(2, 0, 1))

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

self.logger.record(
    "trajectory/video",
    Video(th.ByteTensor([screens]), fps=40),
    exclude=("stdout", "log", "json", "csv"),
)

return True

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

1.12 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:

```bash
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

```bash
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 SB3 Contrib

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

This allows Stable-Baselines3 (SB3) to maintain a stable and compact core, while still providing the latest features, like Truncated Quantile Critics (TQC) or Quantile Regression DQN (QR-DQN).

1.13.1 Why create this repository?

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

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

Features

See documentation for the full list of included features.

**RL Algorithms:**

- Truncated Quantile Critics (TQC)
- Quantile Regression DQN (QR-DQN)

**Gym Wrappers:**

- Time Feature Wrapper

Documentation

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

Installation

To install Stable-Baselines3 contrib with pip, execute:

```
pip install sb3-contrib
```

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

To install Stable Baselines3 master version:

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

To install Stable Baselines3 contrib master version:

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

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

Here is an example of training a Quantile Regression DQN (QR-DQN) agent on the CartPole environment.

```python
from sb3_contrib import QRDQN

dict(n_quantiles=50)
model = QRDQN("MlpPolicy", "CartPole-v1", policy_kwargs=policy_kwargs, verbose=1)
model.learn(total_timesteps=10000, log_interval=4)
model.save("qrdqn_cartpole")
```

**1.14 Imitation Learning**

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

- Behavioral Cloning
- DAgger with synthetic examples
- Adversarial Inverse Reinforcement Learning (AIRL)
- Generative Adversarial Imitation Learning (GAIL)

It also provides CLI scripts for training and saving demonstrations from RL experts, and for training imitation learners on these demonstrations.

**1.14.1 Installation**

Installation requires Python 3.7+:

```
pip install imitation
```

**1.14.2 CLI Quickstart**

```
# Train PPO agent on cartpole and collect expert demonstrations
python -m imitation.scripts.expert_demos with fast cartpole log_dir=quickstart

# Train GAIL from demonstrations
python -m imitation.scripts.train_adversarial with fast gail cartpole rollout_path=quickstart/rollouts/final.pkl

# Train AIRL from demonstrations
python -m imitation.scripts.train_adversarial with fast airl cartpole rollout_path=quickstart/rollouts/final.pkl
```

**Note:** You can remove the fast option to run training to completion. For more CLI options and information on reading Tensorboard plots, see the README.
1.14.3 Python Interface Quickstart

This example script uses the Python API to train BC, GAIL, and AIRL models on CartPole data.

1.15 Migrating from Stable-Baselines

This is a guide to migrate from Stable-Baselines (SB2) to Stable-Baselines3 (SB3).
It also references the main changes.

1.15.1 Overview

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

1.15.2 How to migrate?

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

1.15.3 Breaking Changes

- SB3 requires python 3.6+ (instead of python 3.5+ for SB2)
- Dropped MPI support
- Dropped layer normalized policies (e.g. LnMlpPolicy)
- Dropped parameter noise for DDPG and DQN
- PPO is now closer to the original implementation (no clipping of the value function by default), cf PPO section below
- Orthogonal initialization is only used by A2C/PPO
- The features extractor (CNN extractor) is shared between policy and q-networks for DDPG/SAC/TD3 and only the policy loss used to update it (much faster)
- Tensorboard legacy logging was dropped in favor of having one logger for the terminal and Tensorboard (cf Tensorboard integration)
- We dropped ACKTR/ACER support because of their complexity compared to simpler alternatives (PPO, SAC, TD3) performing as good.
- We dropped GAIL support as we are focusing on model-free RL only, you can however take a look at the imitation project which implements GAIL and other imitation learning algorithms on top of SB3.

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

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

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

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

instead of `from stable_baselines3.common import make_atari_env`

Changes and renaming in parameters

Base-class (all algorithms)

- `load_parameters` -> `set_parameters`
  - `get/set_parameters` return a dictionary mapping object names to their respective PyTorch tensors and other objects representing their parameters, instead of simpler mapping of parameter name to a NumPy array. These functions also return PyTorch tensors rather than NumPy arrays.

Policies

- `cnn_extractor` -> `feature_extractor`, as `feature_extractor` is now used with `MlpPolicy` too

A2C

- `epsilon` -> `rms_prop_eps`
- `lr_schedule` is part of `learning_rate` (it can be a callable).
- `alpha`, `momentum` are modifiable through `policy_kwargs` key `optimizer_kwargs`.

**Warning:** PyTorch implementation of RMSprop differs from Tensorflow’s, which leads to different and potentially more unstable results. Use `stable_baselines3.common.sb2_compat.rmsprop_tf_like.RMSpropTFLike` optimizer to match the results with Tensorflow’s implementation. This can be done through `policy_kwargs`: `A2C(policy_kwargs=dict(optimizer_class=RMSpropTFLike, eps=1e-5))`
PPO

- `cliprange` -> `clip_range`
- `cliprange_vf` -> `clip_range_vf`
- `nminibatches` -> `batch_size`

**Warning:** `nminibatches` gave different batch size depending on the number of environments: `batch_size = (n_steps * n_envs) // nminibatches`

- `clip_range_vf` behavior for PPO is slightly different: Set it to `None` (default) to deactivate clipping (in SB2, you had to pass `-1, None` meant to use `clip_range` for the clipping)
- `lam` -> `gae_lambda`
- `noptepochs` -> `n_epochs`

PPO default hyperparameters are the one tuned for continuous control environment. We recommend taking a look at the *RL Zoo* for hyperparameters tuned for Atari games.

DQN

Only the vanilla DQN is implemented right now but extensions will follow. Default hyperparameters are taken from the nature paper, except for the optimizer and learning rate that were taken from Stable Baselines defaults.

DDPG

DDPG now follows the same interface as SAC/TD3. For state/reward normalization, you should use `VecNormalize` as for all other algorithms.

SAC/TD3

SAC/TD3 now accept any number of critics, e.g. `policy_kwargs=dict(n_critics=3)`, instead of only two before.

**Note:** SAC/TD3 default hyperparameters (including network architecture) now match the ones from the original papers. DDPG is using TD3 defaults.

SAC

SAC implementation matches the latest version of the original implementation: it uses two Q function networks and two target Q function networks instead of two Q function networks and one Value function network (SB2 implementation, first version of the original implementation). Despite this change, no change in performance should be expected.

**Note:** SAC `predict()` method has now `deterministic=False` by default for consistency. To match SB2 behavior, you need to explicitly pass `deterministic=True`
**HER**

The HER implementation now also supports online sampling of the new goals. This is done in a vectorized version. The goal selection strategy RANDOM is no longer supported. HER now supports VecNormalize wrapper but only when online_sampling=True. For performance reasons, the maximum number of steps per episodes must be specified (see HER documentation).

**New logger API**

- Methods were renamed in the logger:
  - `logkv` -> `record`, `writekvs` -> `write`, `writeseq` -> `write_sequence`,
  - `logkvs` -> `record_dict`, `dumpkvs` -> `dump`,
  - `getkvs` -> `get_log_dict`, `logkv_mean` -> `record_mean`.

**Internal Changes**

Please read the Developer Guide section.

**1.15.4 New Features (SB3 vs SB2)**

- Much cleaner and consistent base code (and no more warnings =D!) and static type checks
- Independent saving/loading/predict for policies
- A2C now supports Generalized Advantage Estimation (GAE) and advantage normalization (both are deactivated by default)
- Generalized State-Dependent Exploration (gSDE) exploration is available for A2C/PPO/SAC. It allows to use RL directly on real robots (cf https://arxiv.org/abs/2005.05719)
- Proper evaluation (using separate env) is included in the base class (using EvalCallback), if you pass the environment as a string, you can pass `create_eval_env=True` to the algorithm constructor.
- Better saving/loading: optimizers are now included in the saved parameters and there is two new methods `save_replay_buffer` and `load_replay_buffer` for the replay buffer when using off-policy algorithms (DQN/DDPG/SAC/TD3)
- You can pass `optimizer_class` and `optimizer_kwargs` to `policy_kwargs` in order to easily customize optimizers
- Seeding now works properly to have deterministic results
- Replay buffer does not grow, allocate everything at build time (faster)
- We added a memory efficient replay buffer variant (pass `optimize_memory_usage=True` to the constructor), it reduces drastically the memory used especially when using images
- You can specify an arbitrary number of critics for SAC/TD3 (e.g. `policy_kwargs=dict(n_critics=3)`)

**1.15. Migrating from Stable-Baselines**
1.16 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.

1.16.1 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, NaN mod 1, …) return NaN
- Division by zero:
  - if the operand is not zero (1/0, −2/0, …) returns ± inf
  - if the operand is zero (0/0) returns signaling NaN
- Overflow (exponent too high to represent) returns ± 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.

1.16.2 Anomaly detection with PyTorch

To enable NaN detection in PyTorch you can do

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

1.16.3 Numpy parameters

Numpy has a convenient way of dealing with invalid value: numpy.seterr, which defines for the python process, how it should handle floating point error.

```python
import numpy as np
np.seterr(all='raise')  # define before your code.
```

(continues on next page)
print("numpy test:")

a = np.float64(1.0)
b = np.float64(0.0)
val = a / b  # this will now raise an exception instead of a warning.
print(val)

but this will also avoid overflow issues on floating point numbers:

```python
import numpy as np
np.seterr(all='raise')  # define before your code.

print("numpy overflow test:")

a = np.float64(10)
b = np.float64(1000)
val = a ** b  # this will now raise an exception
print(val)
```

but will not avoid the propagation issues:

```python
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)
```

1.16.4 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.

```python
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)
```

(continues on next page)
```python
self.observation_space = spaces.Box(low=-np.inf, high=np.inf, shape=(1,),
dtype=np.float64)

    def step(self, _action):
        randf = np.random.rand()
        if randf > 0.99:
            obs = float('NaN')
        elif randf > 0.98:
            obs = float('inf')
        else:
            obs = randf
        return [obs], 0.0, False, {}  

    def reset(self):
        return [0.0]

    def render(self, 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.

1.16.5 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).

1.16.6 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
1.17 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.

1.17.1 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

1.17.2 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).

1.17.3 Pre-Processing

To handle different observation spaces, some pre-processing needs to be done (e.g. one-hot encoding for discrete observation). Most of the code for pre-processing is in `common/preprocessing.py` and `common/policies.py`.

For images, environment is automatically wrapped with `VecTransposeImage` if observations are detected to be images with channel-last convention to transform it to PyTorch’s channel-first convention.
1.17.4 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.

1.17.5 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”)

1.17.6 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 (raraffin) published a paper about a generalized version of SDE (the one implemented in SB3): https://arxiv.org/abs/2005.05719

1.17.7 Misc

The rest of the common/ is composed of helpers (e.g. evaluation helpers) or basic components (like the callbacks). The type_aliases.py file contains common type hint aliases like GymStepReturn.

Et voilà?

After reading this guide and the mentioned files, you should be now able to understand the design logic behind the library ;)

1.18 On saving and loading

Stable Baselines3 (SB3) stores both neural network parameters and algorithm-related parameters such as exploration schedule, number of environments and observation/action space. This allows continual learning and easy use of trained agents without training, but it is not without its issues. Following describes the format used to save agents in SB3 along with its pros and shortcomings.

Terminology used in this page:

- **parameters** refer to neural network parameters (also called “weights”). This is a dictionary mapping variable name to a PyTorch tensor.

- **data** refers to RL algorithm parameters, e.g. learning rate, exploration schedule, action/observation space. These depend on the algorithm used. This is a dictionary mapping classes variable names to their values.
1.18.1 Zip-archive

A zip-archived JSON dump, PyTorch state dictionaries and PyTorch variables. The data dictionary (class parameters) is stored as a JSON file, model parameters and optimizers are serialized with `torch.save()` function and these files are stored under a single .zip archive.

Any objects that are not JSON serializable are serialized with cloudpickle and stored as base64-encoded string in the JSON file, along with some information that was stored in the serialization. This allows inspecting stored objects without deserializing the object itself.

This format allows skipping elements in the file, i.e. we can skip deserializing objects that are broken/non-serializable.

File structure:

```
saved_model.zip/
    data           JSON file of class-parameters (dictionary)
    *.optimizer.pth PyTorch optimizers serialized
    policy.pth     PyTorch state dictionary of the policy saved
    pytorch_variables.pth Additional PyTorch variables
    _stable_baselines3_version contains the SB3 version with which the model was saved
```

Pros:

- More robust to unserializable objects (one bad object does not break everything).
- Saved files can be inspected/extracted with zip-archive explorers and by other languages.

Cons:

- More complex implementation.
- Still relies partly on cloudpickle for complex objects (e.g. custom functions) with can lead to incompatibilities between Python versions.

1.19 Exporting models

After training an agent, you may want to deploy/use it in another language or framework, like `tensorflowjs`. Stable Baselines3 does not include tools to export models to other frameworks, but this document aims to cover parts that are required for exporting along with more detailed stories from users of Stable Baselines3.

1.19.1 Background

In Stable Baselines3, the controller is stored inside policies which convert observations into actions. Each learning algorithm (e.g. DQN, A2C, SAC) contains a policy object which represents the currently learned behavior, accessible via `model.policy`.

Policies hold enough information to do the inference (i.e. predict actions), so it is enough to export these policies (cf examples) to do inference in another framework.

**Warning**: When using CNN policies, the observation is normalized during pre-preprocessing. This pre-processing is done inside the policy (dividing by 255 to have values in [0, 1])
1.19.2 Export to ONNX

TODO: help is welcomed!

1.19.3 Export to C++

(using PyTorch JIT) TODO: help is welcomed!

1.19.4 Export to tensorflowjs / ONNX-JS

TODO: contributors help is welcomed! Probably a good starting point: https://github.com/elliotwaite/pytorch-to-javascript-with-onnx-js

1.19.5 Manual export

You can also manually export required parameters (weights) and construct the network in your desired framework.

You can access parameters of the model via agents’ get_parameters function. As policies are also PyTorch modules, you can also access model.policy.state_dict() directly. To find the architecture of the networks for each algorithm, best is to check the policies.py file located in their respective folders.

Note: In most cases, we recommend using PyTorch methods state_dict() and load_state_dict() from the policy, unless you need to access the optimizers’ state dict too. In that case, you need to call get_parameters().

Abstract base classes for RL algorithms.

1.20 Base RL Class

Common interface for all the RL algorithms

class stable_baselines3.common.base_class.BaseAlgorithm(policy, env, policy_base, learning_rate, policy_kwargs=None, tensorboard_log=None, verbose=0, device='auto', support_multi_env=False, create_eval_env=False, monitor_wrapper=True, seed=None, use_sde=False, sde_sample_freq=-1, supported_action_spaces=None)

The base of RL algorithms

Parameters

- **policy**(Type[BasePolicy]) – Policy object
- **env**(Union[Env, VecEnv, 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** (Union[float, Callable[[float], float]]) – learning rate for the optimizer, it can be a function of the current progress remaining (from 1 to 0)
• **policy_kwargs** (Optional[Dict[str, Any]]) – Additional arguments to be passed to the policy on creation
• **tensorboard_log** (Optional[str]) – the log location for tensorboard (if None, no logging)
• **verbose** (int) – The verbosity level: 0 none, 1 training information, 2 debug
• **device** (Union[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)
• **supported_action_spaces** (Optional[Tuple[Space, ...]]) – The action spaces supported by the algorithm.

### Methods

**get_env()**

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

Return type: Optional[VecEnv]

Returns: The current environment

**get_parameters()**

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

Return type: Dict[str, Dict]

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

**get_vec_normalize_env()**

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

Return type: Optional[VecNormalize]

Returns: The VecNormalize env.

**abstract learn**(total_timesteps, callback=None, log_interval=100, tb_log_name='run', eval_env=None, eval_freq=-1, n_eval_episodes=5, eval_log_path=None, reset_num_timesteps=True)

Return a trained model.

Parameters
• **total_timesteps** (int) – The total number of samples (env steps) to train on

• **callback** (Union[None, Callable, List[BaseCallback], BaseCallback]) – callback(s) called at every step with state of the algorithm.

• **log_interval** (int) – The number of timesteps before logging.

• **tb_log_name** (str) – the name of the run for TensorBoard logging

• **eval_env** (Union[Env, VecEnv, None]) – 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)

Return type *BaseAlgorithm*

Returns the trained model

classmethod **load**(path, env=None, device='auto', **kwargs)
Load the model from a zip-file

Parameters

• **path** (Union[str, Path, BufferedIOBase]) – path to the file (or a file-like) where to load the agent from

• **env** (Union[Env, VecEnv, None]) – the new environment to run the loaded model on (can be None if you only need prediction from a trained model) has priority over any saved environment

• **device** (Union[device, str]) – Device on which the code should run.

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

Return type *BaseAlgorithm*

**predict**(observation, state=None, mask=None, deterministic=False)
Get the model’s action(s) from an observation

Parameters

• **observation** (ndarray) – the input observation

• **state** (Optional[ndarray]) – The last states (can be None, used in recurrent policies)

• **mask** (Optional[ndarray]) – The last masks (can be None, used in recurrent policies)

• **deterministic** (bool) – Whether or not to return deterministic actions.

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

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

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

Parameters
• **path** (Union[str, Path, BufferedIOBase]) – path to the file where the rl agent should be saved

• **exclude** (Optional[Iterable[str]]) – name of parameters that should be excluded in addition to the default ones

• **include** (Optional[Iterable[str]]) – name of parameters that might be excluded but should be included anyway

**Return type** None

**set_env**(*env*)

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

**Parameters**

- **env** (Union[Env, VecEnv]) – The environment for learning a policy

**Return type** None

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

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

**Parameters**

- **load_path_or_iter** – Location of the saved data (path or file-like, see save), or a nested dictionary containing nn.Module parameters used by the policy. The dictionary maps object names to a state-dictionary returned by torch.nn.Module.state_dict().

- **exact_match** (bool) – If True, the given parameters should include parameters for each module and each of their parameters, otherwise raises an Exception. If set to False, this can be used to update only specific parameters.

- **device** (Union[device, str]) – Device on which the code should run.

**Return type** None

**set_random_seed**(*seed=None*)

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

**Parameters**

- **seed** (Optional[int]) –

**Return type** None

### 1.20.1 Base Off-Policy Class

The base RL algorithm for Off-Policy algorithm (ex: SAC/TD3)
class stable_baselines3.common.off_policy_algorithm.OffPolicyAlgorithm(policy, env, policy_base, learning_rate, buffer_size=1000000, learning_starts=100, batch_size=256, tau=0.005, gamma=0.99, train_freq=1, gradient_steps=1, n_episodes_rollout=-1, action_noise=None, optimize_memory_usage=False, policy_kwargs=None, tensorboard_log=None, verbose=0, device='auto', support_multi_env=False, create_eval_env=False, monitor_wrapper=True, seed=None, use_sde=False, sde_sample_freq=-1, use_sde_at_warmup=False, sde_support=True, remove_time_limit_termination=False, supported_action_spaces=None)

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

Parameters

- **policy** (Type[BasePolicy]) – Policy object
- **env** (Union[Env, VecEnv, str]) – 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** (*Union[float, Callable[[float, float]], float]*) – learning rate for the optimizer, it can be a function of the current progress remaining (from 1 to 0)
- **buffer_size** (*int*) – size of the replay buffer
- **learning_starts** (*int*) – how many steps of the model to collect transitions for before learning starts
- **batch_size** (*int*) – Minibatch size for each gradient update
- **tau** (*float*) – the soft update coefficient (“Polyak update”, between 0 and 1)
- **gamma** (*float*) – the discount factor
- **train_freq** (*int*) – 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** (*Optional[ActionNoise]*) – the action noise type (None by default), this can help for hard exploration problem. Cf common.noise for the different action noise type.
- **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** (*Optional[Dict[str, Any]*) – Additional arguments to be passed to the policy on creation
- **tensorboard_log** (*Optional[str]*) – the log location for tensorboard (if None, no logging)
- **verbose** (*int*) – The verbosity level: 0 none, 1 training information, 2 debug
- **device** (*Union[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 State Dependent Exploration (SDE) instead of action noise exploration (default: False)
- **sde_sample_freq** (*int*) – Sample a new noise matrix every n steps when using gSDE Default: -1 (only sample at the beginning of the rollout)
• **use_sde_at_warmup** (bool) – Whether to use gSDE instead of uniform sampling during the warm up phase (before learning starts)

• **sde_support** (bool) – Whether the model support gSDE or not

• **remove_time_limit_termination** (bool) – Remove terminations (dones) that are due to time limit. See https://github.com/hill-a/stable-baselines/issues/863

• **supported_action_spaces** (Optional[ Tuple[Space, ...]]) – The action spaces supported by the algorithm.

**collect_rollouts** (env, callback, n_episodes=1, n_steps=-1, action_noise=None, learning_starts=0, replay_buffer=None, log_interval=None)

Collect experiences and store them into a ReplayBuffer.

**Parameters**

- **env** (VecEnv) – The training environment
- **callback** (BaseCallback) – Callback that will be called at each step (and at the beginning and end of the rollout)
- **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** (Optional[ReplayBuffer]) –
- **log_interval** (Optional[int]) – Log data every log_interval episodes

**Return type** RolloutReturn

**Returns**

**learn**(total_timesteps, callback=None, log_interval=4, eval_env=None, eval_freq=-1, n_eval_episodes=5, tb_log_name='run', eval_log_path=None, reset_num_timesteps=True)

Return a trained model.

**Parameters**

- **total_timesteps** (int) – The total number of samples (env steps) to train on
- **callback** (Union[None, Callable, List[BaseCallback], BaseCallback]) – callback(s) called at every step with state of the algorithm.
- **log_interval** (int) – The number of timesteps before logging.
- **tb_log_name** (str) – the name of the run for TensorBoard logging
- **eval_env** (Union[Env, VecEnv, None]) – 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 episodes 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)

  Return type: OffPolicyAlgorithm

  Returns: the trained model

### load_replay_buffer(path)
Load a replay buffer from a pickle file.

Parameters:
- `path` (Union[str, Path, BufferedIOBase]) – Path to the pickled replay buffer.

  Return type: None

### save_replay_buffer(path)
Save the replay buffer as a pickle file.

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

  Return type: None

### train(gradient_steps, batch_size)
Sample the replay buffer and do the updates (gradient descent and update target networks)

  Return type: None

### 1.20.2 Base On-Policy Class

The base RL algorithm for On-Policy algorithm (ex: A2C/PPO)
class stable_baselines3.common.on_policy_algorithm.OnPolicyAlgorithm(policy, env, learning_rate, n_steps, gamma, gae_lambda, ent_coef, vf_coef, max_grad_norm, use_sde, sde_sample_freq, tensorboard_log=None, create_eval_env=False, monitor_wrapper=True, policy_kwargs=None, verbose=0, seed=None, device='auto', _init_setup_model=True, supported_action_spaces=None)

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

Parameters

- **policy** (Union[str, Type[ActorCriticPolicy]]) – The policy model to use (MlpPolicy, CnnPolicy, ...)
- **env** (Union[Env, VecEnv, str]) – The environment to learn from (if registered in Gym, can be str)
- **learning_rate** (Union[float, Callable[[float], float]]) – The learning rate, 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** *(Optional[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** *(bool)* – When creating an environment, whether to wrap it or not in a Monitor wrapper.

• **policy_kwargs** *(Optional[Dict[str, Any]])* – additional arguments to be passed to the policy on creation

• **verbose** *(int)* – the verbosity level: 0 no output, 1 info, 2 debug

• **seed** *(Optional[int]*) – Seed for the pseudo random generators

• **device** *(Union[device, str]*) – Device (cpu, cuda, . . . ) on which the code should be run. Setting it to auto, the code will be run on the GPU if possible.

• **_init_setup_model** *(bool)* – Whether or not to build the network at the creation of the instance

• **supported_action_spaces** *(Optional[Tuple[Space, . . . ]]* – The action spaces supported by the algorithm.

---

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

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

**Parameters**

• **env** *(VecEnv)* – The training environment

• **callback** *(BaseCallback)* – Callback that will be called at each step (and at the beginning and end of the rollout)

• **rollout_buffer** *(RolloutBuffer)* – Buffer to fill with rollouts

• **n_steps** – Number of experiences to collect per environment

**Return type** bool

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

**learn**(total_timesteps, callback=None, log_interval=1, eval_env=None, eval_freq=- 1, n_eval_episodes=5, tb_log_name='OnPolicyAlgorithm', eval_log_path=None, reset_num_timesteps=True)

Return a trained model.

**Parameters**

• **total_timesteps** *(int)* – The total number of samples (env steps) to train on

• **callback** *(Union[None, Callable, List[BaseCallback], BaseCallback])* – callback(s) called at every step with state of the algorithm.

• **log_interval** *(int)* – The number of timesteps before logging.

• **tb_log_name** *(str)* – the name of the run for TensorBoard logging

• **eval_env** *(Union[Env, VecEnv, None])* – 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)

  **Return type** OnPolicyAlgorithm

  **Returns** the trained model

  **train()**

  Consume current rollout data and update policy parameters. Implemented by individual algorithms.

  **Return type** None

### 1.21 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, eps=1e-5))`. Read more [here](https://openai.com/blog/baselines-acktr-a2c/).

### 1.21.1 Notes

- OpenAI blog post: [https://openai.com/blog/baselines-acktr-a2c/](https://openai.com/blog/baselines-acktr-a2c/)

### 1.21.2 Can I use?

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

<table>
<thead>
<tr>
<th>Space</th>
<th>Action</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discrete</td>
<td>✓✓</td>
<td>✓</td>
</tr>
<tr>
<td>Box</td>
<td>✓✓</td>
<td>✓</td>
</tr>
<tr>
<td>MultiDiscrete</td>
<td>✓✓</td>
<td>✓</td>
</tr>
<tr>
<td>MultiBinary</td>
<td>✓✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
1.21.3 Example

Train an A2C agent on CartPole-v1 using 4 environments.

```python
import gym
from stable_baselines3 import A2C
from stable_baselines3.a2c import MlpPolicy
from stable_baselines3.common.env_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")

model = A2C.load("a2c_cartpole")

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

1.21.4 Results

**Atari Games**

The complete learning curves are available in the associated PR #110.

**PyBullet Environments**

Results on the PyBullet benchmark (2M steps) using 6 seeds. The complete learning curves are available in the associated issue #48.

**Note:** Hyperparameters from the gSDE paper were used (as they are tuned for PyBullet envs).

---

*Gaussian* means that the unstructured Gaussian noise is used for exploration, *gSDE* (generalized State-Dependent Exploration) is used otherwise.

<table>
<thead>
<tr>
<th>Environments</th>
<th>A2C</th>
<th>A2C</th>
<th>PPO</th>
<th>PPO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Gaussian</td>
<td>gSDE</td>
<td>Gaussian</td>
<td>gSDE</td>
</tr>
<tr>
<td>HalfCheetah</td>
<td>2003 +/- 54</td>
<td>2032 +/- 122</td>
<td>1976 +/- 479</td>
<td>2826 +/- 45</td>
</tr>
<tr>
<td>Ant</td>
<td>2286 +/- 72</td>
<td>2443 +/- 89</td>
<td>2364 +/- 120</td>
<td>2782 +/- 76</td>
</tr>
<tr>
<td>Hopper</td>
<td>1627 +/- 158</td>
<td>1561 +/- 220</td>
<td>1567 +/- 339</td>
<td>2512 +/- 21</td>
</tr>
<tr>
<td>Walker2D</td>
<td>577 +/- 65</td>
<td>839 +/- 56</td>
<td>1230 +/- 147</td>
<td>2019 +/- 64</td>
</tr>
</tbody>
</table>

**1.21. A2C**
How to replicate the results?

Clone the rl-zoo repo:

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

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

```
python train.py --algo a2c --env $ENV_ID --eval-episodes 10 --eval-freq 10000
```

Plot the results (here for PyBullet envs only):

```
python scripts/all_plots.py -a a2c -e HalfCheetah Ant Hopper Walker2D -f logs/ -o logs/a2c_results
python scripts/plot_from_file.py -i logs/a2c_results.pkl -latex -l A2C
```

1.2.1.5 Parameters

class stable_baselines3.a2c.A2C(policy, env, learning_rate=0.0007, n_steps=5, gamma=0.99, gae_lambda=1.0, ent_coef=0.0, vf_coef=0.5, max_grad_norm=0.5, rms_prop_eps=1e-05, use_rms_prop=True, use_sde=False, sde_sample_freq=-1, normalize_advantage=False, tensorboard_log=None, create_eval_env=False, policy_kwargs=None, verbose=0, seed=None, device='auto', _init_setup_model=True)

Advantage Actor Critic (A2C)


Introduction to A2C: https://hackernoon.com/intuitive-rl-intro-to-advantage-actor-critic-a2c-4ff545978752

Parameters

- **policy**(Union[str, Type[ActorCriticPolicy]]) – The policy model to use (MlpPolicy, CnnPolicy, …)
- **env**(Union[Env, VecEnv, str]) – The environment to learn from (if registered in Gym, can be str)
- **learning_rate**(Union[float, Callable[[float], float]]) – The learning rate, it can be a function of the current progress remaining (from 1 to 0)
- **n_steps**(int) – The number of steps to run for each environment per update (i.e. batch size is n_steps * n_env where n_env is number of environment copies running in parallel)
- **gamma**(float) – Discount factor
- **gae_lambda**(float) – Factor for trade-off of bias vs variance for Generalized Advantage Estimator Equivalent to classic advantage when set to 1.
- **ent_coef**(float) – Entropy coefficient for the loss calculation
- **vf_coef**(float) – Value function coefficient for the loss calculation
- **max_grad_norm**(float) – The maximum value for the gradient clipping
- **rms_prop_eps**(float) – RMSProp epsilon. It stabilizes square root computation in denominator of RMSProp update
• use_rms_prop (bool) – Whether to use RMSprop (default) or Adam as optimizer
• use_sde (bool) – Whether to use generalized State Dependent Exploration (gSDE) instead of action noise exploration (default: False)
• sde_sample_freq (int) – Sample a new noise matrix every n steps when using gSDE Default: -1 (only sample at the beginning of the rollout)
• normalize_advantage (bool) – Whether to normalize or not the advantage
• tensorboard_log (Optional[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 (Optional[Dict[str, Any]]) – additional arguments to be passed to the policy on creation
• verbose (int) – the verbosity level: 0 no output, 1 info, 2 debug
• seed (Optional[int]) – Seed for the pseudo random generators
• device (Union[device, str]) – Device (cpu, cuda, ...) on which the code should be run. Setting it to auto, the code will be run on the GPU if possible.
• _init_setup_model (bool) – Whether or not to build the network at the creation of the instance

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

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

Return type bool

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

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

Return type Optional[VecEnv]

Returns The current environment

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

Return type Dict[str, Dict]

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

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

Return type Optional[VecNormalize]

Returns The VecNormalize env.

learn(total_timesteps, callback=None, log_interval=100, eval_env=None, eval_freq=-1, n_eval_episodes=5, tb_log_name='A2C', eval_log_path=None, reset_num_timesteps=True)

Return a trained model.

Parameters

• total_timesteps (int) – The total number of samples (env steps) to train on
• callback (Union[None, Callable, List[BaseCallback], BaseCallback]) – callback(s) called at every step with state of the algorithm.
• log_interval (int) – The number of timesteps before logging.
• tb_log_name (str) – the name of the run for TensorBoard logging
• eval_env (Union[Env, VecEnv, None]) – 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)

Return type A2C

Returns the trained model

classmethod load(path, env=None, device='auto', **kwargs)

Load the model from a zip-file

Parameters

• path (Union[str, Path, BufferedIOBase]) – path to the file (or a file-like) where to load the agent from
• env (Union[Env, VecEnv, None]) – the new environment to run the loaded model on (can be None if you only need prediction from a trained model) has priority over any saved environment
• device (Union[device, str]) – Device on which the code should run.
• kwargs – extra arguments to change the model when loading

Return type BaseAlgorithm

predict(observation, state=None, mask=None, deterministic=False)

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

Parameters

• observation (ndarray) – the input observation
• state (Optional[ndarray]) – The last states (can be None, used in recurrent policies)
• **mask** (Optional[ndarray]) – The last masks (can be None, used in recurrent policies)

• **deterministic** (bool) – Whether or not to return deterministic actions.

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

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

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

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

**Parameters**

• **path** (Union[str, Path, BufferedIOBase]) – path to the file where the rl agent should be saved

• **exclude** (Optional[Iterable[str]]) – name of parameters that should be excluded in addition to the default ones

• **include** (Optional[Iterable[str]]) – name of parameters that might be excluded but should be included anyway

**Return type** None

**set_env** (*env*)

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

**Parameters**

• **env** (Union[Env, VecEnv]) – The environment for learning a policy

**Return type** None

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

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

**Parameters**

• **load_path_or_iter** – Location of the saved data (path or file-like, see save), or a nested dictionary containing nn.Module parameters used by the policy. The dictionary maps object names to a state-dictionary returned by torch.nn.Module.state_dict().

• **exact_match** (bool) – If True, the given parameters should include parameters for each module and each of their parameters, otherwise raises an Exception. If set to False, this can be used to update only specific parameters.

• **device** (Union[device, str]) – Device on which the code should run.

**Return type** None

**set_random_seed** (*seed=None*)

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

**Parameters**

• **seed** (Optional[int]) –

**Return type** None

**train**()

Update policy using the currently gathered rollout buffer (one gradient step over whole data).

**Return type** None
### 1.21.6 A2C Policies

**stable_baselines3.a2c.MlpPolicy**  
alias of `stable_baselines3.common.policies.ActorCriticPolicy`  

```python

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

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

**Parameters**

- **observation_space** (Space) – Observation space
- **action_space** (Space) – Action space
- **lr_schedule** (Callable[[float], float]) – Learning rate schedule (could be constant)
- **net_arch** (Optional[List[Union[int, Dict[str, List[int]]]]]) – The specification of the policy and value networks.
- **activation_fn** (Type[Module]) – Activation function
- **ortho_init** (bool) – Whether to use or not orthogonal initialization
- **use_sde** (bool) – Whether to use State Dependent Exploration or not
- **log_std_init** (float) – Initial value for the log standard deviation
- **full_std** (bool) – Whether to use \(n_{features} \times n_{actions}\) parameters for the std instead of only \(n_{features}\) when using gSDE
- **sde_net_arch** (Optional[List[int]]) – Network architecture for extracting features when using gSDE. If None, the latent features from the policy will be used. Pass an empty list to use the states as 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 allows to ensure boundaries when using gSDE.

• **features_extractor_class** (Type[BaseFeaturesExtractor]) – Features extractor to use.

• **features_extractor_kwargs** (Optional[Dict[str, Any]]) – Keyword arguments to pass to the features extractor.

• **normalize_images** (bool) – Whether to normalize images or not, dividing by 255.0 (True by default)

• **optimizer_class** (Type[Optimizer]) – The optimizer to use, th.optim.Adam by default

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

**evaluate_actions**(obs, actions)
Evaluate actions according to the current policy, given the observations.

Parameters

• **obs** (Tensor) –
• **actions** (Tensor) –

Return type Tuple[Tensor, Tensor, Tensor]

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

**forward**(obs, deterministic=False)
Forward pass in all the networks (actor and critic)

Parameters

• **obs** (Tensor) – Observation

• **deterministic** (bool) – Whether to sample or use deterministic actions

Return type Tuple[Tensor, Tensor, Tensor]

Returns action, value and log probability of the action

**reset_noise**(n_envs=1)
Sample new weights for the exploration matrix.

Parameters **n_envs** (int) –

Return type None

**stable_baselines3.a2c.CnnPolicy**
alias of **stable_baselines3.common.policies.ActorCriticCnnPolicy**
class stable_baselines3.common.policies.ActorCriticCnnPolicy:

    observation_space (Space) – Observation space
    action_space (Space) – Action space
    lr_schedule (Callable[[float, float]]) – Learning rate schedule (could be constant)
    net_arch (Optional[List[Union[int, Dict[str, List[int]]]]]) – The specification of the policy and value networks.
    activation_fn (Type[Module]) – Activation function
    ortho_init (bool) – Whether to use or not orthogonal initialization
    use_sde (bool) – Whether to use State Dependent Exploration or not
    log_std_init (float) – Initial value for the log standard deviation
    full_std (bool) – Whether to use (n_features x n_actions) parameters for the std instead of only (n_features,) when using gSDE
    sde_net_arch (Optional[List[int]]) – Network architecture for extracting features when using gSDE. If None, the latent features from the policy will be used. Pass an empty list to use the states as 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.

CNN policy class for actor-critic algorithms (has both policy and value prediction). Used by A2C, PPO and the likes.
• **squash_output** (bool) – Whether to squash the output using a tanh function, this allows to ensure boundaries when using gSDE.

• **features_extractor_class** (Type[BaseFeaturesExtractor]) – Features extractor to use.

• **features_extractor_kwargs** (Optional[Dict[str, Any]]) – Keyword arguments to pass to the features extractor.

• **normalize_images** (bool) – Whether to normalize images or not, dividing by 255.0 (True by default)

• **optimizer_class** (Type[Optimizer]) – The optimizer to use, th.optim.Adam by default

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

### 1.22 DDPG

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

**Note:** As DDPG can be seen as a special case of its successor **TD3**, they share the same policies and same implementation.

#### Available Policies

<table>
<thead>
<tr>
<th>Policy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MlpPolicy</td>
<td>alias of stable_baselines3.td3.policies.</td>
</tr>
<tr>
<td></td>
<td>TD3Policy</td>
</tr>
<tr>
<td>CnnPolicy</td>
<td>Policy class (with both actor and critic) for TD3.</td>
</tr>
</tbody>
</table>

#### 1.22.1 Notes


#### 1.22.2 Can I use?

- Recurrent policies:
- Multi processing:
- Gym spaces:
### 1.22.3 Example

```python
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()
```

### 1.22.4 Results

**PyBullet Environments**

Results on the PyBullet benchmark (1M steps) using 6 seeds. The complete learning curves are available in the associated issue #48.

**Note:** Hyperparameters of TD3 from the gSDE paper were used for DDPG.

*Gaussian* means that the unstructured Gaussian noise is used for exploration, gSDE (generalized State-Dependent Exploration) is used otherwise.
### Environments

<table>
<thead>
<tr>
<th>Environments</th>
<th>DDPG</th>
<th>TD3</th>
<th>SAC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Gaussian</td>
<td>Gaussian</td>
<td>gSDE</td>
</tr>
<tr>
<td>HalfCheetah</td>
<td>2272 +/- 69</td>
<td>2774 +/- 35</td>
<td>2984 +/- 202</td>
</tr>
<tr>
<td>Ant</td>
<td>1651 +/- 407</td>
<td>3305 +/- 43</td>
<td>3102 +/- 37</td>
</tr>
<tr>
<td>Hopper</td>
<td>1201 +/- 211</td>
<td>2429 +/- 126</td>
<td>2262 +/- 1</td>
</tr>
<tr>
<td>Walker2D</td>
<td>882 +/- 186</td>
<td>2063 +/- 185</td>
<td>2136 +/- 67</td>
</tr>
</tbody>
</table>

### How to replicate the results?

Clone the rl-zoo repo:

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

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

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

Plot the results:

```
python scripts/all_plots.py -a ddpg -e HalfCheetah Ant Hopper Walker2D -f logs/ -o logs/ddpg_results
python scripts/plot_from_file.py -i logs/ddpg_results.pkl -latex -l DDPG
```

### 1.22.5 Parameters

```python
class stable_baselines3.ddpg.DDPG(policy, env, learning_rate=0.001, buffer_size=1000000, learning_starts=100, batch_size=100, tau=0.005, gamma=0.99, train_freq=-1, gradient_steps=-1, n_episodes_rollout=1, action_noise=None, optimize_memory_usage=False, tensorboard_log=None, create_eval_env=False, policy_kwargs=None, verbose=0, seed=None, device='auto', _init_setup_model=True)
```

Deep Deterministic Policy Gradient (DDPG).


Note: we treat DDPG as a special case of its successor TD3.

**Parameters**

- **policy** (Union[str, Type[TD3Policy]]) – The policy model to use (MlpPolicy, CnnPolicy, ...)
- **env** (Union[Env, VecEnv, str]) – The environment to learn from (if registered in Gym, can be str)
- **learning_rate** (Union[float, Callable[[float], float]]) – 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** *(Optional[ActionNoise])* – the action noise type (None by default), this can help for hard exploration problem. Cf common.noise for the different action noise type.

• **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** *(Optional[Dict[str, Any]])* – additional arguments to be passed to the policy on creation

• **verbose** *(int)* – the verbosity level: 0 no output, 1 info, 2 debug

• **seed** *(Optional[int])* – Seed for the pseudo random generators

• **device** *(Union[device, str])* – Device (cpu, cuda, ...) on which the code should be run. Setting it to auto, the code will be run on the GPU if possible.

• **_init_setup_model** *(bool)* – Whether or not to build the network at the creation of the instance

```python
collect_rollouts(env, callback, n_episodes=1, n_steps=-1, action_noise=None, learning_starts=0, replay_buffer=None, log_interval=None)
```
Collect experiences and store them into a ReplayBuffer.

**Parameters**

• **env** *(VecEnv)* – The training environment

• **callback** *(BaseCallback)* – Callback that will be called at each step (and at the beginning and end of the rollout)

• **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** *(Optional[ReplayBuffer])* –
• **log_interval** *(Optional[int]) – Log data every log_interval episodes*
  
  **Return type** RolloutReturn

  **Returns**

  **get_env()**
  
  Returns the current environment (can be None if not defined).
  
  **Return type** Optional[VecEnv]

  **Returns** The current environment

  **get_parameters()**
  
  Return the parameters of the agent. This includes parameters from different networks, e.g. critics (value functions) and policies (pi functions).
  
  **Return type** Dict[str, Dict]

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

  **get_vec_normalize_env()**
  
  Return the VecNormalize wrapper of the training env if it exists.
  
  **Return type** Optional[VecNormalize]

  **Returns** The VecNormalize env.

  **learn**(total_timesteps, callback=None, log_interval=4, eval_env=None, eval_freq=-1, n_eval_episodes=5, tb_log_name='DDPG', eval_log_path=None, reset_num_timesteps=True)
  
  Return a trained model.

  **Parameters**

  • **total_timesteps** *(int) – The total number of samples (env steps) to train on*

  • **callback** *(Union[None, Callable, List[BaseCallback], BaseCallback]] – callback(s) called at every step with state of the algorithm.*

  • **log_interval** *(int) – The number of timesteps before logging.*

  • **tb_log_name** *(str) – the name of the run for TensorBoard logging*

  • **eval_env** *(Union[Env, VecEnv, None]) – 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)*

  **Return type** OffPolicyAlgorithm

  **Returns** the trained model

  **classmethod load**(path, env=None, device='auto', **kwargs)
  
  Load the model from a zip-file

  **Parameters**

  • **path** *(Union[str, Path, BufferedIOBase]) – path to the file (or a file-like) where to load the agent from*
• **env** (Union[Env, VecEnv, None]) – the new environment to run the loaded model on (can be None if you only need prediction from a trained model) has priority over any saved environment
• **device** (Union[device, str]) – Device on which the code should run.
• **kwargs** – extra arguments to change the model when loading

**Return type** BaseAlgorithm

**load_replay_buffer**(path)
Load a replay buffer from a pickle file.

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

**Return type** None

**predict**(observation, state=None, mask=None, deterministic=False)
Get the model’s action(s) from an observation

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

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

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

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

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

**Return type** None

**save_replay_buffer**(path)
Save the replay buffer as a pickle file.

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

**Return type** None

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

- observation_space
- action_space

**Parameters**
- **env** (Union[Env, VecEnv]) – The environment for learning a policy
Return type None

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

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

Parameters

- **load_path_or_dict** – Location of the saved data (path or file-like, see `save`), or a nested dictionary containing nn.Module parameters used by the policy. The dictionary maps object names to a state-dictionary returned by `torch.nn.Module.state_dict()`.

- **exact_match** (bool) – If True, the given parameters should include parameters for each module and each of their parameters, otherwise raises an Exception. If set to False, this can be used to update only specific parameters.

- **device** (Union[device, str]) – Device on which the code should run.

Return type None

```python
set_random_seed(seed=None)
```

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

Parameters **seed** (Optional[int]) –

Return type None

```python
train(gradient_steps, batch_size=100)
```

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

Return type None

### 1.22.6 DDPG Policies

```python
stable_baselines3.ddpg.MlpPolicy
```

alias of `stable_baselines3.td3.policies.TD3Policy`

```python
class stable_baselines3.td3.policies.TD3Policy(observations_space, action_space, lr_schedule, net_arch=None, activation_fn=<class 'torch.nn.modules.activation.ReLU'>, features_extractor_class=<class 'stable_baselines3.common.torch_layers.FlattenExtractor'>, features_extractor_kwargs=None, normalize_images=True, optimizer_class=<class 'torch.optim.adam.Adam'>, optimizer_kwargs=None, n_critics=2, share_features_extractor=True)
```

Policy class (with both actor and critic) for TD3.

Parameters

- **observation_space** (Space) – Observation space

- **action_space** (Space) – Action space

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

• **activation_fn (Type[Module])** – Activation function

• **features_extractor_class (Type[BaseFeaturesExtractor])** – Features extractor to use.

• **features_extractor_kwargs (Optional[Dict[str, Any]])** – Keyword arguments to pass to the features extractor.

• **normalize_images (bool)** – Whether to normalize images or not, dividing by 255.0 (True by default)

• **optimizer_class (Type[Optimizer])** – The optimizer to use, th.optim.Adam by default

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

• **n_critics (int)** – Number of critic networks to create.

• **share_features_extractor (bool)** – Whether to share or not the features extractor between the actor and the critic (this saves computation time)

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

**Return type** Tensor


Policy class (with both actor and critic) for TD3.

**Parameters**

• **observation_space (Space)** – Observation space

• **action_space (Space)** – Action space

• **lr_schedule (Callable[[float, float]])** – Learning rate schedule (could be constant)

• **net_arch (Union[List[int], Dict[str, List[int]], None])** – The specification of the policy and value networks.

• **activation_fn (Type[Module])** – Activation function
• **features_extractor_class** (Type[BaseFeaturesExtractor]) – Features extractor to use.

• **features_extractor_kwargs** (Optional[Dict[str, Any]]) – Keyword arguments to pass to the features extractor.

• **normalize_images** (bool) – Whether to normalize images or not, dividing by 255.0 (True by default)

• **optimizer_class** (Type[Optimizer]) – The optimizer to use, th.optim.Adam by default

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

• **n_critics** (int) – Number of critic networks to create.

• **share_features_extractor** (bool) – Whether to share or not the features extractor between the actor and the critic (this saves computation time)

### 1.23 DQN

Deep Q Network (DQN) builds on Fitted Q-Iteration (FQI) and make use of different tricks to stabilize the learning with neural networks: it uses a replay buffer, a target network and gradient clipping.

**Available Policies**

<table>
<thead>
<tr>
<th>MlpPolicy</th>
<th>alias of stable_baselines3.dqn.policies.DQNPolicy</th>
</tr>
</thead>
<tbody>
<tr>
<td>CnnPolicy</td>
<td>Policy class for DQN when using images as input.</td>
</tr>
</tbody>
</table>

#### 1.23.1 Notes

- Further reference: [https://www.nature.com/articles/nature14236](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.

#### 1.23.2 Can I use?

- Recurrent policies:
- Multi processing:
- Gym spaces:
### 1.23.3 Example

```python
code
import gym
import numpy as np
from stable_baselines3 import DQN
from stable_baselines3.dqn import MlpPolicy

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

### 1.23.4 Results

#### Atari Games

The complete learning curves are available in the associated PR #110.

**How to replicate the results?**

Clone the rl-zoo repo:

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

Run the benchmark (replace $ENV_ID by the env id, for instance BreakoutNoFrameskip-v4):

```bash
code
python train.py --algo dqn --env $ENV_ID --eval-episodes 10 --eval-freq 10000
```

Plot the results:

<table>
<thead>
<tr>
<th>Space</th>
<th>Action</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discrete</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Box</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>MultiDiscrete</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>MultiBinary</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
1.23.5 Parameters

```python
class stable_baselines3.dqn.DQN(policy, env, learning_rate=0.0001, buffer_size=1000000, learning_starts=50000, batch_size=32, tau=1.0, gamma=0.99, train_freq=4, gradient_steps=1, n_episodes_rollout=-1, optimize_memory_usage=False, target_update_interval=10000, exploration_fraction=0.1, exploration_initial_eps=1.0, exploration_final_eps=0.05, max_grad_norm=10, tensorboard_log=None, create_eval_env=False, policy_kwargs=None, verbose=0, seed=None, device='auto', _init_setup_model=True)
```

Deep Q-Network (DQN)

Paper: https://arxiv.org/abs/1312.5602, https://www.nature.com/articles/nature14236 Default hyperparameters are taken from the nature paper, except for the optimizer and learning rate that were taken from Stable Baselines defaults.

**Parameters**

- **policy** (Union[str, Type[DQNPolicy]]) – The policy model to use (MlpPolicy, CnnPolicy,...)
- **env** (Union[Env, VecEnv, str]) – The environment to learn from (if registered in Gym, can be str)
- **learning_rate** (Union[float, Callable[[float], float]]) – The learning rate, it can be a function of the current progress remaining (from 1 to 0)
- **buffer_size** (int) – size of the replay buffer
- **learning_starts** (int) – how many steps of the model to collect transitions for before learning starts
- **batch_size** (Optional[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** (Optional[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** (Optional[Dict[str, Any]]) – additional arguments to be passed to the policy on creation

• **verbose** (int) – the verbosity level: 0 no output, 1 info, 2 debug

• **seed** (Optional[int]) – Seed for the pseudo random generators

• **device** (Union[device, str]) – Device (cpu, cuda, . . . ) on which the code should be run. Setting it to auto, the code will be run on the GPU if possible.

• **_init_setup_model** (bool) – Whether or not to build the network at the creation of the instance

**collect_rollouts**(env, callback, n_episodes=1, n_steps=-1, action_noise=None, learning_starts=0, replay_buffer=None, log_interval=None)

Collect experiences and store them into a ReplayBuffer.

Parameters

• **env** (VecEnv) – The training environment

• **callback** (BaseCallback) – Callback that will be called at each step (and at the beginning and end of the rollout)

• **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** (Optional[ReplayBuffer]) –

• **log_interval** (Optional[int]) – Log data every log_interval episodes

**Return type** RolloutReturn

**get_env**()

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

**Return type** Optional[VecEnv]

**Returns** The current environment
get_parameters()
Return the parameters of the agent. This includes parameters from different networks, e.g. critics (value functions) and policies (pi functions).

**Return type** Dict[str, Dict]

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

get_vec_normalize_env()
Return the VecNormalize wrapper of the training env if it exists.

**Return type** Optional[VecNormalize]

**Returns** The VecNormalize env.

learn(total_timesteps, callback=None, log_interval=4, eval_env=None, eval_freq=-1, n_eval_episodes=5, tb_log_name='DQN', eval_log_path=None, reset_num_timesteps=True)
Return a trained model.

**Parameters**
- **total_timesteps** (int) – The total number of samples (env steps) to train on
- **callback** (Union[None, Callable, List[BaseCallback], BaseCallback]) – callback(s) called at every step with state of the algorithm.
- **log_interval** (int) – The number of timesteps before logging.
- **tb_log_name** (str) – the name of the run for TensorBoard logging
- **eval_env** (Union[Env, VecEnv, None]) – 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 episodes 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)

**Return type** OffPolicyAlgorithm

**Returns** the trained model

classmethod load(path, env=None, device='auto', **kwargs)
Load the model from a zip-file

**Parameters**
- **path** (Union[str, Path, BufferedIOBase]) – path to the file (or a file-like) where to load the agent from
- **env** (Union[Env, VecEnv, None]) – the new environment to run the loaded model on (can be None if you only need prediction from a trained model) has priority over any saved environment
- **device** (Union[device, str]) – Device on which the code should run.
- **kwargs** – extra arguments to change the model when loading

**Return type** BaseAlgorithm
**load_replay_buffer** *(path)*

Load a replay buffer from a pickle file.

**Parameters**

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

**Return type** None

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

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

**Parameters**

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

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

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

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

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

**Parameters**

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

**Return type** None

**save_replay_buffer** *(path)*

Save the replay buffer as a pickle file.

**Parameters**

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

**Return type** None

**set_env** *(env)*

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

**Parameters**

- **env** *(Union[Env, VecEnv])* – The environment for learning a policy

**Return type** None

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

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

**Parameters**
• **load_path_or_iter** – Location of the saved data (path or file-like, see `save`), or a nested dictionary containing nn.Module parameters used by the policy. The dictionary maps object names to a state-dictionary returned by `torch.nn.Module.state_dict()`.

• **exact_match** (bool) – If True, the given parameters should include parameters for each module and each of their parameters, otherwise raises an Exception. If set to False, this can be used to update only specific parameters.

• **device** (Union[device, str]) – Device on which the code should run.

  Return type None

  **set_random_seed**(seed=None)
  
  Set the seed of the pseudo-random generators (python, numpy, pytorch, gym, action_space)

  Parameters seed(Optional[int]) –
  
  Return type None

  **train**(gradient_steps, batch_size=100)
  
  Sample the replay buffer and do the updates (gradient descent and update target networks)

  Return type None

### 1.23.6 DQN Policies

**stable_baselines3.dqn.MlpPolicy**

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

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

Policy class with Q-Value Net and target net for DQN

Parameters

• **observation_space** (Space) – Observation space

• **action_space** (Space) – Action space

• **lr_schedule** (Callable[[float], float]) – Learning rate schedule (could be constant)

• **net_arch** (Optional[List[int]]) – The specification of the policy and value networks.

• **activation_fn** (Type[Module]) – Activation function

• **features_extractor_class** (Type[BaseFeaturesExtractor]) – Features extractor to use.

• **features_extractor_kwargs** (Optional[Dict[str, Any]]) – Keyword arguments to pass to the features extractor.
• **normalize_images** (bool) – Whether to normalize images or not, dividing by 255.0 (True by default)

• **optimizer_class** (Type[Optimizer]) – The optimizer to use, th.optim.Adam by default

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

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

**Return type** Tensor

class stable_baselines3.dqn.CnnPolicy

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

Policy class for DQN when using images as input.

**Parameters**

• **observation_space** (Space) – Observation space

• **action_space** (Space) – Action space

• **lr_schedule** (Callable[[float, float]]) – Learning rate schedule (could be constant)

• **net_arch** (Optional[List[int]]) – The specification of the policy and value networks.

• **activation_fn** (Type[Module]) – Activation function

• **features_extractor_class** (Type[BaseFeaturesExtractor]) – Features extractor to use.

• **normalize_images** (bool) – Whether to normalize images or not, dividing by 255.0 (True by default)

• **optimizer_class** (Type[Optimizer]) – The optimizer to use, th.optim.Adam by default

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

Hindsight Experience Replay (HER)

HER is an algorithm that works with off-policy methods (DQN, SAC, TD3 and DDPG for example). HER uses the fact that even if a desired goal was not achieved, other goal may have been achieved during a rollout. It creates “virtual” transitions by relabeling transitions (changing the desired goal) from past episodes.

**Warning:** HER requires the environment to inherits from gym.GoalEnv

**Warning:** For performance reasons, the maximum number of steps per episodes must be specified. In most cases, it will be inferred if you specify max_episode_steps when registering the environment or if you use a gym.wrappers.TimeLimit (and env.spec is not None). Otherwise, you can directly pass max_episode_length to the model constructor

**Warning:** HER supports VecNormalize wrapper but only when online_sampling=True

1.24.1 Notes

- Original paper: https://arxiv.org/abs/1707.01495
- OpenAI paper: Plappert et al. (2018)
- OpenAI blog post: https://openai.com/blog/ingredients-for-robotics-research/

1.24.2 Can I use?

Please refer to the used model (DQN, SAC, TD3 or DDPG) for that section.

1.24.3 Example

```python
from stable_baselines3 import HER, DDPG, DQN, SAC, TD3
from stable_baselines3.her.goal_selection_strategy import GoalSelectionStrategy
from stable_baselines3.common.bit_flipping_env import BitFlippingEnv
from stable_baselines3.common.vec_env import DummyVecEnv
from stable_baselines3.common.vec_env.obs_dict_wrapper import ObsDictWrapper

model_class = DQN  # works also with SAC, DDPG and TD3
N_BITS = 15

env = BitFlippingEnv(n_bits=N_BITS, continuous=model_class in [DDPG, SAC, TD3], max_steps=N_BITS)

# Available strategies (cf paper): future, final, episode
goal_selection_strategy = 'future'  # equivalent to GoalSelectionStrategy.FUTURE

# If True the HER transitions will get sampled online
```

(continues on next page)
```python
online_sampling = True
# Time limit for the episodes
max_episode_length = N_BITS

# Initialize the model
model = HER('MlpPolicy', env, model_class, n_sampled_goal=4, goal_selection_strategy='future', online_sampling=online_sampling, verbose=1, max_episode_length=max_episode_length)

# Train the model
model.learn(1000)

model.save("./her_bit_env")
model = HER.load('./her_bit_env', env=env)

obs = env.reset()
for _ in range(100):
    action, _ = model.model.predict(obs, deterministic=True)
    obs, reward, done, _ = env.step(action)
    if done:
        obs = env.reset()
```

### 1.24.4 Results

This implementation was tested on the parking env using 3 seeds.

The complete learning curves are available in the associated PR #120.

**How to replicate the results?**

**Clone the rl-zoo repo:**

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

**Run the benchmark:**

```
python train.py --algo her --env parking-v0 --eval-episodes 10 --eval-freq 10000
```

**Plot the results:**

```
python scripts/all_plots.py -a her -e parking-v0 -f logs/ --no-million
```

### 1.24.5 Parameters

```python
class stable_baselines3.her.HER(policy, env, model_class, n_sampled_goal=4,
goal_selection_strategy='future', online_sampling=False,
max_episode_length=None, *args, **kwargs)
```

For performance reasons, the maximum number of steps per episodes must be specified. In most cases, it will be inferred if you specify `max_episode_steps` when registering the environment or if you use a `gym.wrappers.TimeLimit` (and `env.spec` is not None). Otherwise, you can directly pass `max_episode_length` to the model constructor.

For additional offline algorithm specific arguments please have a look at the corresponding documentation.

**Parameters**

- `policy` (Union[Type[BasePolicy]]) – The policy model to use.
- `env` (Union[Env, VecEnv, str]) – The environment to learn from (if registered in Gym, can be str)
- `model_class` (Type[OffPolicyAlgorithm]) – Off policy model which will be used with hindsight experience replay. (SAC, TD3, DDPG, DQN)
- `n_sampled_goal` (int) – Number of sampled goals for replay. (offline sampling)
- `goal_selection_strategy` (Union[GoalSelectionStrategy, str]) – Strategy for sampling goals for replay. One of ['episode', 'final', 'future', 'random']
- `online_sampling` (bool) – Sample HER transitions online.
- `learning_rate` – learning rate for the optimizer, it can be a function of the current progress remaining (from 1 to 0)
- `max_episode_length` (Optional[int]) – The maximum length of an episode. If not specified, it will be automatically inferred if the environment uses a `gym.wrappers.TimeLimit` wrapper.

**collect_rollouts** (`env, callback, n_episodes=1, n_steps=-1, action_noise=None, learning_starts=0, log_interval=None`)

Collect experiences and store them into a ReplayBuffer.

**Parameters**

- `env` (VecEnv) – The training environment
- `callback` (BaseCallback) – Callback that will be called at each step (and at the beginning and end of the rollout)
- `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.
- `log_interval` (Optional[int]) – Log data every `log_interval` episodes

**Return type** RolloutReturn

**Returns**

**learn** (`total_timesteps, callback=None, log_interval=4, eval_env=None, eval_freq=-1, n_eval_episodes=5, tb_log_name='HER', eval_log_path=None, reset_num_timesteps=True`)

Return a trained model.
Parameters

- **total_timesteps (int)** – The total number of samples (env steps) to train on.
- **callback (Union[None, Callable, List[BaseCallback], BaseCallback])** – callback(s) called at every step with state of the algorithm.
- **log_interval (int)** – The number of timesteps before logging.
- **tb_log_name (str)** – the name of the run for TensorBoard logging.
- **eval_env (Union[Env, VecEnv, None])** – 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).

Return type **BaseAlgorithm**

Returns the trained model.

**classmethod load**(path, env=None, device='auto', **kwargs)

Load the model from a zip-file.

Parameters

- **path (Union[str, Path, BufferedReader])** – path to the file (or a file-like) where to load the agent from.
- **env (Union[Env, VecEnv, None])** – the new environment to run the loaded model on (can be None if you only need prediction from a trained model) has priority over any saved environment.
- **device (Union[device, str])** – Device on which the code should run.
- **kwargs** – extra arguments to change the model when loading.

Return type **BaseAlgorithm**

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

Load a replay buffer from a pickle file and set environment for replay buffer (only online sampling).

Parameters

- **path (Union[str, Path, BufferedReader])** – Path to the pickled replay buffer.
- **truncate_last_trajectory (bool)** – Only for online sampling. If set to True we assume that the last trajectory in the replay buffer was finished. If it is set to False we assume that we continue the same trajectory (same episode).

Return type **None**

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

Get the model’s action(s) from an observation.

Parameters

- **observation (ndarray)** – the input observation.
• **state** *(Optional[ndarray])* – The last states (can be None, used in recurrent policies)

• **mask** *(Optional[ndarray])* – The last masks (can be None, used in recurrent policies)

• **deterministic** *(bool)* – Whether or not to return deterministic actions.

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

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

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

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

**Parameters**

• **path** *(Union[str, Path, BufferedIOBase])* – path to the file where the rl agent should be saved

• **exclude** *(Optional[Iterable[str]])* – name of parameters that should be excluded in addition to the default one

• **include** *(Optional[Iterable[str]])* – name of parameters that might be excluded but should be included anyway

**Return type** None

### 1.24.6 Goal Selection Strategies

**class** `stable_baselines3.her.GoalSelectionStrategy` *(value)*

The strategies for selecting new goals when creating artificial transitions.

### 1.24.7 Obs Dict Wrapper

**class** `stable_baselines3.her.ObsDictWrapper` *(env)*

Wrapper for a VecEnv which overrides the observation space for Hindsight Experience Replay to support dict observations.

**Parameters**

• **env** *– The vectorized environment to wrap.*

**close** ()

Clean up the environment’s resources.

**Return type** None

**static convert_dict** *(observation_dict, observation_key='observation', goal_key='desired_goal')*

Concatenate observation and (desired) goal of observation dict.

**Parameters**

• **observation_dict** *(Dict[str, ndarray])* – Dictionary with observation.

• **observation_key** *(str)* – Key of observation in dictionary.

• **goal_key** *(str)* – Key of (desired) goal in dictionary.

**Return type** ndarray

**Returns** Concatenated observation.

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

Check if environments are wrapped with a given wrapper.
Parameters

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

Return type List[bool]
Returns True if the env is wrapped, False otherwise, for each env queried.

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** (Union[None, int, Iterable[int]]) – Indices of envs whose method to call
• **method_args** – Any positional arguments to provide in the call
• **method_kwargs** – Any keyword arguments to provide in the call

Return type List[Any]
Returns List of items returned by the environment’s method call

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** (Union[None, int, Iterable[int]]) – Indices of envs to get attribute from

Return type List[Any]
Returns List of values of ‘attr_name’ in all environments

get_images()
Return RGB images from each environment

Return type Sequence[ndarray]

getattr_depth_check(name, already_found)
See base class.

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

getattr_recursive(name)
Recursively check wrappers to find attribute.

Parameters **name** (str) – name of attribute to look for

Return type Any
Returns attribute
render (mode='human')
Gym environment rendering

Parameters
mode (str) – the rendering type

Return type Optional[ndarray]

reset ()
Reset all the environments and return an array of observations, or a tuple of observation arrays.
If step_async is still doing work, that work will be cancelled and step_wait() should not be called until step_async() is invoked again.

Returns observation

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.

Return type List[Optional[int]]

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

set_attr (attr_name, value, indices=None)
Set attribute inside vectorized environments.

Parameters
• attr_name (str) – The name of attribute to assign new value
• value (Any) – Value to assign to attr_name
• indices (Union[None, int, Iterable[int]]) – Indices of envs to assign value

Return type None

Returns

step (actions)
Step the environments with the given action

Parameters
actions (ndarray) – the action

Return type Tuple[Union[ndarray, Dict[str, ndarray], Tuple[ndarray, ...]], ndarray, ndarray, List[Dict]]

Returns observation, reward, done, information

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.

Return type None

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

Returns observation, reward, done, information
1.24.8 HER Replay Buffer

```python
class stable_baselines3.her.HerReplayBuffer(env, buffer_size, max_episode_length, goal_selection_strategy, observation_space, action_space, device='cpu', n_envs=1, her_ratio=0.8)
```

Replay buffer for sampling HER (Hindsight Experience Replay) transitions. In the online sampling case, these new transitions will not be saved in the replay buffer and will only be created at sampling time.

**Parameters**

- `env` (ObsDictWrapper) – The training environment
- `buffer_size` (int) – The size of the buffer measured in transitions.
- `max_episode_length` (int) – The length of an episode. (time horizon)
- `goal_selection_strategy` (GoalSelectionStrategy) – Strategy for sampling goals for replay. One of ['episode', ‘final’, ‘future’]
- `observation_space` (Space) – Observation space
- `action_space` (Space) – Action space
- `device` (Union[device,str]) – PyTorch device
- `n_envs` (int) – Number of parallel environments

**Her_ratio** The ratio between HER transitions and regular transitions in percent (between 0 and 1, for online sampling) The default value `her_ratio=0.8` corresponds to 4 virtual transitions for one real transition (4 / (4 + 1) = 0.8)

**add**(obs, next_obs, action, reward, done, infos)

Add elements to the buffer.

**Return type** None

**extend**(args, **kwargs)

Add a new batch of transitions to the buffer.

**Return type** None

**reset**()

Reset the buffer.

**Return type** None

**sample**(batch_size, env)

Sample function for online sampling of HER transition, this replaces the “regular” replay buffer `sample()` method in the `train()` function.

**Parameters**

- `batch_size` (int) – Number of element to sample
- `env` (Optional[VecNormalize]) – Associated gym VecEnv to normalize the observations/rewards when sampling

**Return type** Union[ReplayBufferSamples, Tuple[ndarray,...]]

**Returns** Samples.

**sample_goals**(episode_indices, her_indices, transitions_indices)

Sample goals based on `goal_selection_strategy`. This is a vectorized (fast) version.

**Parameters**
• **episode_indices** (ndarray) – Episode indices to use.
• **her_indices** (ndarray) – HER indices.
• **transitions_indices** (ndarray) – Transition indices to use.

Returns type ndarray
Returns Return sampled goals.

### sample_offline(n_sampled_goal=None)
Sample function for offline sampling of HER transition, in that case, only one episode is used and transitions are added to the regular replay buffer.

Parameters
- **n_sampled_goal** (Optional[int]) – Number of sampled goals for replay

Returns
- Union[ReplayBufferSamples, Tuple[ndarray, ...]]

at most(n_sampled_goal * episode_length) HER transitions.

### set_env(env)
Sets the environment.

Parameters
- **env** (ObsDictWrapper) –

Returns type None

### size()

Returns type int

Returns The current number of transitions in the buffer.

### store_episode()
Increment episode counter and reset transition pointer.

Returns type None

### static swap_and_flatten(arr)
Swap and then flatten axes 0 (buffer_size) and 1 (n_envs) to convert shape from [n_steps, n_envs, ...] (when ... is the shape of the features) to [n_steps * n_envs, ...] (which maintain the order)

Parameters
- **arr** (ndarray) –

Returns
- ndarray

### to_torch(array, copy=True)
Convert a numpy array to a PyTorch tensor. Note: it copies the data by default

Parameters

- **array** (ndarray) –
- **copy** (bool) – Whether to copy or not the data (may be useful to avoid changing things be reference)

Returns
- Tensor
1.25 PPO

The Proximal Policy Optimization algorithm combines ideas from A2C (having multiple workers) and TRPO (it uses a trust region to improve the actor).

The main idea is that after an update, the new policy should be not too far from the old policy. For that, ppo uses clipping to avoid too large update.

Note: PPO contains several modifications from the original algorithm not documented by OpenAI: advantages are normalized and value function can be also clipped.

1.25.1 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/

1.25.2 Can I use?

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

<table>
<thead>
<tr>
<th>Space</th>
<th>Action</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discrete</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Box</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MultiDiscrete</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MultiBinary</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

1.25.3 Example

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

```python
import gym
from stable_baselines3 import PPO
from stable_baselines3.ppo import MlpPolicy
from stable_baselines3.common.env_util import make_vec_env

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

model = PPO(MlpPolicy, env, verbose=1)
model.learn(total_timesteps=25000)
model.save("ppo_cartpole")
```

(continues on next page)
```python
model = PPO.load("ppo_cartpole")
obs = env.reset()
while True:
    action, _states = model.predict(obs)
    obs, rewards, dones, info = env.step(action)
    env.render()
```

### 1.25.4 Results

#### Atari Games

The complete learning curves are available in the associated PR #110.

#### PyBullet Environments

Results on the PyBullet benchmark (2M steps) using 6 seeds. The complete learning curves are available in the associated issue #48.

**Note:** Hyperparameters from the gSDE paper were used (as they are tuned for PyBullet envs).

<table>
<thead>
<tr>
<th>Environments</th>
<th>A2C Gaussian</th>
<th>A2C gSDE</th>
<th>PPO Gaussian</th>
<th>PPO gSDE</th>
</tr>
</thead>
<tbody>
<tr>
<td>HalfCheetah</td>
<td>2003 +/- 54</td>
<td>2032 +/- 122</td>
<td>1976 +/- 479</td>
<td>2826 +/- 45</td>
</tr>
<tr>
<td>Ant</td>
<td>2286 +/- 72</td>
<td>2443 +/- 89</td>
<td>2364 +/- 120</td>
<td>2782 +/- 76</td>
</tr>
<tr>
<td>Hopper</td>
<td>1627 +/- 158</td>
<td>1561 +/- 220</td>
<td>1567 +/- 339</td>
<td>2512 +/- 21</td>
</tr>
<tr>
<td>Walker2D</td>
<td>577 +/- 65</td>
<td>839 +/- 56</td>
<td>1230 +/- 147</td>
<td>2019 +/- 64</td>
</tr>
</tbody>
</table>

#### How to replicate the results?

Clone the rl-zoo repo:

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

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

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

Plot the results (here for PyBullet ppo envs only):

```bash
python scripts/all_plots.py -a ppo -e HalfCheetah Ant Hopper Walker2D -f logs/ -o logs/ppo_results
python scripts/plot_from_file.py -i logs/ppo_results.pkl -latex -1 PPO
```
1.25.5 Parameters

class stable_baselines3.ppo.PPO(
policy,  
env,  
learning_rate=0.0003,  
n_steps=2048,  
batch_size=64,  
n_epochs=10,  
gamma=0.99,  
gae_lambda=0.95,  
clip_range=0.2,  
clip_range_vf=None,  
ent_coef=0.0,  
vf_coef=0.5,  
max_grad_norm=0.5,  
use_sde=False,  
sde_sample_freq=-1,  
target_kl=None,  
tensorboard_log=None,  
create_eval_env=False,  
policy_kwargs=None,  
verbose=0,  
seed=None,  
device='auto',  
_init_setup_model=True)

Proximal Policy Optimization algorithm (PPO) (clip version)


Parameters

- **policy** (Union[str, Type[ActorCriticPolicy]]) – The policy model to use (MlpPolicy, CnnPolicy, ...)
- **env** (Union[Env, VecEnv, str]) – The environment to learn from (if registered in Gym, can be str)
- **learning_rate** (Union[float, Callable[[float], float]]) – The learning rate, it can be a function of the current progress remaining (from 1 to 0)
- **n_steps** (int) – The number of steps to run for each environment per update (i.e. rollout buffer size is n_steps * n_envs where n_envs is number of environment copies running in parallel) NOTE: n_steps * n_envs must be greater than 1 (because of the advantage normalization) See https://github.com/pytorch/pytorch/issues/29372
- **batch_size** (Optional[int]) – Minibatch size
- **n_epochs** (int) – Number of epoch when optimizing the surrogate loss
- **gamma** (float) – Discount factor
- **gae_lambda** (float) – Factor for trade-off of bias vs variance for Generalized Advantage Estimator
- **clip_range** (Union[float, Callable[[float], float]]) – Clipping parameter, it can be a function of the current progress remaining (from 1 to 0).
- **clip_range_vf** (Union[None, float, Callable[[float], float]]) – Clipping parameter for the value function, it can be a function of the current progress remaining (from 1 to 0). This is a parameter specific to the OpenAI implementation. If None is passed (default), no clipping will be done on the value function. IMPORTANT: this clipping depends on the reward scaling.
- **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** *(Optional [float]*) – Limit the KL divergence between updates, because the clipping is not enough to prevent large update see issue #213 (cf https://github.com/hill-a/stable-baselines/issues/213) By default, there is no limit on the kl div.

- **tensorboard_log** *(Optional [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** *(Optional [Dict [str, Any]*) – additional arguments to be passed to the policy on creation

- **verbose** *(int)* – the verbosity level: 0 no output, 1 info, 2 debug

- **seed** *(Optional [int]*) – Seed for the pseudo random generators

- **device** *(Union [device, str]*) – Device (cpu, cuda, ...) on which the code should be run. Setting it to auto, the code will be run on the GPU if possible.

- **_init_setup_model** *(bool)* – Whether or not to build the network at the creation of the instance

### collect_rollouts

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

**Parameters**

- **env** *(VecEnv)* – The training environment

- **callback** *(BaseCallback)* – Callback that will be called at each step (and at the beginning and end of the rollout)

- **rollout_buffer** *(RolloutBuffer)* – Buffer to fill with rollouts

- **n_steps** – Number of experiences to collect per environment

**Return type** bool

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

### get_env

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

**Return type** Optional [VecEnv]

**Returns** The current environment

### get_parameters

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

**Return type** Dict [str, Dict]

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

### get_vec_normalize_env

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

**Return type** Optional [VecNormalize]

**Returns** The VecNormalize env.
learn(total_timesteps, callback=None, log_interval=1, eval_env=None, eval_freq=-1, n_eval_episodes=5, tb_log_name='PPO', eval_log_path=None, reset_num_timesteps=True)

Return a trained model.

Parameters

- **total_timesteps** (int) – The total number of samples (env steps) to train on
- **callback** (Union[None, Callable, List[BaseCallback], BaseCallback]) – callback(s) called at every step with state of the algorithm.
- **log_interval** (int) – The number of timesteps before logging.
- **tb_log_name** (str) – the name of the run for TensorBoard logging
- **eval_env** (Union[Env, VecEnv, None]) – 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)

Return type PPO

Returns the trained model

classmethod load(path, env=None, device='auto', **kwargs)

Load the model from a zip-file

Parameters

- **path** (Union[str, Path, BufferedIOBase]) – path to the file (or a file-like) where to load the agent from
- **env** (Union[Env, VecEnv, None]) – the new environment to run the loaded model on (can be None if you only need prediction from a trained model) has priority over any saved environment
- **device** (Union[device, str]) – Device on which the code should run.
- **kwargs** – extra arguments to change the model when loading

Return type BaseAlgorithm

predict(observation, state=None, mask=None, deterministic=False)

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

Parameters

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

Return type Tuple[ndarray, Optional[ndarray]]

Returns the model’s action and the next state (used in recurrent policies)
save (path, exclude=None, include=None)
Save all the attributes of the object and the model parameters in a zip-file.

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

Return type None

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

Parameters env (Union[Env, VecEnv]) – The environment for learning a policy

Return type None

set_parameters (load_path_or_dict, exact_match=True, device='auto')
Load parameters from a given zip-file or a nested dictionary containing parameters for different modules (see get_parameters).

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

Return type None

set_random_seed (seed=None)
Set the seed of the pseudo-random generators (python, numpy, pytorch, gym, action_space)

Parameters seed (Optional[int]) –

Return type None

train ()
Update policy using the currently gathered rollout buffer.

Return type None
1.25.6 PPO Policies


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

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

Parameters

- **observation_space (Space)** – Observation space
- **action_space (Space)** – Action space
- **lr_schedule (Callable[[float], float])** – Learning rate schedule (could be constant)
- **net_arch (Optional[List[Union[int, Dict[str, List[int]]]]])** – The specification of the policy and value networks.
- **activation_fn (Type[Module])** – Activation function
- **ortho_init (bool)** – Whether to use or not orthogonal initialization
- **use_sde (bool)** – Whether to use State Dependent Exploration or not
- **log_std_init (float)** – Initial value for the log standard deviation
- **full_std (bool)** – Whether to use (n_features x n_actions) parameters for the std instead of only (n_features,) when using gSDE
- **sde_net_arch (Optional[List[int]]**) – Network architecture for extracting features when using gSDE. If None, the latent features from the policy will be used. Pass an empty list to use the states as 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 allows to ensure boundaries when using gSDE.

• **features_extractor_class** (Type[BaseFeaturesExtractor]) – Features extractor to use.

• **features_extractor_kwargs** (Optional[Dict[str, Any]]) – Keyword arguments to pass to the features extractor.

• **normalize_images** (bool) – Whether to normalize images or not, dividing by 255.0 (True by default)

• **optimizer_class** (Type[Optimizer]) – The optimizer to use, `th.optim.Adam` by default

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

**evaluate_actions**( obs, actions)
Evaluate actions according to the current policy, given the observations.

Parameters

• **obs** (Tensor)

• **actions** (Tensor)

Return type Tuple[Tensor, Tensor, Tensor]

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

**forward**( obs, deterministic=False)
Forward pass in all the networks (actor and critic)

Parameters

• **obs** (Tensor) – Observation

• **deterministic** (bool) – Whether to sample or use deterministic actions

Return type Tuple[Tensor, Tensor, Tensor]

Returns action, value and log probability of the action

**reset_noise**( n_envs=1)
Sample new weights for the exploration matrix.

Parameters **n_envs**(int) –

Return type None

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

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

**Parameters**

- `observation_space` (Space) – Observation space
- `action_space` (Space) – Action space
- `lr_schedule` (Callable[[float, float]]) – Learning rate schedule (could be constant)
- `net_arch` (Optional[List[Union[int, Dict[str, List[int]]]]]) – The specification of the policy and value networks.
- `activation_fn` (Type[Module]) – Activation function
- `ortho_init` (bool) – Whether to use or not orthogonal initialization
- `use_sde` (bool) – Whether to use State Dependent Exploration or not
- `log_std_init` (float) – Initial value for the log standard deviation
- `full_std` (bool) – Whether to use (n_features x n_actions) parameters for the std instead of only (n_features,) when using gSDE
- `sde_net_arch` (Optional[List[int]]) – Network architecture for extracting features when using gSDE. If None, the latent features from the policy will be used. Pass an empty list to use the states as 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 allows to ensure boundaries when using gSDE.

• **features_extractor_class** (Type[BaseFeaturesExtractor]) – Features extractor to use.

• **features_extractor_kwargs** (Optional[Dict[str, Any]]) – Keyword arguments to pass to the features extractor.

• **normalize_images** (bool) – Whether to normalize images or not, dividing by 255.0 (True by default)

• **optimizer_class** (Type[Optimizer]) – The optimizer to use, th.optim.Adam by default

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

**1.26 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**

<table>
<thead>
<tr>
<th>Policy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MlpPolicy</td>
<td>alias of stable_baselines3.sac.policies.</td>
</tr>
<tr>
<td>CnnPolicy</td>
<td>SACPolicy Policy class (with both actor and critic) for SAC.</td>
</tr>
</tbody>
</table>

**1.26.1 Notes**

- Original paper: [https://arxiv.org/abs/1801.01290](https://arxiv.org/abs/1801.01290)
- Original Implementation: [https://github.com/haarnoja/sac](https://github.com/haarnoja/sac)
- Blog post on using SAC with real robots: [https://bair.berkeley.edu/blog/2018/12/14/sac/](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

1.26. SAC
1.26.2 Can I use?

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

<table>
<thead>
<tr>
<th>Space</th>
<th>Action</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discrete</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Box</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MultiDiscrete</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>MultiBinary</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

1.26.3 Example

```python
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")

model = SAC.load("sac_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()
```

1.26.4 Results

PyBullet Environments

Results on the PyBullet benchmark (1M steps) using 3 seeds. The complete learning curves are available in the associated issue #48.

**Note:** Hyperparameters from the gSDE paper were used (as they are tuned for PyBullet envs).

*Gaussian* means that the unstructured Gaussian noise is used for exploration, *gSDE* (generalized State-Dependent Exploration) is used otherwise.
Environments

<table>
<thead>
<tr>
<th>Environments</th>
<th>SAC Gaussian</th>
<th>SAC gSDE</th>
<th>TD3 Gaussian</th>
</tr>
</thead>
<tbody>
<tr>
<td>HalfCheetah</td>
<td>2757 +/- 53</td>
<td>2984 +/- 202</td>
<td>2774 +/- 35</td>
</tr>
<tr>
<td>Ant</td>
<td>3146 +/- 35</td>
<td>3102 +/- 37</td>
<td>3305 +/- 43</td>
</tr>
<tr>
<td>Hopper</td>
<td>2422 +/- 168</td>
<td>2262 +/- 1</td>
<td>2429 +/- 126</td>
</tr>
<tr>
<td>Walker2D</td>
<td>2184 +/- 54</td>
<td>2136 +/- 67</td>
<td>2063 +/- 185</td>
</tr>
</tbody>
</table>

How to replicate the results?

Clone the rl-zoo repo:

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

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

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

Plot the results:

```
python scripts/all_plots.py -a sac -e HalfCheetah Ant Hopper Walker2D -f logs/ -o logs/sac_results
python scripts/plot_from_file.py -i logs/sac_results.pkl -latex -l SAC
```

1.26.5 Parameters

class stable_baselines3.sac.SAC(policy, env, learning_rate=0.0003, buffer_size=1000000, learning_starts=100, batch_size=256, tau=0.005, gamma=0.99, train_freq=1, gradient_steps=1, n_episodes_rollout=1, action_noise=None, optimize_memory_usage=False, ent_coef='auto', target_update_interval=1, target_entropy='auto', use_sde=False, sde_sample_freq=-1, use_sde_at_warmup=False, tensorboard_log=None, create_eval_env=False, policy_kwargs=None, verbose=0, seed=None, device='auto', _init_setup_model=True)


Note: we use double q target and not value target as discussed in https://github.com/hill-a/stable-baselines/issues/270

Parameters

- **policy** (Union[str, Type[SACPolicy]]) – The policy model to use (MlpPolicy, CnnPolicy, ...)
- **env** (Union[Env, VecEnv, str]) – The environment to learn from (if registered in Gym, can be str)
• **learning_rate** (Union[float, Callable[[float], float]]) – 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** (Optional[ActionNoise]) – the action noise type (None by default), this can help for hard exploration problem. Cf common.noise for the different action noise type.
• **optimize_memory_usage** (bool) – Enable a memory efficient variant of the replay buffer at a cost of more complexity. See https://github.com/DLR-RM/stable-baselines3/issues/37#issuecomment-637501195
• **ent_coef** (Union[str, float]) – Entropy regularization coefficient. (Equivalent to inverse of reward scale in the original SAC paper.) Controlling exploration/exploitation trade-off. Set it to ‘auto’ to learn it automatically (and ‘auto_0.1’ for using 0.1 as initial value)
• **target_update_interval** (int) – update the target network every target_network_update_freq gradient steps.
• **target_entropy** (Union[str, float]) – target entropy when learning ent_coef (ent_coef = 'auto')
• **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** (Optional[Dict[str, Any]]) – additional arguments to be passed to the policy on creation
• **verbose** (int) – the verbosity level: 0 no output, 1 info, 2 debug
• **seed** (Optional[int]) – Seed for the pseudo random generators
• **device** (Union[device, str]) – Device (cpu, cuda, ...) on which the code should be run. Setting it to auto, the code will be run on the GPU if possible.

• **_init_setup_model** (bool) – Whether or not to build the network at the creation of the instance

**collect_rollouts** (env, callback, n_episodes=1, n_steps=-1, action_noise=None, learning_starts=0, replay_buffer=None, log_interval=None)

Collect experiences and store them into a ReplayBuffer.

**Parameters**

• **env** (VecEnv) – The training environment

• **callback** (BaseCallback) – Callback that will be called at each step (and at the beginning and end of the rollout)

• **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** (Optional[ReplayBuffer]) –

• **log_interval** (Optional[int]) – Log data every log_interval episodes

**Return type** RolloutReturn

**Returns**

**get_env** ()

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

**Return type** Optional[VecEnv]

**Returns** The current environment

**get_parameters** ()

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

**Return type** Dict[str, Dict]

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

**get_vec_normalize_env** ()

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

**Return type** Optional[VecNormalize]

**Returns** The VecNormalize env.

**learn** (total_timesteps, callback=None, log_interval=4, eval_env=None, eval_freq=-1, n_eval_episodes=5, tb_log_name='SAC', eval_log_path=None, reset_num_timesteps=True)

Return a trained model.

**Parameters**

• **total_timesteps** (int) – The total number of samples (env steps) to train on
**callback** (Union[None, Callable, List[BaseCallback], BaseCallback]) – callback(s) called at every step with state of the algorithm.

**log_interval** (int) – The number of timesteps before logging.

**tb_log_name** (str) – the name of the run for TensorBoard logging

**eval_env** (Union[Env, VecEnv, None]) – 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)

Return type **OffPolicyAlgorithm**

Returns the trained model

**classmethod load**( *path*, *env=None, device='auto', **kwargs* )

Load the model from a zip-file

Parameters

- **path** (Union[str, Path, BufferedIOBase]) – path to the file (or a file-like) where to load the agent from

- **env** (Union[Env, VecEnv, None]) – the new environment to run the loaded model on (can be None if you only need prediction from a trained model) has priority over any saved environment

- **device** (Union[device, str]) – Device on which the code should run.

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

Return type **BaseAlgorithm**

**load_replay_buffer**( *path* )

Load a replay buffer from a pickle file.

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

Return type None

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

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

Parameters

- **observation** (ndarray) – the input observation

- **state** (Optional[ndarray]) – The last states (can be None, used in recurrent policies)

- **mask** (Optional[ndarray]) – The last masks (can be None, used in recurrent policies)

- **deterministic** (bool) – Whether or not to return deterministic actions.

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

The model’s action and the next state (used in recurrent policies)

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

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

Parameters

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

Return type None

save_replay_buffer(path)

Save the replay buffer as a pickle file.

Parameters

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

Return type None

set_env(env)

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

- observation_space
- action_space

Parameters

- **env** (Union[Env, VecEnv]) – The environment for learning a policy

Return type None

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

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

Parameters

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

Return type None

set_random_seed(seed=None)

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

Parameters

- **seed** (Optional[int]) –

Return type None

train(gradient_steps, batch_size=64)

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

Return type None

1.26. SAC
1.26.6 SAC Policies

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

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

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

Parameters

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

• *n_critics* (int) – Number of critic networks to create.

• *share_features_extractor* (bool) – Whether to share or not the features extractor between the actor and the critic (this saves computation time)

```python
def forward(obs, deterministic=False)
```

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

**Return type** Tensor

```
def reset_noise(batch_size=1)
```

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

**Parameters**

- **batch_size** (int) –

**Return type** None

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

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

**Parameters**

• *observation_space* (Space) – Observation space

• *action_space* (Space) – Action space

• *lr_schedule* (Callable[[float], float]) – Learning rate schedule (could be constant)

• *net_arch* (Union[List[int], Dict[str, List[int]], None]) – The specification of the policy and value networks.

• *activation_fn* (Type[Module]) – Activation function

• *use_sde* (bool) – Whether to use State Dependent Exploration or not

• *log_std_init* (float) – Initial value for the log standard deviation

• *sde_net_arch* (Optional[List[int]]) – Network architecture for extracting features when using gSDE. If None, the latent features from the policy will be used. Pass an empty list to use the states as features.
• **use_expln** (bool) – Use expln() function instead of exp() when using gSDE to ensure a positive standard deviation (cf paper). It allows to keep variance above zero and prevent it from growing too fast. In practice, exp() is usually enough.

• **clip_mean** (float) – Clip the mean output when using gSDE to avoid numerical instability.

• **features_extractor_class** (Type[BaseFeaturesExtractor]) – Features extractor to use.

• **normalize_images** (bool) – Whether to normalize images or not, dividing by 255.0 (True by default)

• **optimizer_class** (Type[Optimizer]) – The optimizer to use, th.optim.Adam by default

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

• **n_critics** (int) – Number of critic networks to create.

• **share_features_extractor** (bool) – Whether to share or not the features extractor between the actor and the critic (this saves computation time)

### 1.27 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**

<table>
<thead>
<tr>
<th>Policy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MlpPolicy</td>
<td>alias of stable_baselines3.td3.policies.</td>
</tr>
<tr>
<td></td>
<td>TD3Policy</td>
</tr>
<tr>
<td>CnnPolicy</td>
<td>Policy class (with both actor and critic) for TD3.</td>
</tr>
</tbody>
</table>

#### 1.27.1 Notes

- Original Implementation: [https://github.com/sfujim/TD3](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
1.27.2 Can I use?

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

<table>
<thead>
<tr>
<th>Space</th>
<th>Action</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discrete</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Box</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MultiDiscrete</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MultiBinary</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

1.27.3 Example

```python
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()
```
1.27.4 Results

PyBullet Environments

Results on the PyBullet benchmark (1M steps) using 3 seeds. The complete learning curves are available in the associated issue #48.

Note: Hyperparameters from the gSDE paper were used (as they are tuned for PyBullet envs).

Gaussian means that the unstructured Gaussian noise is used for exploration, gSDE (generalized State-Dependent Exploration) is used otherwise.

<table>
<thead>
<tr>
<th>Environments</th>
<th>SAC Gaussian</th>
<th>SAC gSDE</th>
<th>TD3 Gaussian</th>
</tr>
</thead>
<tbody>
<tr>
<td>HalfCheetah</td>
<td>2757 +/- 53</td>
<td>2984 +/- 202</td>
<td>2774 +/- 35</td>
</tr>
<tr>
<td>Ant</td>
<td>3146 +/- 35</td>
<td>3102 +/- 37</td>
<td>3305 +/- 43</td>
</tr>
<tr>
<td>Hopper</td>
<td>2422 +/- 168</td>
<td>2262 +/- 1</td>
<td>2429 +/- 126</td>
</tr>
<tr>
<td>Walker2D</td>
<td>2184 +/- 54</td>
<td>2136 +/- 67</td>
<td>2063 +/- 185</td>
</tr>
</tbody>
</table>

How to replicate the results?

Clone the rl-zoo repo:

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

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

```
python train.py --algo td3 --env $ENV_ID --eval-episodes 10 --eval-freq 10000
```

Plot the results:

```
python scripts/all_plots.py -a td3 -e HalfCheetah Ant Hopper Walker2D -f logs/ -o logs/td3_results
python scripts/plot_from_file.py -i logs/td3_results.pkl -latex -l TD3
```

1.27.5 Parameters

```
class stable_baselines3.td3.TD3 (policy, env, learning_rate=0.001, buffer_size=1000000, learning_starts=100, batch_size=100, tau=0.005, gamma=0.99, train_freq=1, gradient_steps=1, n_episodes_rollout=1, action_noise=None, optimize_memory_usage=False, policy_delay=2, target_policy_noise=0.2, target_noise_clip=0.5, tensorboard_log=None, create_eval_env=False, policy_kwargs=None, verbose=0, seed=None, device='auto', _init_setup_model=True)
```

Twin Delayed DDPG (TD3) Addressing Function Approximation Error in Actor-Critic Methods.

• **policy** (Union[Union[str, Type[TD3Policy]]]) – The policy model to use (MlpPolicy, CnnPolicy, ...)

• **env** (Union[Env, VecEnv, str]) – The environment to learn from (if registered in Gym, can be str)

• **learning_rate** (Union[float, Callable[[float], float]]) – 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** (Optional[ActionNoise]) – the action noise type (None by default), this can help for hard exploration problem. Cf common.noise for the different action noise type.

• **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.

• **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** (Optional[Dict[str, Any]]) – additional arguments to be passed to the policy on creation

• **verbose** (int) – the verbosity level: 0 no output, 1 info, 2 debug

• **seed** (Optional[int]) – Seed for the pseudo random generators

• **device** (Union[device, str]) – Device (cpu, cuda, ...) on which the code should be run. Setting it to auto, the code will be run on the GPU if possible.
**_init_setup_model** *(bool)* – Whether or not to build the network at the creation of the instance

**collect_rollouts** *(env, callback, n_episodes=1, n_steps=-1, action_noise=None, learning_starts=0, replay_buffer=None, log_interval=None)*

Collect experiences and store them into a ReplayBuffer.

**Parameters**

- **env** *(VecEnv)* – The training environment
- **callback** *(BaseCallback)* – Callback that will be called at each step (and at the beginning and end of the rollout)
- **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** *(Optional[ReplayBuffer])* –
- **log_interval** *(Optional[int])* – Log data every **log_interval** episodes

**Return type** RolloutReturn

**Returns**

**get_env()**

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

**Return type** Optional[VecEnv]

**Returns** The current environment

**get_parameters()**

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

**Return type** Dict[str, Dict]

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

**get_vec_normalize_env()**

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

**Return type** Optional[VecNormalize]

**Returns** The VecNormalize env.

**learn** *(total_timesteps, callback=None, log_interval=4, eval_env=None, eval_freq=-1, n_eval_episodes=5, tb_log_name='TD3', eval_log_path=None, reset_num_timesteps=True)*

Return a trained model.

**Parameters**

- **total_timesteps** *(int)* – The total number of samples (env steps) to train on
- **callback** *(Union[None, Callable, List[BaseCallback], BaseCallback])* – callback(s) called at every step with state of the algorithm.
• **log_interval** (int) – The number of timesteps before logging.

• **tb_log_name** (str) – the name of the run for TensorBoard logging

• **eval_env** (Union[Env, VecEnv, None]) – 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)

Return type *OffPolicyAlgorithm*

Returns the trained model

**classmethod load**(path, env=None, device='auto', **kwargs)

Load the model from a zip-file

Parameters

• **path** (Union[str, Path, BufferedIOBase]) – path to the file (or a file-like) where to load the agent from

• **env** (Union[Env, VecEnv, None]) – the new environment to run the loaded model on (can be None if you only need prediction from a trained model) has priority over any saved environment

• **device** (Union[device, str]) – Device on which the code should run.

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

Return type *BaseAlgorithm*

**load_replay_buffer**(path)

Load a replay buffer from a pickle file.

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

Return type None

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

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

Parameters

• **observation** (ndarray) – the input observation

• **state** (Optional[ndarray]) – The last states (can be None, used in recurrent policies)

• **mask** (Optional[ndarray]) – The last masks (can be None, used in recurrent policies)

• **deterministic** (bool) – Whether or not to return deterministic actions.

Return type Tuple[ndarray, Optional[ndarray]]

Returns the model’s action and the next state (used in recurrent policies)
**save** *(path, exclude=None, include=None)*  
Save all the attributes of the object and the model parameters in a zip-file.

**Parameters**

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

**Return type** None

**save_replay_buffer** *(path)*  
Save the replay buffer as a pickle file.

**Parameters**

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

**Return type** None

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

**Parameters**

- **env** *(Union[Env, VecEnv])* – The environment for learning a policy

**Return type** None

**set_parameters** *(load_path_or_iter, exact_match=True, device='auto')*
Load parameters from a given zip-file or a nested dictionary containing parameters for different modules (see get_parameters).

**Parameters**

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

**Return type** None

**set_random_seed** *(seed=None)*  
Set the seed of the pseudo-random generators (python, numpy, pytorch, gym, action_space)

**Parameters**

- **seed** *(Optional[int])* –

**Return type** None

**train** *(gradient_steps, batch_size=100)*  
Sample the replay buffer and do the updates (gradient descent and update target networks)

**Return type** None
1.27.6 TD3 Policies

```python
class stable_baselines3.td3.policies.TD3Policy(  
    observation_space,  
    action_space,  
    lr_schedule,  
    net_arch=None,  
    activation_fn=<class 'torch.nn.modules.activation.ReLU'>,  
    features_extractor_class=<class 'stable_baselines3.common.torch_layers.FlattenExtractor'>,  
    features_extractor_kwargs=None,  
    normalize_images=True,  
    optimizer_class=<class 'torch.optim.adam.Adam'>,  
    optimizer_kwargs=None,  
    n_critics=2,  
    share_features_extractor=True)
```

Policy class (with both actor and critic) for TD3.

**Parameters**

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

```python
forward(observation, deterministic=False)
```

 Defines the computation performed at every call.

 Should be overridden by all subclasses.

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

Policy class (with both actor and critic) for TD3.

Parameters

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

1.28 Atari Wrappers

class stable_baselines3.common.atari_wrappers.AtariWrapper(env, noop_max=30, frame_skip=4, screen_size=84, terminal_on_life_loss=True, clip_reward=True)

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** (*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)
Clips the reward to {+1, 0, -1} by its sign.

Parameters

- **env** (*Env*) – the environment

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

Parameters

- **reward** (*float*) –

Returns

float

class stable_baselines3.common.atari_wrappers.EpisodicLifeEnv(env)
Make end-of-life == end-of-episode, but only reset on true game over. Done by DeepMind for the DQN and co. since it helps value estimation.

Parameters

- **env** (*Env*) – the environment to wrap

reset(**kwargs**)
Calls the Gym environment reset, only when lives are exhausted. This way all states are still reachable even though lives are episodic, and the learner need not know about any of this behind-the-scenes.

Parameters

- **kwargs** – Extra keywords passed to env.reset() call

Returns

ndarray

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

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

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)

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

class stable_baselines3.common.atari_wrappers.FireResetEnv(env)
    Take action on reset for environments that are fixed until firing.

    Parameters
    env (Env) – the environment to wrap

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

    Returns: observation (object): the initial observation.

    Return type ndarray

class stable_baselines3.common.atari_wrappers.MaxAndSkipEnv(env, skip=4)
    Return only every skip-th frame (frameskipping)

    Parameters
    • env (Env) – the environment
    • skip (int) – number of skip-th frame

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

    Returns: observation (object): the initial observation.

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

    step(action)
    Step the environment with the given action Repeat action, sum reward, and max over last observations.

    Parameters
    action (int) – the action

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

    Returns observation, reward, done, information

class stable_baselines3.common.atari_wrappers.NoopResetEnv(env, noop_max=30)
    Sample initial states by taking random number of no-ops on reset. No-op is assumed to be action 0.

    Parameters
    • env (Env) – the environment to wrap
noop_max (int) – the maximum value of no-ops to run

reset(**kwargs)

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

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

Returns: observation (object): the initial observation.

Return type: ndarray

class stable_baselines3.common.atari_wrappers.WarpFrame (env, width=84, height=84)

Convert to grayscale and warp frames to 84x84 (default) as done in the Nature paper and later work.

Parameters

• env (Env) – the environment
• width (int) –
• height (int) –

observation (frame)

returns the current observation from a frame

Parameters frame (ndarray) – environment frame

Return type: ndarray

Returns: the observation

1.29 Environments Utils

stable_baselines3.common.env_util.is_wrapped (env, wrapper_class)

Check if a given environment has been wrapped with a given wrapper.

Parameters

• env (Type[Env]) – Environment to check
• wrapper_class (Type[Wrapper]) – Wrapper class to look for

Return type: bool

Returns: True if environment has been wrapped with wrapper_class.

stable_baselines3.common.env_util.make_atari_env (env_id, n_envs=1, seed=None, start_index=0, monitor_dir=None, wrapper_kwargs=None, env_kwargs=None, vec_env_cls=None, vec_env_kwargs=None, monitor_kwargs=None)

Create a wrapped, monitored VecEnv for Atari. It is a wrapper around make_vec_env that includes common preprocessing for Atari games.

Parameters
• **env_id** (Union[str, Type[Env]]) – the environment ID or the environment class
• **n_envs** (int) – the number of environments you wish to have in parallel
• **seed** (Optional[int]) – the initial seed for the random number generator
• **start_index** (int) – start rank index
• **monitor_dir** (Optional[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** (Optional[Dict[str, Any]]) – Optional keyword argument to pass to the AtariWrapper
• **env_kwargs** (Optional[Dict[str, Any]]) – Optional keyword argument to pass to the env constructor
• **vec_env_cls** (Union[DummyVecEnv, SubprocVecEnv, None]) – A custom VecEnv class constructor. Default: None.
• **vec_env_kwargs** (Optional[Dict[str, Any]]) – Keyword arguments to pass to the VecEnv class constructor.
• **monitor_kwargs** (Optional[Dict[str, Any]]) – Keyword arguments to pass to the Monitor class constructor.

**Return type** VecEnv

**Returns** The wrapped environment

```python
stable_baselines3.common.env_util.make_vec_env(env_id, n_envs=1, seed=None, start_index=0, monitor_dir=None, wrapper_class=None, env_kwargs=None, vec_env_cls=None, vec_env_kwargs=None)
```

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

**Parameters**

• **env_id** (Union[str, Type[Env]]) – the environment ID or the environment class
• **n_envs** (int) – the number of environments you wish to have in parallel
• **seed** (Optional[int]) – the initial seed for the random number generator
• **start_index** (int) – start rank index
• **monitor_dir** (Optional[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** (Optional[Callable[[Env], Env]]) – 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** (Optional[Dict[str, Any]]) – Optional keyword argument to pass to the env constructor
• **vec_env_cls** (Optional[Type[Union[DummyVecEnv, SubprocVecEnv]]]) – A custom VecEnv class constructor. Default: None.
• `vec_env_kwargs` (Optional[Dict[str, Any]]) – Keyword arguments to pass to the VecEnv class constructor.

• `monitor_kwargs` (Optional[Dict[str, Any]]) – Keyword arguments to pass to the Monitor class constructor.

**Return type** VecEnv

**Returns** The wrapped environment

```python
stable_baselines3.common.env_util.unwrap_wrapper(env, wrapper_class)
```

Retrieve a VecEnvWrapper object by recursively searching.

**Parameters**

• `env` (Env) – Environment to unwrap

• `wrapper_class` (Type[Wrapper]) – Wrapper to look for

**Return type** Optional[Wrapper]

**Returns** Environment unwrapped till `wrapper_class` if it has been wrapped with it

### 1.30 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.

```python
class stable_baselines3.common.distributions.BernoulliDistribution(action_dims)
```

Bernoulli distribution for MultiBinary action spaces.

**Parameters** `action_dim` – Number of binary actions

`actions_from_params` (action_logits, deterministic=False)

Returns samples from the probability distribution given its parameters.

**Return type** Tensor

**Returns** actions

`entropy` ()

Returns Shannon’s entropy of the probability

**Return type** Tensor

**Returns** the entropy, or None if no analytical form is known

1.30. Probability Distributions 141
log_prob \(\text{(actions)}\)

Returns the log likelihood

**Parameters**

- \(x\) – the taken action

**Return type** Tensor

**Returns** The log likelihood of the distribution

log_prob_from_params \(\text{(action_logits)}\)

Returns samples and the associated log probabilities from the probability distribution given its parameters.

**Return type** Tuple[Tensor, Tensor]

**Returns** actions and log prob

mode()

Returns the most likely action (deterministic output) from the probability distribution

**Return type** Tensor

**Returns** the stochastic action

proba_distribution \(\text{(action_logits)}\)

Set parameters of the distribution.

**Return type** BernoulliDistribution

**Returns** self

proba_distribution_net \(\text{(latent_dim)}\)

Create the layer that represents the distribution: it will be the logits of the Bernoulli distribution.

**Parameters**

- \(\text{latent_dim}\) (int) – Dimension of the last layer of the policy network (before the action layer)

**Return type** Module

**Returns**

sample()

Returns a sample from the probability distribution

**Return type** Tensor

**Returns** the stochastic action

class stable_baselines3.common.distributions.CategoricalDistribution \(\text{(action_dim)}\)

Categorical distribution for discrete actions.

**Parameters**

- \(\text{action_dim}\) (int) – Number of discrete actions

actions_from_params \(\text{(action_logits, deterministic=False)}\)

Returns samples from the probability distribution given its parameters.

**Return type** Tensor

**Returns** actions

entropy()

Returns Shannon’s entropy of the probability

**Return type** Tensor

**Returns** the entropy, or None if no analytical form is known

log_prob \(\text{(actions)}\)

Returns the log likelihood
Parameters \( x \) – the taken action

Return type Tensor

Returns The log likelihood of the distribution

**log_prob_from_params** \((\text{action_logits})\)

Returns samples and the associated log probabilities from the probability distribution given its parameters.

Return type Tuple[Tensor, Tensor]

Returns actions and log prob

**mode**()

Returns the most likely action (deterministic output) from the probability distribution

Return type Tensor

Returns the stochastic action

**proba_distribution** \((\text{action_logits})\)

Set parameters of the distribution.

Return type CategoricalDistribution

Returns self

**proba_distribution_net** \((\text{latent_dim})\)

Create the layer that represents the distribution: it will be the logits of the Categorical distribution. You can then get probabilities using a softmax.

Parameters latent_dim \((\text{int})\) – Dimension of the last layer of the policy network (before the action layer)

Return type Module

Returns

**sample**()

Returns a sample from the probability distribution

Return type Tensor

Returns the stochastic action

---

**class stable_baselines3.common.distributions.DiagGaussianDistribution** \((\text{action_dim})\)

Gaussian distribution with diagonal covariance matrix, for continuous actions.

Parameters action_dim \((\text{int})\) – Dimension of the action space.

**actions_from_params** \((\text{mean_actions, log_std, deterministic=False})\)

Returns samples from the probability distribution given its parameters.

Return type Tensor

Returns actions

**entropy**()

Returns Shannon’s entropy of the probability

Return type Tensor

Returns the entropy, or None if no analytical form is known

**log_prob** \((\text{actions})\)

Get the log probabilities of actions according to the distribution. Note that you must first call the proba_distribution() method.
Parameters `actions` (Tensor) –
Return type Tensor
Returns

`log_prob_from_params` *(mean_actions, log_std)*
Compute the log probability of taking an action given the distribution parameters.

Parameters
- `mean_actions` (Tensor) –
- `log_std` (Tensor) –
Return type Tuple[Tensor, Tensor]
Returns

`mode` ()
Returns the most likely action (deterministic output) from the probability distribution

Return type Tensor
Returns the stochastic action

`proba_distribution` *(mean_actions, log_std)*
Create the distribution given its parameters (mean, std)

Parameters
- `mean_actions` (Tensor) –
- `log_std` (Tensor) –
Return type `DiagGaussianDistribution`
Returns

`proba_distribution_net` *(latent_dim, log_std_init=0.0)*
Create the layers and parameter that represent the distribution: one output will be the mean of the Gaussian, the other parameter will be the standard deviation (log std in fact to allow negative values)

Parameters
- `latent_dim` (int) – Dimension of the last layer of the policy (before the action layer)
- `log_std_init` (float) – Initial value for the log standard deviation
Return type Tuple[Module, Parameter]
Returns

`sample` ()
Returns a sample from the probability distribution

Return type Tensor
Returns the stochastic action

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

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

Return type Tensor
Returns actions
abstract entropy()  
Returns Shannon’s entropy of the probability

    Return type  Optional[Tensor]  
    Returns  the entropy, or None if no analytical form is known

get_actions(deterministic=False)  
Return actions according to the probability distribution.

    Parameters  deterministic (bool) -  
    Return type  Tensor  
    Returns

abstract log_prob(x)  
Returns the log likelihood

    Parameters  x (Tensor) – the taken action  
    Return type  Tensor  
    Returns  The log likelihood of the distribution

abstract log_prob_from_params(*args, **kwargs)  
Returns samples and the associated log probabilities from the probability distribution given its parameters.

    Return type  Tuple[Tensor, Tensor]  
    Returns  actions and log prob

abstract mode()  
Returns the most likely action (deterministic output) from the probability distribution

    Return type  Tensor  
    Returns  the stochastic action

abstract proba_distribution(*args, **kwargs)  
Set parameters of the distribution.

    Return type  Distribution  
    Returns  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.

    Return type  Union[Module, Tuple[Module, Parameter]]

abstract sample()  
Returns a sample from the probability distribution

    Return type  Tensor  
    Returns  the stochastic action

class stable_baselines3.common.distributions.MultiCategoricalDistribution(action_dims)  
MultiCategorical distribution for multi discrete actions.

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

actions_from_params(action_logits, deterministic=False)  
Returns samples from the probability distribution given its parameters.
Return type Tensor
Returns actions

entropy()
Returns Shannon’s entropy of the probability

Return type Tensor
Returns the entropy, or None if no analytical form is known

log_prob(actions)
Returns the log likelihood

Parameters x – the taken action

Return type Tensor
Returns The log likelihood of the distribution

log_prob_from_params(action_logits)
Returns samples and the associated log probabilities from the probability distribution given its parameters.

Return type Tuple[Tensor, Tensor]
Returns actions and log prob

mode()
Returns the most likely action (deterministic output) from the probability distribution

Return type Tensor
Returns the stochastic action

proba_distribution(action_logits)
Set parameters of the distribution.

Return type MultiCategoricalDistribution
Returns self

proba_distribution_net(latent_dim)
Create the layer that represents the distribution: it will be the logits (flattened) of the MultiCategorical distribution. You can then get probabilities using a softmax on each sub-space.

Parameters latent_dim (int) – Dimension of the last layer of the policy network (before the action layer)

Return type Module
Returns

sample()
Returns a sample from the probability distribution

Return type Tensor
Returns the stochastic action

class stable_baselines3.common.distributions.SquashedDiagGaussianDistribution(action_dim, epsilon=1e-06)

Gaussian distribution with diagonal covariance matrix, followed by a squashing function (tanh) to ensure bounds.

Parameters

• action_dim (int) – Dimension of the action space.
• \textbf{epsilon} (float) – small value to avoid NaN due to numerical imprecision.

\textbf{entropy}()

Returns Shannon’s entropy of the probability

\begin{itemize}
    \item \textbf{Return type} Optional[Tensor]
    \item \textbf{Returns} the entropy, or None if no analytical form is known
\end{itemize}

\textbf{log_prob} (actions, gaussian_actions=\textit{None})

Get the log probabilities of actions according to the distribution. Note that you must first call the \texttt{proba_distribution()} method.

\begin{itemize}
    \item \textbf{Parameters} actions (Tensor)–
    \item \textbf{Return type} Tensor
    \item \textbf{Returns}
\end{itemize}

\textbf{log_prob_from_params} (mean_actions, log_std)

Compute the log probability of taking an action given the distribution parameters.

\begin{itemize}
    \item \textbf{Parameters}
        \begin{itemize}
            \item \textbf{mean_actions} (Tensor)–
            \item \textbf{log_std} (Tensor)–
        \end{itemize}
    \item \textbf{Return type} Tuple[Tensor, Tensor]
    \item \textbf{Returns}
\end{itemize}

\textbf{mode}()

Returns the most likely action (deterministic output) from the probability distribution

\begin{itemize}
    \item \textbf{Return type} Tensor
    \item \textbf{Returns} the stochastic action
\end{itemize}

\textbf{proba_distribution} (mean_actions, log_std)

Create the distribution given its parameters (mean, std)

\begin{itemize}
    \item \textbf{Parameters}
        \begin{itemize}
            \item \textbf{mean_actions} (Tensor)–
            \item \textbf{log_std} (Tensor)–
        \end{itemize}
    \item \textbf{Return type} \texttt{SquashedDiagGaussianDistribution}
    \item \textbf{Returns}
\end{itemize}

\textbf{sample}()

Returns a sample from the probability distribution

\begin{itemize}
    \item \textbf{Return type} Tensor
    \item \textbf{Returns} the stochastic action
\end{itemize}

\texttt{class stable_baselines3.common.distributions.StateDependentNoiseDistribution (action\_dim,}
\begin{itemize}
    \item \texttt{full\_std=}True,
    \item \texttt{use\_expln=}False,
    \item \texttt{squash\_output=}False,
    \item \texttt{learn\_features=}False,
    \item \texttt{epsilon=}1e-06
\end{itemize}


\textbf{1.30. Probability Distributions}
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 \((n\_features \times n\_actions)\) parameters for the std instead of only \((n\_features,)\).
- **use_expln** *(bool)* – Use `expln()` function instead of `exp()` to ensure a positive standard deviation (cf paper). It allows to keep variance above zero and prevent it from growing too fast. In practice, `exp()` is usually enough.
- **squash_output** *(bool)* – Whether to squash the output using a tanh function, this ensures bounds are satisfied.
- **learn_features** *(bool)* – Whether to learn features for gSDE or not. This will enable gradients to be backpropagated through the features `latent_sde` in the code.
- **epsilon** *(float)* – Small value to avoid NaN due to numerical imprecision.

**actions_from_params** *(mean_actions, log_std, latent_sde, deterministic=False)*

Returns samples from the probability distribution given its parameters.

**entropy**

Returns Shannon’s entropy of the probability

**get_std** *(log_std)*

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

**log_prob** *(actions)*

Returns the log likelihood

**log_prob_from_params** *(mean_actions, log_std, latent_sde)*

Returns samples and the associated log probabilities from the probability distribution given its parameters.

**mode**

Returns the most likely action (deterministic output) from the probability distribution
Returns the stochastic action

```python
proba_distribution(mean_actions, log_std, latent_sde)
```

Create the distribution given its parameters (mean, std)

Parameters

- `mean_actions` (Tensor)
- `log_std` (Tensor)
- `latent_sde` (Tensor)

Return type: `StateDependentNoiseDistribution`

Returns

```python
proba_distribution_net(latent_dim, log_std_init=-2.0, latent_sde_dim=None)
```

Create the layers and parameter that represent the distribution: one output will be the deterministic action, the other parameter will be the standard deviation of the distribution that control the weights of the noise matrix.

Parameters

- `latent_dim` (int) – Dimension of the last layer of the policy (before the action layer)
- `log_std_init` (float) – Initial value for the log standard deviation
- `latent_sde_dim` (Optional[int]) – Dimension of the last layer of the features extractor for gSDE. By default, it is shared with the policy network.

Return type: `Tuple[Module, Parameter]`

Returns

```python
sample()
```

Returns a sample from the probability distribution

Return type: `Tensor`

Returns the stochastic action

```python
sample_weights(log_std, batch_size=1)
```

Sample weights for the noise exploration matrix, using a centered Gaussian distribution.

Parameters

- `log_std` (Tensor)
- `batch_size` (int)

Return type: `None`

```python
class stable_baselines3.common.distributions.TanhBijector(epsilon=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.

```python
static atanh(x)
```

Inverse of Tanh

Taken from pyro: https://github.com/pyro-ppl/pyro

0.5 * torch.log((1 + x) / (1 - x))

Return type: `Tensor`

```python
static inverse(y)
```

Inverse tanh.

1.30. Probability Distributions
Parameters \( y \) (Tensor) –

Return type Tensor

Returns

\[
\text{stable_baselines3.common.distributions.make_proba_distribution}(\text{action_space}, \text{use_sde}=\text{False}, \text{dist_kwargs}=\text{None})
\]

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

Parameters

- \( \text{action_space} \) (Space) – the input action space
- \( \text{use_sde} \) (bool) – Force the use of StateDependentNoiseDistribution instead of Diag-GaussianDistribution
- \( \text{dist_kwargs} \) (Optional[Dict[str, Any]]) – Keyword arguments to pass to the probability distribution

Return type Distribution

Returns the appropriate Distribution object

\[
\text{stable_baselines3.common.distributions.sum_independent_dims}(\text{tensor})
\]

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

Parameters \( \text{tensor} \) (Tensor) – shape: (n_batch, n_actions) or (n_batch,)

Return type Tensor

Returns shape: (n_batch,)

### 1.31 Evaluation Helper

\[
\text{stable_baselines3.common.evaluation.evaluate_policy}(\text{model}, \text{env}, \text{n_eval_episodes}=10, \text{deterministic}=\text{True}, \text{render}=\text{False}, \text{callback}=\text{None}, \text{reward_threshold}=\text{None}, \text{return_episode_rewards}=\text{False}, \text{warn}=\text{True})
\]

Runs policy for \( n\_\text{eval\_episodes} \) episodes and returns average reward. This is made to work only with one env.

**Note:** If environment has not been wrapped with Monitor wrapper, reward and episode lengths are counted as it appears with env.step calls. If the environment contains wrappers that modify rewards or episode lengths (e.g. reward scaling, early episode reset), these will affect the evaluation results as well. You can avoid this by wrapping environment with Monitor wrapper before anything else.

Parameters

- \( \text{model} \) (BaseAlgorithm) – The RL agent you want to evaluate.
- \( \text{env} \) (Union[Env, VecEnv]) – The gym environment. In the case of a VecEnv this must contain only one environment.
- \( \text{n\_eval\_episodes} \) (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** (Optional[Callable[[Dict[str, Any], Dict[str, Any]], None]]) – callback function to do additional checks, called after each step. Gets locals() and globals() passed as parameters.
- **reward_threshold** (Optional[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 rewards and episode lengths per episode will be returned instead of the mean.
- **warn** (bool) – If True (default), warns user about lack of a Monitor wrapper in the evaluation environment.

**Return type** Union[Tuple[float, float], Tuple[List[float], List[int]]]

**Returns** Mean reward per episode, std of reward per episode. Returns ([float], [int]) when `return_episode_rewards` is True, first list containing per-episode rewards and second containing per-episode lengths (in number of steps).

### 1.32 Gym Environment Checker

`stable_baselines3.common.env_checker.check_env(env, warn=True, skip_render_check=True)`

Check that an environment follows Gym API. This is particularly useful when using a custom environment. Please take a look at https://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** (Env) – The Gym environment that will be checked
- **warn** (bool) – Whether to output additional warnings mainly related to the interaction with Stable Baselines
- **skip_render_check** (bool) – Whether to skip the checks for the render method. True by default (useful for the CI)

**Return type** None

### 1.33 Monitor Wrapper

`class stable_baselines3.common.monitor.Monitor(env, filename=None, allow_early_resets=True, reset_keywords=(), info_keywords=())`

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

**Parameters**

- **env** (Env) – The environment
- **filename** (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

    Return type None

**get_episode_lengths**()
Returns the number of timesteps of all the episodes

    Return type List[int]

**get_episode_rewards**()
Returns the rewards of all the episodes

    Return type List[float]

**get_episode_times**()
Returns the runtime in seconds of all the episodes

    Return type List[float]

**get_total_steps**()
Returns the total number of timesteps

    Return type int

**reset**(**kwargs**)
Calls the Gym environment reset. Can only be called if the environment is over, or if allow_early_resets is True

    Parameters **kwargs – Extra keywords saved for the next episode. only if defined by reset_keywords

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

**step**(action)
Step the environment with the given action

    Parameters action (Union[ndarray, int]) – the action

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

    Returns observation, reward, done, information

**stable_baselines3.common.monitor.get_monitor_files**(path)
get all the monitor files in the given path

    Parameters path (str) – the logging folder

    Return type List[str]
Returns the log files

```
stable_baselines3.common.monitor.load_results(path)
```

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

Parameters

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

Return type DataFrame

Returns the logged data

### 1.34 Logger

```
class stable_baselines3.common.logger.CSVOutputFormat(filename)
```

close()

closes the file

Return type None

```
write(key_values, key_excluded, step=0)
```

Write a dictionary to file

Parameters

- **key_values** (Dict[str, Any]) –
- **key_excluded** (Dict[str, Union[str, Tuple[str, ...]]]) –
- **step** (int) –

Return type None

```
class stable_baselines3.common.logger.Figure(figure, close)
```

Figure data class storing a matplotlib figure and whether to close the figure after logging it

Parameters

- **figure** (figure) – figure to log
- **close** (bool) – if true, close the figure after logging it

Exception

```
exception stable_baselines3.common.logger.FormatUnsupportedError(unsupported_formats, value_description)
```

```
class stable_baselines3.common.logger.HumanOutputFormat(filename_or_file)
```

close()

closes the file

Return type None

```
write(key_values, key_excluded, step=0)
```

Write a dictionary to file

Parameters

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

---

1.34. Logger 153
Return type None

write_sequence(sequence)
write_sequence an array to file

Parameters sequence (List) –

Return type None

class stable_baselines3.common.logger.Image(image, dataformats)
Image data class storing an image and data format

Parameters

• image (Union[Tensor, ndarray, str]) – image to log

• dataformats (str) – Image data format specification of the form NCHW, NHWC, CHW, HWC, HW, WH, etc. More info in add_image method doc at https://pytorch.org/docs/stable/tensorboard.html Gym envs normally use ‘HWC’ (channel last)

class stable_baselines3.common.logger.JSONOutputFormat(filename)

close()
closes the file

Return type None

write(key_values, key_excluded, step=0)
Write a dictionary to file

Parameters

• key_values (Dict[str, Any]) –

• key_excluded (Dict[str, Union[str, Tuple[str, ...]]]) –

• step (int) –

Return type None

class stable_baselines3.common.logger.KVWriter
Key Value writer

close()
Close owned resources

Return type None

write(key_values, key_excluded, step=0)
Write a dictionary to file

Parameters

• key_values (Dict[str, Any]) –

• key_excluded (Dict[str, Union[str, Tuple[str, ...]]]) –

• step (int) –

Return type None

class stable_baselines3.common.logger.SeqWriter
sequence writer

write_sequence(sequence)
write_sequence an array to file
Parameters `sequence` (List) –

Return type None

class `stable_baselines3.common.logger.TensorBoardOutputFormat` (`folder`)

close()
closes the file

Return type None
write (`key_values`, `key_excluded`, `step=0`)
Write a dictionary to file

Parameters
- `key_values` (Dict[str, Any]) –
- `key_excluded` (Dict[str, Union[str, Tuple[str, ...]]) –
- `step` (int) –

Return type None

class `stable_baselines3.common.logger.Video` (`frames`, `fps`)
Video data class storing the video frames and the frame per seconds

Parameters
- `frames` (Tensor) – frames to create the video from
- `fps` (Union[float, int]) – frames per second

stable_baselines3.common.logger.configure (`folder=None, format_strings=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'])

Return type None

stable_baselines3.common.logger.debug (`*args`)  
Write the sequence of args, with no separators, to the console and output files (if you’ve configured an output file). Using the DEBUG level.

Parameters `args` – log the arguments

Return type None

stable_baselines3.common.logger.dump (`step=0`)  
Write all of the diagnostics from the current iteration

Return type None

stable_baselines3.common.logger.dump_tabular (`step=0`)  
Write all of the diagnostics from the current iteration

Return type None
stable_baselines3.common.logger.error(*args)
Write the sequence of args, with no separators, to the console and output files (if you’ve configured an output file). Using the ERROR level.

Parameters
- **args**: log the arguments

Return type: None

stable_baselines3.common.logger.filter_excluded_keys(key_values, key_excluded, _format)
Filters the keys specified by `key_excluded` for the specified format

Parameters
- **key_values**: `Dict[str, Any]` – log dictionary to be filtered
- **key_excluded**: `Dict[str, Union[str, Tuple[str, ...]]]` – keys to be excluded per format
- **_format**: `str` – format for which this filter is run

Return type: `Dict[str, Any]`

Returns: dict without the excluded keys

stable_baselines3.common.logger.get_dir()
Get directory that log files are being written to. will be None if there is no output directory (i.e., if you didn’t call start)

Return type: `str`

Returns: the logging directory

stable_baselines3.common.logger.get_level()
Get logging threshold on current logger.

Return type: `int`

Return: the logging level (can be DEBUG=10, INFO=20, WARN=30, ERROR=40, DISABLED=50)

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

Return type: `Dict`

Returns: the logged values

stable_baselines3.common.logger.info(*args)
Write the sequence of args, with no separators, to the console and output files (if you’ve configured an output file). Using the INFO level.

Parameters
- **args**: log the arguments

Return type: None

stable_baselines3.common.logger.log(*args, level=20)
Write the sequence of args, with no separators, to the console and output files (if you’ve configured an output file).

Parameters
- **args**: log the arguments

- **level**: int. (see logger.py docs) If the global logger level is higher than the level argument here, don’t print to stdout.
Return type  None

stable_baselines3.common.logger.make_output_format(_format, log_dir, log_suffix="")

return a logger for the requested format

Parameters

• _format (str) – the requested format to log to (‘stdout’, ‘log’, ‘json’ or ‘csv’ or ‘tensorboard’)
• log_dir (str) – the logging directory
• log_suffix (str) – the suffix for the log file

Return type  KVWriter

Returns  the logger

stable_baselines3.common.logger.read_csv(filename)

read a csv file using pandas

Parameters  filename (str) – the file path to read

Return type  DataFrame

Returns  the data in the csv

stable_baselines3.common.logger.read_json(filename)

read a json file using pandas

Parameters  filename (str) – the file path to read

Return type  DataFrame

Returns  the data in the json

stable_baselines3.common.logger.record(key, value, exclude=None)

Log a value of some diagnostic Call this once for each diagnostic quantity, each iteration If called many times, last value will be used.

Parameters

• key (str) – save to log this key
• value (Any) – save to log this value
• exclude (Union[str, Tuple[str, ...], None]) – outputs to be excluded

Return type  None

stable_baselines3.common.logger.record_dict(key_values)

Log a dictionary of key-value pairs.

Parameters  key_values (Dict[str, Any]) – the list of keys and values to save to log

Return type  None

stable_baselines3.common.logger.record_mean(key, value, exclude=None)

The same as record(), but if called many times, values averaged.

Parameters

• key (str) – save to log this key
• value (Union[int, float]) – save to log this value
• exclude (Union[str, Tuple[str, ...], None]) – outputs to be excluded
stable_baselines3.common.logger.\texttt{record_tabular}(key, value, exclude=None)

Log a value of some diagnostic Call this once for each diagnostic quantity, each iteration If called many times, last value will be used.

**Parameters**
- \texttt{key} (str) – save to log this key
- \texttt{value} (Any) – save to log this value
- \texttt{exclude} (Union[str, Tuple[str, ...], None]) – outputs to be excluded

stable_baselines3.common.logger.\texttt{reset}()

reset the current logger

stable_baselines3.common.logger.\texttt{set_level}(level)

Set logging threshold on current logger.

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

stable_baselines3.common.logger.\texttt{warn}(*args)

Write the sequence of args, with no separators, to the console and output files (if you’ve configured an output file). Using the WARN level.

**Parameters**
- \texttt{args} – log the arguments

1.35 Action Noise

\texttt{class} stable_baselines3.common.noise.\texttt{ActionNoise}

The action noise base class

\texttt{reset}()

call end of episode reset for the noise

\texttt{Return type} None

\texttt{class} stable_baselines3.common.noise.\texttt{NormalActionNoise}(mean, sigma)

A Gaussian action noise

**Parameters**
- \texttt{mean} (ndarray) – the mean value of the noise
- \texttt{sigma} (ndarray) – the scale of the noise (std here)

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

**reset** ()
reset the Ornstein Uhlenbeck noise, to the initial position

Return type None

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

Parameters

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

**reset** (indices=None)
Reset all the noise processes, or those listed in indices

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

Return type None

1.36 Utils

stable_baselines3.common.utils.check_for_correct_spaces(env, observation_space, action_space)
Checks that the environment has same spaces as provided ones. Used by BaseAlgorithm to check if spaces match after loading the model with given env. Checked parameters: - observation_space - action_space

Parameters

• **env** *(Union[Env, VecEnv])* – Environment to check for valid spaces
• **observation_space** *(Space)* – Observation space to check against
• **action_space** *(Space)* – Action space to check against

Return type None

stable_baselines3.common.utils.configure_logger(verbos=0, tensorboard_log=None, tb_log_name=", reset_num_timesteps=True)
Configure the logger’s outputs.

Parameters

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

Return type None

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

Parameters val (float) –
Return type Callable[[float], float]

Return

stable_baselines3.common.utils.explained_variance(y_pred, y_true)
Computes fraction of variance that ypred explains about y. Returns 1 - Var[y-ypred] / Var[y]

interpretation: ev=0 => might as well have predicted zero ev=1 => perfect prediction ev<0 => worse than just predicting zero

Parameters
• y_pred (ndarray) – the prediction
• y_true (ndarray) – the expected value

Return type ndarray

Returns explained variance of ypred and y

stable_baselines3.common.utils.get_device(device='auto')
Retrieve PyTorch device. It checks that the requested device is available first. For now, it supports only cpu and cuda. By default, it tries to use the gpu.

Parameters device (Union[device, str]) = One for ‘auto’, ‘cuda’, ‘cpu’

Return type device

Returns

stable_baselines3.common.utils.get_latest_run_id(log_path=None, log_name='')
Returns the latest run number for the given log name and log path, by finding the greatest number in the directories.

Return type int

Returns latest run number

stable_baselines3.common.utils.get_linear_fn(start, end, end_fraction)
Create a function that interpolates linearly between start and end between progress_remaining = 1 and progress_remaining = end_fraction. This is used in DQN for linearly annealing the exploration fraction (epsilon for the epsilon-greedy strategy).

Params start value to start with if progress_remaining = 1
Params end value to end with if progress_remaining = 0
Params end_fraction fraction of progress_remaining where end is reached e.g 0.1 then end is reached after 10% of the complete training process.

Return type Callable[[float], float]

Returns
stable_baselines3.common.utils.get_schedule_fn(value_schedule)
Transform (if needed) learning rate and clip range (for PPO) to callable.

Parameters

value_schedule (Union[Callable[[float], float], float, int]) –

Returns

Callable[[float], float]

stable_baselines3.common.utils.is_vectorized_observation(observation, observation_space)
For every observation type, detects and validates the shape, then returns whether or not the observation is vectorized.

Parameters

• observation (ndarray) – the input observation to validate
• observation_space (Space) – the observation space

Returns

bool

whether the given observation is vectorized or not

stable_baselines3.common.utils.polyak_update(params, target_params, tau)
Perform a Polyak average update on target_params using params: target parameters are slowly updated towards the main parameters. tau, the soft update coefficient controls the interpolation: tau=1 corresponds to copying the parameters to the target ones whereas nothing happens when tau=0. The Polyak update is done in place, with no_grad, and therefore does not create intermediate tensors, or a computation graph, reducing memory cost and improving performance. We scale the target params by 1−tau (in-place), add the new weights, scaled by tau and store the result of the sum in the target params (in place). See https://github.com/DLR-RM/stable-baselines3/issues/93

Parameters

• params (Iterable[Parameter]) – parameters to use to update the target params
• target_params (Iterable[Parameter]) – parameters to update
• tau (float) – the soft update coefficient (“Polyak update”, between 0 and 1)

Returns

None

stable_baselines3.common.utils.safe_mean(arr)
Compute the mean of an array if there is at least one element. For empty array, return NaN. It is used for logging only.

Parameters

arr (Union[ndarray, list, deque]) –

Returns

ndarray

stable_baselines3.common.utils.set_random_seed(seed, using_cuda=False)
Seed the different random generators.

Parameters

• seed : int
• using_cuda : bool

Returns

None

stable_baselines3.common.utils.update_learning_rate(optimizer, learning_rate)
Update the learning rate for a given optimizer. Useful when doing linear schedule.

Parameters

• optimizer (Optimizer) –
• learning_rate (float) –
Return type: None

\[
\text{stable_baselines3.common.utils.zip\_strict(*iterables)}
\]

The \text{zip()} function but enforces that iterables are of equal length. Raises \text{ValueError} if iterables not of equal length. Code inspired by Stackoverflow answer for question #32954486.

**Parameters**

*iterables* – iterables to \text{zip()}

**Return type**

Iterable

## 1.37 Changelog

### 1.37.1 Pre-Release 0.11.0a5 (WIP)

**Breaking Changes:**

- \text{evaluate\_policy} now returns rewards/episode lengths from a \text{Monitor} wrapper if one is present, this allows to return the unnormalized reward in the case of Atari games for instance.

- Renamed \text{common.vec\_env.is\_wrapped} to \text{common.vec\_env.is\_vecenv\_wrapped} to avoid confusion with the new \text{is\_wrapped()} helper

**New Features:**

- Add support for \text{VecFrameStack} to stack on first or last observation dimension, along with automatic check for image spaces.

- \text{VecFrameStack} now has a \text{channels\_order} argument to tell if observations should be stacked on the first or last observation dimension (originally always stacked on last).

- Added \text{common.env\_util.is\_wrapped} and \text{common.env\_util.unwrap\_wrapper} functions for checking/unwrapping an environment for specific wrapper.

- Added \text{env\_is\_wrapped()} method for \text{VecEnv} to check if its environments are wrapped with given Gym wrappers.

- Added \text{monitor\_kwargs} parameter to \text{make\_vec\_env} and \text{make\_atari\_env}

- Wrap the environments automatically with a \text{Monitor} wrapper when possible.

- \text{EvalCallback} now logs the success rate when available (\text{is\_success} must be present in the info dict)

- Added new wrappers to log images and matplotlib figures to tensorboard. (@zampanteymedio)

**Bug Fixes:**

- Fixed bug where code added VecTranspose on channel-first image environments (thanks @qxcv)

- Fixed DQN predict method when using single \text{gym.Env} with deterministic=False

- Fixed bug that the arguments order of \text{explained\_variance()} in \text{ppo.py} and \text{a2c.py} is not correct (@thisray)

- Fixed bug where full \text{HerReplayBuffer} leads to an index error. (@megan-klaiber)

- Fixed bug where replay buffer could not be saved if it was too big (> 4 Gb) for python<3.8 (thanks @hn2)
• Added informative PPO construction error in edge-case scenario where $n_{steps} \times n_{envs} = 1$ (size of rollout buffer), which otherwise causes downstream breaking errors in training (@decodyng)

Deprecations:

Others:

• Add more issue templates
• Add signatures to callable type annotations (@ernestum)
• Improve error message in NatureCNN
• Added checks for supported action spaces to improve clarity of error messages for the user
• Renamed variables in the $train()$ method of SAC, TD3 and DQN to match SB3-Contrib.
• Updated docker base image to Ubuntu 18.04
• Set tensorboard min version to 2.2.0 (earlier version are apparently not working with PyTorch)
• Added warning for PPO when $n_{steps} \times n_{envs}$ is not a multiple of $batch\_size$ (last mini-batch truncated) (@decodyng)

Documentation:

• Updated algorithm table
• Minor docstring improvements regarding rollout (@stheid)
• Fix migration doc for A2C (epsilon parameter)
• Fix $clip\_range$ docstring
• Fix duplicated parameter in EvalCallback docstring (thanks @tfederico)
• Added example of learning rate schedule
• Added SUMO-RL as example project (@LucasAlegre)
• Fix docstring of classes in atari_wrappers.py which were inside the constructor (@LucasAlegre)
• Added SB3-Contrib page
• Fix bug in the example code of DQN (@AptX395)

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

HER with online and offline sampling, bug fixes for features extraction
Stable Baselines3 Documentation, Release 0.11.0a5

Breaking Changes:

- **Warning:** Renamed `common.cmd_util` to `common.env_util` for clarity (affects `make_vec_env` and `make_atari_env` functions)

New Features:

- Allow custom actor/critic network architectures using `net_arch=dict(qf=[400, 300], pi=[64, 64])` for off-policy algorithms (SAC, TD3, DDPG)
- Added Hindsight Experience Replay HER (@megan-klaiber)
- VecNormalize now supports gym.spaces.Dict observation spaces
- Support logging videos to Tensorboard (@SwamyDev)
- Added `share_features_extractor` argument to SAC and TD3 policies

Bug Fixes:

- Fix GAE computation for on-policy algorithms (off-by one for the last value) (thanks @Wovchena)
- Fixed potential issue when loading a different environment
- Fix ignoring the exclude parameter when recording logs using json, csv or log as logging format (@SwamyDev)
- Make `make_vec_env` support the `env_kwargs` argument when using an env ID str (@ManifoldFR)
- Fix model creation initializing CUDA even when `device="cpu"` is provided
- Fix `check_env` not checking if the env has a Dict actionspace before calling `_check_nan` (@wmmc88)
- Update the check for spaces unsupported by Stable Baselines 3 to include checks on the action space (@wmmc88)
- Fixed feature extractor bug for target network where the same net was shared instead of being separate. This bug affects SAC, DDPG and TD3 when using CnnPolicy (or custom feature extractor)
- Fixed a bug when passing an environment when loading a saved model with a CnnPolicy, the passed env was not wrapped properly (the bug was introduced when implementing HER so it should not be present in previous versions)

Deprecations:

Others:

- Improved typing coverage
- Improved error messages for unsupported spaces
- Added `.vscode` to the gitignore
Documentation:

- Added first draft of migration guide
- Added intro to imitation library (@shwang)
- Enabled doc for CnnPolicies
- Added advanced saving and loading example
- Added base doc for exporting models
- Added example for getting and setting model parameters

1.37.3 Pre-Release 0.9.0 (2020-10-03)

Bug fixes, get/set parameters and improved docs

Breaking Changes:

- Removed device keyword argument of policies; use `policy.to(device)` instead. (@qxcv)
- Rename `BaseClass.get_torch_variables` -> `BaseClass._get_torch_save_params` and `BaseClass.excluded_save_params` -> `BaseClass._excluded_save_params`
- Renamed saved items tensors to pytorch_variables for clarity
- `make_atari_env`, `make_vec_env` and `set_random_seed` must be imported with (and not directly from `stable_baselines3.common`):

```python
from stable_baselines3.common.cmd_util import make_atari_env, make_vec_env
from stable_baselines3.common.utils import set_random_seed
```

New Features:

- Added `unwrap_vec_wrapper()` to `common.vec_env` to extract VecEnvWrapper if needed
- Added `StopTrainingOnMaxEpisodes` to callback collection (@xicocaio)
- Added `device` keyword argument to `BaseAlgorithm.load()` (@liorcohen5)
- Callbacks have access to rollout collection locals as in SB2. (@PartiallyTyped)
- Added `get_parameters` and `set_parameters` for accessing/setting parameters of the agent
- Added actor/critic loss logging for TD3. (@mloo3)

Bug Fixes:

- Added `unwrap_vec_wrapper()` to `common.vec_env` to extract VecEnvWrapper if needed
- Fixed a bug where the environment was reset twice when using `evaluate_policy`
- Fix logging of `clip_fraction` in PPO (@diditforlulz273)
- Fixed a bug where cuda support was wrongly checked when passing the GPU index, e.g., `device="cuda:0"` (@liorcohen5)
- Fixed a bug when the random seed was not properly set on cuda when passing the GPU index
Deprecations:

Others:

- Improve typing coverage of the VecEnv
- Fix type annotation of \texttt{make_vec_env}@\texttt{ManifoldFR}
- Removed \texttt{AlreadySteppingError} and \texttt{NotSteppingError} that were not used
- Fixed typos in SAC and TD3
- Reorganized functions for clarity in BaseClass (save/load functions close to each other, private functions at top)
- Clarified docstrings on what is saved and loaded to/from files
- Simplified \texttt{save_to_zip_file} function by removing duplicate code
- Store library version along with the saved models
- DQN loss is now logged

Documentation:

- Added \texttt{StopTrainingOnMaxEpisodes} details and example (@xicocaio)
- Updated custom policy section (added custom feature extractor example)
- Re-enable \texttt{sphinx\_autodoc\_typehints}
- Updated doc style for type hints and remove duplicated type hints

1.37.4 Pre-Release 0.8.0 (2020-08-03)

DQN, DDPG, bug fixes and performance matching for Atari games

Breaking Changes:

- \texttt{AtariWrapper} and other Atari wrappers were updated to match SB2 ones
- \texttt{save\_replay\_buffer} now receives as argument the file path instead of the folder path (@tirafesi)
- Refactored \texttt{Critic} class for TD3 and SAC, it is now called \texttt{ContinuousCritic} and has an additional parameter \texttt{n\_critics}
- \texttt{SAC} and \texttt{TD3} now accept an arbitrary number of critics (e.g. \texttt{policy\_kwargs=\texttt{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 QN 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)

1.37.5 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

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

Tensorboard support, refactored logger

Breaking Changes:

- Remove State-Dependent Exploration (SDE) support for TD3
- Methods were renamed in the logger:
  - `logkv` -> `record`, `writekvs` -> `write`, `writeseq` -> `write_sequence`,
  - `logkvs` -> `record_dict`, `dumpkvs` -> `dump`,
  - `getkvs` -> `get_log_dict`, `logkv_mean` -> `record_mean`,

1.37. Changelog
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

1.37.7 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 customize 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:

1.37.8 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

1.37.9 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

1.37.10 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

1.37.11 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
1.37.12 Maintainers

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

1.37.13 Contributors:

In random order . . .

Thanks to the maintainers of V2: @hill-a @enerijunior @AdamGleave @Miffyli
And all the contributors: @bjmuld @iambenzo @ianandforth @r7vme @brendenpetersen @huvar @abhiskk @JohannesAck @EliasHasle @mrakgr @Bleyddyne @antoine-galataud @junhyeokahn @AdamGleave @keshaviyengar @tperol @XMaster96 @kantneel @Pastafarianist @GerardMaggiolino @PatrickWalter214 @yutingsz @sc420 @Aaahh @biltubbs @Miffyli @dwiel @miguelrass @qxcv @jaberkow @eavelardev @ruifeng96150 @pedrohbtb @srivatsankrishnan @evilsocket @MarvineGothic @jdossgollin @stheid @SyllogismRXS @rusu24edward @jbulow @Antynyon @seheevic @justinkerry @edbeeching @flodorner @KuKuXia @NeoExtended @PartiallyTyped @mmcenta @richardwu @kinalmehta @rolandgvc @tkelestemur @mlloo3 @tirafesi @blurLake @koulakis @joeljosephjin @shwang @rk37 @andyshil12 @RaphaelWag @xicocaio @didiforlulz273 @liorcohen5 @ManifoldFR @mlloo3 @SwamyDev @wmmch88 @megan-klaiber @thisray @tfederico @hn2 @LucasAlegre @AptX395 @zampanteymedio @decodyng

1.38 Projects

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

1.38.1 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

1.38.2 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
1.38.3 SUMO-RL

A simple interface to instantiate RL environments with SUMO for Traffic Signal Control.

- Supports Multiagent RL
- Compatibility with gym.Env and popular RL libraries such as stable-baselines3 and RLlib
- Easy customisation: state and reward definitions are easily modifiable

Author: Lucas Alegre
Github: https://github.com/LucasAlegre/sumo-rl
CITING STABLE BASELINES3

To cite this project in publications:

```latex
@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}},
}
```
To any interested in making the rl baselines better, there are still some improvements that need to be done. You can check issues in the repo.

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

• genindex
• search
• modindex
<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>stable_baselines3.a2c</td>
<td>76</td>
</tr>
<tr>
<td>stable_baselines3.common.atari_wrappers</td>
<td>136</td>
</tr>
<tr>
<td>stable_baselines3.common.base_class</td>
<td>66</td>
</tr>
<tr>
<td>stable_baselines3.common.callbacks</td>
<td>44</td>
</tr>
<tr>
<td>stable_baselines3.common.distributions</td>
<td>141</td>
</tr>
<tr>
<td>stable_baselines3.common.env_checker</td>
<td>151</td>
</tr>
<tr>
<td>stable_baselines3.common.env_util</td>
<td>139</td>
</tr>
<tr>
<td>stable_baselines3.common.evaluation</td>
<td>150</td>
</tr>
<tr>
<td>stable_baselines3.common.logger</td>
<td>153</td>
</tr>
<tr>
<td>stable_baselines3.common.monitor</td>
<td>151</td>
</tr>
<tr>
<td>stable_baselines3.common.noise</td>
<td>158</td>
</tr>
<tr>
<td>stable_baselines3.common.off_policy_algorithm</td>
<td>69</td>
</tr>
<tr>
<td>stable_baselines3.common.on_policy_algorithm</td>
<td>73</td>
</tr>
<tr>
<td>stable_baselines3.common.utils</td>
<td>159</td>
</tr>
<tr>
<td>stable_baselines3.common.vec_env</td>
<td>23</td>
</tr>
<tr>
<td>stable_baselines3.ddpg</td>
<td>85</td>
</tr>
<tr>
<td>stable_baselines3.dqn</td>
<td>93</td>
</tr>
<tr>
<td>stable_baselines3.her</td>
<td>100</td>
</tr>
<tr>
<td>stable_baselines3.ppo</td>
<td>109</td>
</tr>
<tr>
<td>stable_baselines3.sac</td>
<td>119</td>
</tr>
<tr>
<td>stable_baselines3.td3</td>
<td>128</td>
</tr>
</tbody>
</table>
INDEX

A

A2C (class in stable_baselines3.a2c), 78
ActionNoise (class in stable_baselines3.common.noise), 158
actions_from_params() (stable_baselines3.common.distributions.BernoulliDistribution method), 141
actions_from_params() (stable_baselines3.common.distributions.CategoricalDistribution method), 142
actions_from_params() (stable_baselines3.common.distributions.DiagGaussianDistribution method), 143
actions_from_params() (stable_baselines3.common.distributions.Distribution method), 144
actions_from_params() (stable_baselines3.common.distributions.MultiCategoricalDistribution method), 145
actions_from_params() (stable_baselines3.common.distributions.StateDependentNoiseDistribution method), 148
add() (stable_baselines3.her.HerReplayBuffer method), 108
atanh() (stable_baselines3.common.distributions.TanhBijector static method), 149
AtariWrapper (class in stable_baselines3.common.atari_wrappers), 136

B

BaseAlgorithm (class in stable_baselines3.common.base_class), 66
BaseCallback (class in stable_baselines3.common.callbacks), 44
BernoulliDistribution (class in stable_baselines3.common.distributions), 141

C

CallbackList (class in stable_baselines3.common.callbacks), 45
CategoricalDistribution (class in stable_baselines3.common.distributions), 142
check_env() (in module stable_baselines3.common.env_checker), 151
check_for_correct_spaces() (in module stable_baselines3.common.utils), 159
CheckpointCallback (class in stable_baselines3.common.callbacks), 45
ClippedEnv (class in stable_baselines3.common.atari_wrappers), 137
close() (stable_baselines3.common.logger.CSVOutputFormat method), 153
close() (stable_baselines3.common.logger.HumanOutputFormat method), 153
close() (stable_baselines3.common.logger.JSONOutputFormat method), 154
close() (stable_baselines3.common.logger.KVWriter method), 154
close() (stable_baselines3.common.logger.TensorBoardOutputFormat method), 155
close() (stable_baselines3.common.monitor.Monitor method), 152
close() (stable_baselines3.common.vec_env.DummyVecEnv method), 26
close() (stable_baselines3.common.vec_env.SubprocVecEnv method), 28
close() (stable_baselines3.common.vec_env.VecEnv method), 24
close() (stable_baselines3.common.vec_env.VecFrameStack method), 29
close() (stable_baselines3.common.vec_env.VecTransposeImage method), 33
close() (stable_baselines3.common.vec_env.VecVideoRecorder method), 31
close() (stable_baselines3.her.ObsDicWrapper method), 105
CnnPolicy (class in stable_baselines3.ddpg), 92
CnnPolicy (class in stable_baselines3.dqn), 100
CnnPolicy (class in stable_baselines3.sac), 127
CnnPolicy (class in stable_baselines3.td3), 136
CnnPolicy (in module stable_baselines3.a2c), 83
filter_excluded_keys() (in module stable_baselines3.common.logger), 156
FireResetEnv (class in stable_baselines3.common.atari_wrappers), 138
FormatUnsupportedError, 153
G
get_actions() (stable_baselines3.common.distributions.Distribution method), 145
get_attr() (stable_baselines3.common.vec_env.DummyVecEnv method), 26
get_attr() (stable_baselines3.common.vec_env.SubprocVecEnv method), 28
get_attr() (stable_baselines3.common.vec_env.VecEnv method), 25
get_attr() (stable_baselines3.her.ObsDictWrapper method), 106
get_device() (in module stable_baselines3.common.utils), 160
get_dir() (in module stable_baselines3.common.logger), 156
get_env() (stable_baselines3.a2c.A2C method), 79
get_env() (stable_baselines3.common.base_class.BaseAlgorithm method), 67
get_env() (stable_baselines3.ddpg.DDPG method), 89
get_env() (stable_baselines3.dqn.DQN method), 96
get_env() (stable_baselines3.ppo.PPO method), 113
get_env() (stable_baselines3.sac.SAC method), 123
get_env() (stable_baselines3.td3.TD3 method), 132
get_episode_lengths() (stable_baselines3.common.monitor.Monitor method), 152
get_episode_rewards() (stable_baselines3.common.monitor.Monitor method), 152
get_episode_times() (stable_baselines3.common.monitor.Monitor method), 152
get_images() (stable_baselines3.common.vec_env.DummyVecEnv method), 27
get_images() (stable_baselines3.common.vec_env.SubprocVecEnv method), 28
get_images() (stable_baselines3.common.vec_env.VecEnv method), 25
get_images() (stable_baselines3.her.ObsDictWrapper method), 106
get_latest_run_id() (in module stable_baselines3.common.utils), 160
get_level() (in module stable_baselines3.common.logger), 156
get_linear_fn() (in module stable_baselines3.common.utils), 160
get_log_dict() (in module stable_baselines3.common.logger), 156
get_monitor_files() (in module stable_baselines3.common.monitor), 152
get_original_obs() (stable_baselines3.common.vec_env.VecNormalize method), 30
get_original_reward() (stable_baselines3.common.vec_env.VecNormalize method), 30
get_parameters() (stable_baselines3.a2c.A2C method), 79
get_parameters() (stable_baselines3.common.base_class.BaseAlgorithm method), 67
get_parameters() (stable_baselines3.ddpg.DDPG method), 89
get_parameters() (stable_baselines3.dqn.DQN method), 96
get_parameters() (stable_baselines3.ppo.PPO method), 113
get_parameters() (stable_baselines3.sac.SAC method), 123
get_parameters() (stable_baselines3.td3.TD3 method), 132
get_schedule_fn() (in module stable_baselines3.common.distributions.StateDependentNoiseDistribution method), 148
get_std() (stable_baselines3.common.distributions.StateDependentNoiseDistribution method), 148
get_vec_normalize_env() (stable_baselines3.a2c.A2C method), 79
get_vec_normalize_env() (stable_baselines3.common.base_class.BaseAlgorithm method), 67
get_vec_normalize_env() (stable_baselines3.ddpg.DDPG method), 89
get_vec_normalize_env() (stable_baselines3.dqn.DQN method), 97
get_vec_normalize_env() (stable_baselines3.ppo.PPO method), 113
get_vec_normalize_env() (stable_baselines3.sac.SAC method), 123
get_vec_normalize_env() (stable_baselines3.td3.TD3 method), 132
getattr_depth_check() (stable_baselines3.common.vec_env.VecEnv method), 25
getattr_depth_check() (stable_baselines3.her.ObsDictWrapper method), 106
Stable Baselines3 Documentation, Release 0.11.0a5

load() (stable_baselines3.a2c.A2C method), 80
load() (stable_baselines3.common.base_class.BaseAlgorithm class method), 68
load() (stable_baselines3.common.vec_env.VecNormalize static method), 30
load() (stable_baselines3.ddpg.DDPG class method), 89
load() (stable_baselines3.dqn.DQN class method), 97
load() (stable_baselines3.her.HER class method), 104
load() (stable_baselines3.ppo.PPO class method), 114
load() (stable_baselines3.sac.SAC class method), 124
load() (stable_baselines3.td3.TD3 class method), 133
load_replay_buffer() (stable_baselines3.common.off_policy_algorithm.OffPolicyAlgorithm method), 73
load_replay_buffer() (stable_baselines3.ddpg.DDPG method), 90
load_replay_buffer() (stable_baselines3.dqn.DQN method), 97
load_replay_buffer() (stable_baselines3.her.HER method), 104
load_replay_buffer() (stable_baselines3.sac.SAC method), 124
load_replay_buffer() (stable_baselines3.td3.TD3 method), 133
load_results() (in module stable_baselines3.common.monitor), 153
log() (in module stable_baselines3.common.logger), 156
log_prob() (stable_baselines3.common.distributions.BernoulliDistribution method), 141
log_prob() (stable_baselines3.common.distributions.CategoricalDistribution method), 142
log_prob() (stable_baselines3.common.distributions.DiagGaussianDistribution method), 143
log_prob() (stable_baselines3.common.distributions.Distribution method), 145
log_prob() (stable_baselines3.common.distributions.MultiCategoricalDistribution method), 146
log_prob() (stable_baselines3.common.distributions.SquashedDiagGaussianDistribution method), 147
log_prob() (stable_baselines3.common.distributions.StateDependentNoiseDistribution method), 148
log_prob_from_params() (stable_baselines3.common.distributions.BernoulliDistribution method), 142
log_prob_from_params() (stable_baselines3.common.distributions.CategoricalDistribution method), 143
log_prob_from_params() (stable_baselines3.common.distributions.DiagGaussianDistribution method), 144
log_prob_from_params() (stable_baselines3.common.distributions.Distribution method), 145

Index
OrnsteinUhlenbeckActionNoise (class in stable_baselines3.common.noise), 158

polyak_update() (in module stable_baselines3.common.noise), 158

PPO (class in stable_baselines3.ppo), 112

PPO (class in stable_baselines3.common.noise), 112

predict() (stable_baselines3.ppo.PPO method), 112

predict() (stable_baselines3.common.base_class.BaseAlgorithm method), 68

predict() (stable_baselines3.ppo.PPO method), 114

proba_distribution() (stable_baselines3.common.distributions.Distribution method), 145

proba_distribution() (stable_baselines3.common.distributions.CategoricalDistribution method), 143

proba_distribution() (stable_baselines3.common.distributions.BernoulliDistribution method), 142

proba_distribution() (stable_baselines3.common.distributions.MultiCategoricalDistribution method), 146

proba_distribution() (stable_baselines3.common.distributions.Distribution method), 145

proba_distribution() (stable_baselines3.common.distributions.CategoricalDistribution method), 143

proba_distribution() (stable_baselines3.common.distributions.BernoulliDistribution method), 142

proba_distribution() (stable_baselines3.common.distributions.BernoulliDistribution method), 142
SAC (class in stable_baselines3.sac), 121
safe_mean() (in module stable_baselines3.common.utils), 161
sample() (stable_baselines3.common.distributions.BernoulliDistribution method), 142
sample() (stable_baselines3.common.distributions.CategoricalDistribution method), 143
sample() (stable_baselines3.common.distributions.DiagGaussianDistribution method), 144
sample() (stable_baselines3.common.distributions.Distribution method), 145
sample() (stable_baselines3.common.distributions.MultiCategoricalDistribution method), 146
sample() (stable_baselines3.common.distributions.SquashedDiagGaussianDistribution method), 147
sample() (stable_baselines3.common.distributions.StateDependentNoiseDistribution method), 149
sample() (stable_baselines3.her.HerReplayBuffer method), 108
sample_goals() (stable_baselines3.her.HerReplayBuffer method), 109
sample_offline() (stable_baselines3.her.HerReplayBuffer method), 109
sample_weights() (stable_baselines3.common.distributions.StateDependentNoiseObservation method), 149
save() (stable_baselines3.a2c.A2C method), 81
save() (stable_baselines3.common.base_class.BaseAlgorithm method), 68
save() (stable_baselines3.common.vec_env.BaseAlgorithm method), 31
save() (stable_baselines3.common.vec_env.TransposeImage method), 33
save() (stable_baselines3.common.vec_env.VideoRecorder method), 31
save() (stable_baselines3.her.HerReplayBuffer method), 108
save() (stable_baselines3.her.ObsDictWrapper method), 107
save_replay_buffer() (stable_baselines3.ddpg.DDPG method), 90
save_replay_buffer() (stable_baselines3.dqn.DQN method), 98
save_replay_buffer() (stable_baselines3.sac.SAC method), 125
save_replay_buffer() (stable_baselines3.td3.TD3 method), 134
seed() (stable_baselines3.common.env.SubprocVecEnv method), 28
seed() (stable_baselines3.common.env.VecEnv method), 25
seed() (stable_baselines3.her.ObsDictWrapper method), 107
SeqWriter (class in stable_baselines3.common.logger), 154
set_attr() (stable_baselines3.common.vec_env.DummyVecEnv method), 27
set_env() (stable_baselines3.dqn.DQN method), 98
set_env() (stable_baselines3.her.HerReplayBuffer method), 109
set_env() (stable_baselines3.her.ObsDictWrapper method), 107
set_env() (stable_baselines3.sac.SAC method), 125
set_env() (stable_baselines3.td3.TD3 method), 134
set_parameters() (stable_baselines3.a2c.A2C method), 81
set_parameters() (stable_baselines3.common.base_class.BaseAlgorithm method), 69
set_parameters() (stable_baselines3.ddpg.DDPG method), 90
set_parameters() (stable_baselines3.dqn.DQN method), 98
set_parameters() (stable_baselines3.her.HerReplayBuffer method), 109
set_parameters() (stable_baselines3.ppo.PPO method), 115
set_parameters() (stable_baselines3.sac.SAC method), 125
set_parameters() (stable_baselines3.td3.TD3 method), 134
set_level() (in module stable_baselines3.common.logger), 158
set_parameters() (stable_baselines3.a2c.A2C method), 81
set_parameters() (stable_baselines3.common.base_class.BaseAlgorithm method), 69
set_parameters() (stable_baselines3.ddpg.DDPG method), 91
set_parameters() (stable_baselines3.dqn.DQN method), 98
set_parameters() (stable_baselines3.her.HerReplayBuffer method), 105
set_parameters() (stable_baselines3.ppo.PPO method), 115
set_parameters() (stable_baselines3.sac.SAC method), 125
set_parameters() (stable_baselines3.td3.TD3 method), 134