Stable Baselines Documentation

Release 2.7.0

Stable Baselines Contributors

Nov 23, 2019

User Guide

Stable Baselines is a set of improved implementations of Reinforcement Learning (RL) algorithms based on OpenAI Baselines.

Github repository: https://github.com/hill-a/stable-baselines

RL Baselines Zoo (collection of pre-trained agents): https://github.com/araffin/rl-baselines-zoo

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

You can read a detailed presentation of Stable Baselines in the Medium article: link

CHAPTER 1

Main differences with OpenAI Baselines

This toolset is a fork of OpenAI Baselines, with a major structural refactoring, and code cleanups:

- Unified structure for all algorithms
- PEP8 compliant (unified code style)
- Documented functions and classes
- More tests & more code coverage
- Additional algorithms: SAC and TD3 (+ HER support for DQN, DDPG, SAC and TD3)

1.1 Installation

1.1.1 Prerequisites

Baselines requires python3 (>=3.5) with the development headers. You'll also need system packages CMake, OpenMPI and zlib. Those can be installed as follows

Ubuntu

```
sudo apt-get update && sudo apt-get install cmake libopenmpi-dev python3-dev zlib1g-
↪dev
```

Mac OS X

Installation of system packages on Mac requires Homebrew. With Homebrew installed, run the following:

```
brew install cmake openmpi
```

Windows 10

We recommend using Anaconda for windows users.

0. Create a new environment in the Anaconda Navigator (at least python 3.5) and install `zlib` in this environment.
1. Install MPI for Windows (you need to download and install `msmpisetup.exe`)
2. Clone Stable-Baselines Github repo and replace the line `gym[atari,classic_control]>=0.10.9` in `setup.py` by this one: `gym[classic_control]>=0.10.9`
3. Install Stable-Baselines from source, inside the folder, run `pip install -e .`

4. [Optional] If you want to use atari environments, you need to install this package: https://github.com/j8lp/atari-py (using again `pip install -e .`)

1.1.2 Stable Release

```
pip install stable-baselines
```

1.1.3 Bleeding-edge version

With support for running tests and building the documentation.

```
git clone https://github.com/hill-a/stable-baselines && cd stable-baselines
pip install -e .[docs,tests]
```

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 Baselines Zoo.

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

Use Built Images

GPU image (requires nvidia-docker):

```
docker pull araffin/stable-baselines
```

CPU only:

```
docker pull araffin/stable-baselines-cpu
```

Build the Docker Images

Build GPU image (with nvidia-docker):

```
docker build . -f docker/Dockerfile.gpu -t stable-baselines
```

Build CPU image:

```
docker build . -f docker/Dockerfile.cpu -t stable-baselines-cpu
```

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

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

Run the images (CPU/GPU)

Run the nvidia-docker GPU image

```
docker run -it --runtime=nvidia --rm --network host --ipc=host --name test --mount␣
↪src="$(pwd)",target=/root/code/stable-baselines,type=bind araffin/stable-baselines␣
↪bash -c 'cd /root/code/stable-baselines/ && pytest tests/'
```

Or, with the shell file:

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

Run the docker CPU image

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

Or, with the shell file:

```
./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 explicitely the name `test` to the container, otherwise it will be assigned a random name
- `--mount src=...` give access of the local directory (`pwd` command) to the container (it will be map to `/root/code/stable-baselines`), so all the logs created in the container in this folder will be kept
- `bash -c '...'` Run command inside the docker image, here run the tests (`pytest tests/`)

1.2 Getting Started

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

Here is a quick example of how to train and run PPO2 on a cartpole environment:

```
import gym

from stable_baselines.common.policies import MlpPolicy
from stable_baselines.common.vec_env import DummyVecEnv
from stable_baselines import PPO2

env = gym.make('CartPole-v1')
env = DummyVecEnv([lambda: env])  # The algorithms require a vectorized environment
↪to run

model = PPO2(MlpPolicy, env, verbose=1)
model.learn(total_timesteps=10000)

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

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

```
from stable_baselines import PPO2

model = PPO2('MlpPolicy', 'CartPole-v1').learn(10000)
```

Fig. 1: Define and train a RL agent in one line of code!

1.3 Reinforcement Learning Resources

Stable-Baselines 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
- More resources

1.4 RL Algorithms

This table displays the rl algorithms that are implemented in the stable baselines project, along with some useful characteristics: support for recurrent policies, discrete/continuous actions, multiprocessing.

Name	Refactored[1]	Recurrent	`Box`	`Discrete`	Multi Processing
A2C	✓	✓	✓	✓	✓
ACER	✓	✓	[4]	✓	✓
ACKTR	✓	✓	[4]	✓	✓
DDPG	✓		✓		✓[3]
DQN	✓			✓	
HER	✓		✓	✓	
GAIL[2]	✓	✓	✓	✓	✓[3]
PPO1	✓		✓	✓	✓[3]
PPO2	✓	✓	✓	✓	✓
SAC	✓		✓		
TD3	✓		✓		
TRPO	✓		✓	✓	✓[3]

Note: Non-array spaces such as `Dict` or `Tuple` are not currently supported by any algorithm, except HER for dict when working with `gym.GoalEnv`

Actions `gym.spaces`:

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

Note: Some logging values (like *ep_rewmean*, *eplenmean*) are only available when using a Monitor wrapper See Issue #339 for more info.

1.5 Examples

1.5.1 Try it online with Colab Notebooks!

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

- Getting Started
- Training, Saving, Loading
- Multiprocessing
- Monitor Training and Plotting
- Atari Games
- Breakout (trained agent included)

[1] Whether or not the algorithm has be refactored to fit the `BaseRLModel` class.
[4] TODO, in project scope.
[3] Multi Processing with MPI.
[2] Only implemented for TRPO.

- Hindsight Experience Replay
- RL Baselines zoo

1.5.2 Basic Usage: Training, Saving, Loading

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

Fig. 2: 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`

Note: `load` function re-creates model from scratch on each call, which can be slow. If you need to e.g. evaluate same model with multiple different sets of parameters, consider using `load_parameters` instead.

```
import gym

from stable_baselines import DQN

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

# Instantiate the agent
model = DQN('MlpPolicy', env, learning_rate=1e-3, prioritized_replay=True, 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")

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

1.5.3 Multiprocessing: Unleashing the Power of Vectorized Environments

Try it in a CO notebook

Fig. 3: CartPole Environment

```
import gym
import numpy as np

from stable_baselines.common.policies import MlpPolicy
from stable_baselines.common.vec_env import SubprocVecEnv
from stable_baselines.common import set_global_seeds
from stable_baselines import ACKTR

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

    :param env_id: (str) the environment ID
    :param num_env: (int) the number of environments you wish to have in subprocesses
    :param seed: (int) the inital seed for RNG
    :param rank: (int) index of the subprocess
    """
    def _init():
        env = gym.make(env_id)
        env.seed(seed + rank)
        return env
    set_global_seeds(seed)
    return _init

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)])

model = ACKTR(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()
```

1.5.4 Using Callback: Monitoring Training

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.

Try it in a notebook

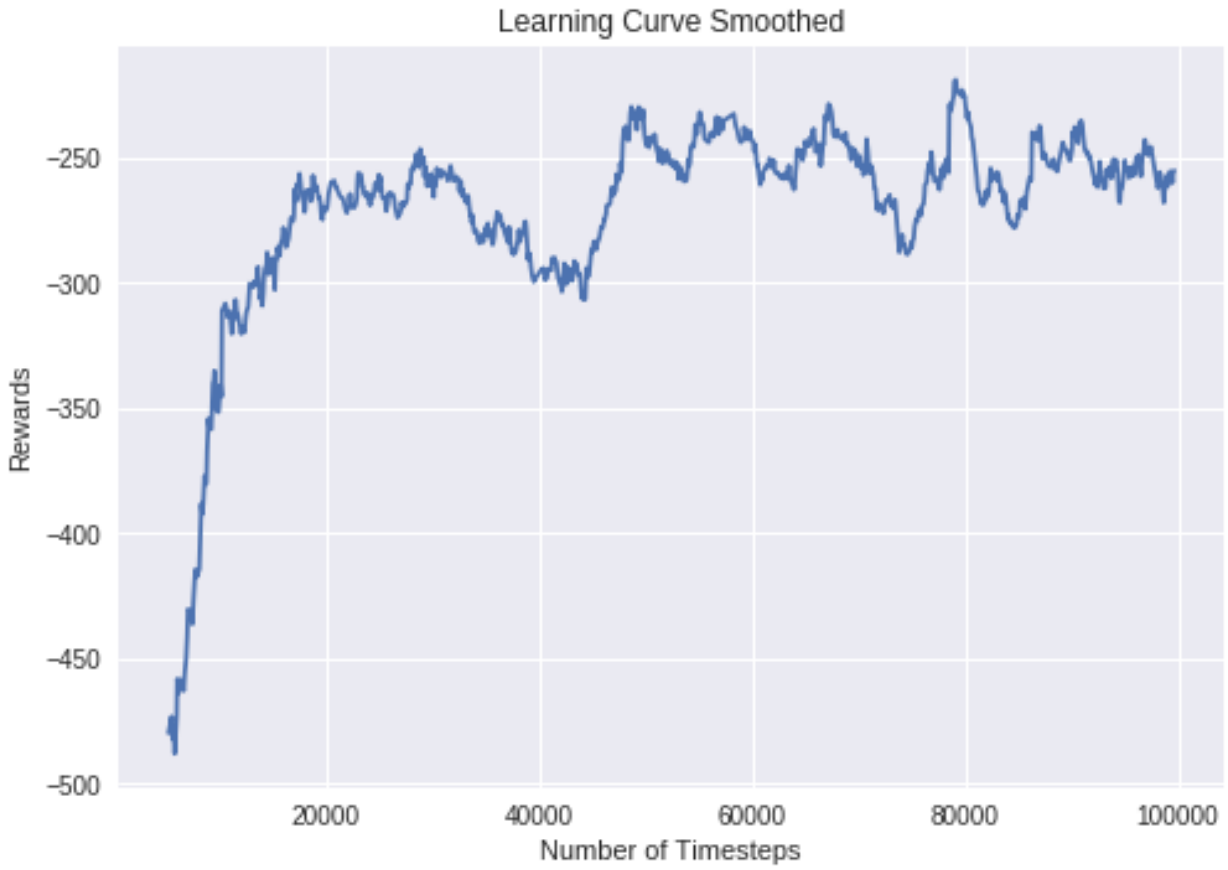


Fig. 4: Learning curve of DDPG on LunarLanderContinuous environment

```
import os

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

from stable_baselines.ddpg.policies import LnMlpPolicy
from stable_baselines.bench import Monitor
from stable_baselines.results_plotter import load_results, ts2xy
from stable_baselines import DDPG
from stable_baselines.ddpg import AdaptiveParamNoiseSpec


best_mean_reward, n_steps = -np.inf, 0

def callback(_locals, _globals):
  """
  Callback called at each step (for DQN an others) or after n steps (see ACER or PPO2)
  :param _locals: (dict)
```

(continues on next page)

(continued from previous page)

```
  :param _globals: (dict)
  """
  global n_steps, best_mean_reward
  # Print stats every 1000 calls
  if (n_steps + 1) % 1000 == 0:
      # Evaluate policy training performance
      x, y = ts2xy(load_results(log_dir), 'timesteps')
      if len(x) > 0:
          mean_reward = np.mean(y[-100:])
          print(x[-1], 'timesteps')
          print("Best mean reward: {:.2f} - Last mean reward per episode: {:.2f}".
↪format(best_mean_reward, mean_reward))

          # New best model, you could save the agent here
          if mean_reward > best_mean_reward:
              best_mean_reward = mean_reward
              # Example for saving best model
              print("Saving new best model")
              _locals['self'].save(log_dir + 'best_model.pkl')
  n_steps += 1
  return True


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

# Create and wrap the environment
env = gym.make('LunarLanderContinuous-v2')
env = Monitor(env, log_dir, allow_early_resets=True)

# Add some param noise for exploration
param_noise = AdaptiveParamNoiseSpec(initial_stddev=0.1, desired_action_stddev=0.1)
# Because we use parameter noise, we should use a MlpPolicy with layer normalization
model = DDPG(LnMlpPolicy, env, param_noise=param_noise, verbose=0)
# Train the agent
model.learn(total_timesteps=int(1e5), callback=callback)
```

1.5.5 Atari Games

Fig. 5: Trained A2C agent on Breakout

Fig. 6: Pong Environment

Training a RL agent on Atari games is straightforward thanks to `make_atari_env` helper function. It will do all the preprocessing and multiprocessing for you.

Try it in a CO notebook

```
from stable_baselines.common.cmd_util import make_atari_env
from stable_baselines.common.vec_env import VecFrameStack
from stable_baselines import ACER

# There already exists an environment generator
# that will make and wrap atari environments correctly.
# Here we are also multiprocessing training (num_env=4 => 4 processes)
env = make_atari_env('PongNoFrameskip-v4', num_env=4, seed=0)
# Frame-stacking with 4 frames
env = VecFrameStack(env, n_stack=4)

model = ACER('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()
```

1.5.6 Mujoco: 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 Mujoco. 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: We cannot provide a notebook for this example because Mujoco is a proprietary engine and requires a license.

```
import gym

from stable_baselines.common.policies import MlpPolicy
from stable_baselines.common.vec_env import DummyVecEnv, VecNormalize
from stable_baselines import PPO2

env = DummyVecEnv([lambda: gym.make("Reacher-v2")])
# Automatically normalize the input features
env = VecNormalize(env, norm_obs=True, norm_reward=False,
                   clip_obs=10.)

model = PPO2(MlpPolicy, env)
model.learn(total_timesteps=2000)

# Don't forget to save the running average when saving the agent
log_dir = "/tmp/"
model.save(log_dir + "ppo_reacher")
env.save_running_average(log_dir)
```

1.5.7 Custom Policy Network

Stable baselines provides default policy networks for images (CNNPolicies) and other type of inputs (MlpPolicies). However, you can also easily define a custom architecture for the policy network (see custom policy section):

```
import gym

from stable_baselines.common.policies import FeedForwardPolicy
from stable_baselines.common.vec_env import DummyVecEnv
from stable_baselines import A2C

# Custom MLP policy of three layers of size 128 each
class CustomPolicy(FeedForwardPolicy):
    def __init__(self, *args, **kwargs):
        super(CustomPolicy, self).__init__(*args, **kwargs,
                                           net_arch=[dict(pi=[128, 128, 128], vf=[128,
↪ 128, 128])],
                                           feature_extraction="mlp")

model = A2C(CustomPolicy, 'LunarLander-v2', verbose=1)
# Train the agent
model.learn(total_timesteps=100000)
```

1.5.8 Accessing and modifying model parameters

You can access model's parameters via `load_parameters` and `get_parameters` functions, which use dictionaries that map variable names to NumPy arrays.

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.

You can access original Tensorflow Variables with function `get_parameter_list`.

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

```
import gym
import numpy as np

from stable_baselines.common.policies import MlpPolicy
from stable_baselines.common.vec_env import DummyVecEnv
from stable_baselines import A2C

def mutate(params):
    """Mutate parameters by adding normal noise to them"""
    return dict((name, param + np.random.normal(size=param.shape))
                for name, param in params.items())

def evaluate(env, model):
    """Return mean fitness (sum of episodic rewards) for given model"""
    episode_rewards = []
    for _ in range(10):
        reward_sum = 0
        done = False
        obs = env.reset()
        while not done:
            action, _states = model.predict(obs)
            obs, reward, done, info = env.step(action)
            reward_sum += reward
        episode_rewards.append(reward_sum)
```

(continues on next page)

(continued from previous page)

```
    return np.mean(episode_rewards)

# Create env
env = gym.make('CartPole-v1')
env = DummyVecEnv([lambda: env])
# Create policy with a small network
model = A2C(MlpPolicy, env, ent_coef=0.0, learning_rate=0.1,
            policy_kwargs={'net_arch': [8, ]})

# Use traditional actor-critic policy gradient updates to
# find good initial parameters
model.learn(total_timesteps=5000)

# Get the parameters as the starting point for ES
mean_params = model.get_parameters()

# Include only variables with "/pi/" (policy) or "/shared" (shared layers)
# in their name: Only these ones affect the action.
mean_params = dict((key, value) for key, value in mean_params.items()
                   if ("/pi/" in key or "/shared" in key))

for iteration in range(10):
    # Create population of candidates and evaluate them
    population = []
    for population_i in range(100):
        candidate = mutate(mean_params)
        # Load new policy parameters to agent.
        # Tell function that it should only update parameters
        # we give it (policy parameters)
        model.load_parameters(candidate, exact_match=False)
        fitness = evaluate(env, model)
        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)[:10]
    mean_params = dict(
        (name, np.stack([top_candidate[0][name] for top_candidate in top_candidates]).
↪mean(0))
        for name in mean_params.keys()
    )
    mean_fitness = sum(top_candidate[1] for top_candidate in top_candidates) / 10.0
    print("Iteration {:<3} Mean top fitness: {:.2f}".format(iteration, mean_fitness))
```

1.5.9 Recurrent Policies

This example demonstrate how to train a recurrent policy and how to test it properly.

Warning: One current limitation of recurrent policies is that you must test them with the same number of environments they have been trained on.

```
from stable_baselines import PPO2

# For recurrent policies, with PPO2, the number of environments run in parallel
# should be a multiple of nminibatches.
```

(continues on next page)

(continued from previous page)

```
model = PPO2('MlpLstmPolicy', 'CartPole-v1', nminibatches=1, verbose=1)
model.learn(50000)

# Retrieve the env
env = model.get_env()

obs = env.reset()
# Passing state=None to the predict function means
# it is the initial state
state = None
# When using VecEnv, done is a vector
done = [False for _ in range(env.num_envs)]
for _ in range(1000):
    # We need to pass the previous state and a mask for recurrent policies
    # to reset lstm state when a new episode begin
    action, state = model.predict(obs, state=state, mask=done)
    obs, reward , done, _ = env.step(action)
    # Note: with VecEnv, env.reset() is automatically called

    # Show the env
    env.render()
```

1.5.10 Hindsight Experience Replay (HER)

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

Fig. 7: 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_baselines import HER, SAC, DDPG, TD3
from stable_baselines.ddpg import NormalActionNoise

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,
```

(continues on next page)

(continued from previous page)

```
            goal_selection_strategy='future',
            verbose=1, buffer_size=int(1e6),
            learning_rate=1e-3,
            gamma=0.95, batch_size=256,
            policy_kwargs=dict(layers=[256, 256, 256]))

# DDPG Hyperparams:
# NOTE: it works even without action noise
# n_actions = env.action_space.shape[0]
# noise_std = 0.2
# action_noise = NormalActionNoise(mean=np.zeros(n_actions), sigma=noise_std * np.
↪ones(n_actions))
# model = HER('MlpPolicy', env, DDPG, n_sampled_goal=n_sampled_goal,
#             goal_selection_strategy='future',
#             verbose=1, buffer_size=int(1e6),
#             actor_lr=1e-3, critic_lr=1e-3, action_noise=action_noise,
#             gamma=0.95, batch_size=256,
#             policy_kwargs=dict(layers=[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)
      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()
```

1.5.11 Continual Learning

You can also move from learning on one environment to another for continual learning (PPO2 on `DemonAttack-v0`, then transferred on `SpaceInvaders-v0`):

```
from stable_baselines.common.cmd_util import make_atari_env
from stable_baselines import PPO2

# There already exists an environment generator
# that will make and wrap atari environments correctly
env = make_atari_env('DemonAttackNoFrameskip-v4', num_env=8, seed=0)

model = PPO2('CnnPolicy', env, verbose=1)
model.learn(total_timesteps=10000)
```

(continues on next page)

(continued from previous page)

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

# The number of environments must be identical when changing environments
env = make_atari_env('SpaceInvadersNoFrameskip-v4', num_env=8, seed=0)

# change env
model.set_env(env)
model.learn(total_timesteps=10000)

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

1.5.12 Record a Video

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

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

```
import gym
from stable_baselines.common.vec_env import VecVideoRecorder, DummyVecEnv

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

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

obs = env.reset()

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

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

1.5.13 Bonus: Make a GIF of a Trained Agent

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

```
import imageio
import numpy as np

from stable_baselines.common.policies import MlpPolicy
from stable_baselines 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[0]) for i, img in enumerate(images)␣
↪if i%2 == 0], fps=29)
```

1.6 Vectorized Environments

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

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

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

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

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

For more information, see Python's multiprocessing guidelines.

1.6.1 VecEnv

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

An abstract asynchronous, vectorized environment.

Parameters

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

close()

Clean up the environment's resources.

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

Call instance methods of vectorized environments.

Parameters

- **method_name** – (str) The name of the environment method to invoke.
- **indices** – (list,int) Indices of envs whose method to call
- **method_args** – (tuple) Any positional arguments to provide in the call
- **method_kwargs** – (dict) Any keyword arguments to provide in the call

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

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

Return attribute from vectorized environment.

Parameters

- **attr_name** – (str) The name of the attribute whose value to return
- **indices** – (list,int) Indices of envs to get attribute from

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

get_images()

Return RGB images from each environment

getattr_depth_check(*name*, *already_found*)

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

Parameters

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

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

render(**args*, ***kwargs*)

Gym environment rendering

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

reset()

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

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

Returns ([int] or [float]) observation

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

Parameters

- **attr_name** – (str) The name of attribute to assign new value
- **value** – (obj) Value to assign to *attr_name*
- **indices** – (list,int) Indices of envs to assign value

Returns (NoneType)

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

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

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

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

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

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

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

1.6.2 DummyVecEnv

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

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

close()
Clean up the environment's resources.

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

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

get_images()
Return RGB images from each environment

render(**args*, ***kwargs*)
Gym environment rendering

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

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

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

Returns ([int] or [float]) observation

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

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

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

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

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

1.6.3 SubprocVecEnv

class stable_baselines.common.vec_env.**SubprocVecEnv**(*env_fns*, *start_method=None*)
Creates a multiprocess vectorized wrapper for multiple environments, distributing each environment to its own process, allowing significant speed up when the environment is computationally complex.

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

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

Parameters

- **env_fns** – ([Gym Environment]) Environments to run in subprocesses
- **start_method** – (str) method used to start the subprocesses. Must be one of the methods returned by multiprocessing.get_all_start_methods(). Defaults to 'fork' on available platforms, and 'spawn' otherwise.

close()
Clean up the environment's resources.

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

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

get_images()
Return RGB images from each environment

render(*mode='human'*, **args*, ***kwargs*)
Gym environment rendering

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

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

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

Returns ([int] or [float]) observation

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

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

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

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

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

1.6.4 Wrappers

VecFrameStack

class stable_baselines.common.vec_env.**VecFrameStack**(*venv*, *n_stack*)
Frame stacking wrapper for vectorized environment

Parameters

- **venv** – (VecEnv) the vectorized environment to wrap
- **n_stack** – (int) Number of frames to stack

close()
Clean up the environment's resources.

reset()
Reset all environments

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

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

VecNormalize

class stable_baselines.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 the unnormalized observation

Returns (numpy float)

load_running_average(*path*)

Parameters **path** – (str) path to log dir

reset()
Reset all environments

save_running_average(*path*)

Parameters **path** – (str) path to log dir

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

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

VecVideoRecorder

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

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

Parameters

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

close()
Clean up the environment's resources.

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

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

Returns ([int] or [float]) observation

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

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

VecCheckNan

class stable_baselines.common.vec_env.**VecCheckNan**(*venv*, *raise_exception=False*, *warn_once=True*, *check_inf=True*)
NaN and inf checking wrapper for vectorized environment, will raise a warning by default, allowing you to know from what the NaN of inf originated from.

Parameters

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

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

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

Returns ([int] or [float]) observation

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

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

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

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

1.7 Using Custom Environments

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

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

```
import gym
from gym import spaces

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

(continues on next page)

(continued from previous page)

```
  metadata = {'render.modes': ['human']}

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

  def step(self, action):
    ...
  def reset(self):
    ...
  def render(self, mode='human', close=False):
    ...
```

Then you can define and train a RL agent with:

```
# Instantiate and wrap the env
env = DummyVecEnv([lambda: CustomEnv(arg1, ...)])
# Define and Train the agent
model = A2C(CnnPolicy, env).learn(total_timesteps=1000)
```

You can 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.

1.8 Custom Policy Network

Stable baselines provides default policy networks (see *Policies*) for images (CNNPolicies) and other type of input features (MlpPolicies).

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

```
import gym
import tensorflow as tf

from stable_baselines import PPO2

# Custom MLP policy of two layers of size 32 each with tanh activation function
policy_kwargs = dict(act_fun=tf.nn.tanh, net_arch=[32, 32])
# Create the agent
model = PPO2("MlpPolicy", "CartPole-v1", policy_kwargs=policy_kwargs, verbose=1)
# Retrieve the environment
env = model.get_env()
# Train the agent
```

(continues on next page)

(continued from previous page)

```
model.learn(total_timesteps=100000)
# Save the agent
model.save("ppo2-cartpole")

del model
# the policy_kwargs are automatically loaded
model = PPO2.load("ppo2-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 makes usually the code clearer. `policy_kwargs` should be rather used when doing hyperparameter search.

```
import gym

from stable_baselines.common.policies import FeedForwardPolicy, register_policy
from stable_baselines.common.vec_env import DummyVecEnv
from stable_baselines import A2C

# Custom MLP policy of three layers of size 128 each
class CustomPolicy(FeedForwardPolicy):
    def __init__(self, *args, **kwargs):
        super(CustomPolicy, self).__init__(*args, **kwargs,
                                           net_arch=[dict(pi=[128, 128, 128],
                                                          vf=[128, 128, 128])],
                                           feature_extraction="mlp")

# Create and wrap the environment
env = gym.make('LunarLander-v2')
env = DummyVecEnv([lambda: env])

model = A2C(CustomPolicy, env, verbose=1)
# Train the agent
model.learn(total_timesteps=100000)
# Save the agent
model.save("a2c-lunar")

del model
# When loading a model with a custom policy
# you MUST pass explicitly the policy when loading the saved model
model = A2C.load("a2c-lunar", policy=CustomPolicy)
```

Warning: When loading a model with a custom policy, you must pass the custom policy explicitly when loading the model. (cf previous example)

You can also register your policy, to help with code simplicity: you can refer to your custom policy using a string.

```
import gym

from stable_baselines.common.policies import FeedForwardPolicy, register_policy
from stable_baselines.common.vec_env import DummyVecEnv
```

(continues on next page)

(continued from previous page)

```
from stable_baselines import A2C

# Custom MLP policy of three layers of size 128 each
class CustomPolicy(FeedForwardPolicy):
    def __init__(self, *args, **kwargs):
        super(CustomPolicy, self).__init__(*args, **kwargs,
                                           net_arch=[dict(pi=[128, 128, 128],
                                                          vf=[128, 128, 128])],
                                           feature_extraction="mlp")

# Register the policy, it will check that the name is not already taken
register_policy('CustomPolicy', CustomPolicy)

# Because the policy is now registered, you can pass
# a string to the agent constructor instead of passing a class
model = A2C(policy='CustomPolicy', env='LunarLander-v2', verbose=1).learn(total_
↪timesteps=100000)
```

Deprecated since version 2.3.0: Use `net_arch` instead of `layers` parameter to define the network architecture. It allows to have a greater control.

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

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

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

1.8.1 Examples

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

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

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

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

(continues on next page)

(continued from previous page)

```
                    |
                  value
```

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

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

The `LstmPolicy` can be used to construct recurrent policies in a similar way:

```
class CustomLSTMPolicy(LstmPolicy):
    def __init__(self, sess, ob_space, ac_space, n_env, n_steps, n_batch, n_lstm=64,␣
↪reuse=False, **_kwargs):
        super().__init__(sess, ob_space, ac_space, n_env, n_steps, n_batch, n_lstm,␣
↪reuse,
                         net_arch=[8, 'lstm', dict(vf=[5, 10], pi=[10])],
                         layer_norm=True, feature_extraction="mlp", **_kwargs)
```

Here the `net_arch` parameter takes an additional (mandatory) 'lstm' entry within the shared network section. The LSTM is shared between value network and policy network.

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

```
import gym
import tensorflow as tf

from stable_baselines.common.policies import ActorCriticPolicy, register_policy,␣
↪nature_cnn
from stable_baselines.common.vec_env import DummyVecEnv
from stable_baselines import A2C

# Custom MLP policy of three layers of size 128 each for the actor and 2 layers of 32␣
↪for the critic,
# with a nature_cnn feature extractor
class CustomPolicy(ActorCriticPolicy):
    def __init__(self, sess, ob_space, ac_space, n_env, n_steps, n_batch, reuse=False,
↪ **kwargs):
        super(CustomPolicy, self).__init__(sess, ob_space, ac_space, n_env, n_steps,␣
↪n_batch, reuse=reuse, scale=True)

        with tf.variable_scope("model", reuse=reuse):
            activ = tf.nn.relu

            extracted_features = nature_cnn(self.processed_obs, **kwargs)
            extracted_features = tf.layers.flatten(extracted_features)

            pi_h = extracted_features
            for i, layer_size in enumerate([128, 128, 128]):
                pi_h = activ(tf.layers.dense(pi_h, layer_size, name='pi_fc' + str(i)))
            pi_latent = pi_h

            vf_h = extracted_features
```

(continues on next page)

(continued from previous page)

```
            for i, layer_size in enumerate([32, 32]):
                vf_h = activ(tf.layers.dense(vf_h, layer_size, name='vf_fc' + str(i)))
            value_fn = tf.layers.dense(vf_h, 1, name='vf')
            vf_latent = vf_h

            self._proba_distribution, self._policy, self.q_value = \
                self.pdtype.proba_distribution_from_latent(pi_latent, vf_latent, init_
↪scale=0.01)

        self._value_fn = value_fn
        self._setup_init()

    def step(self, obs, state=None, mask=None, deterministic=False):
        if deterministic:
            action, value, neglogp = self.sess.run([self.deterministic_action, self.
↪value_flat, self.neglogp],
                                                   {self.obs_ph: obs})
        else:
            action, value, neglogp = self.sess.run([self.action, self.value_flat,␣
↪self.neglogp],
                                                   {self.obs_ph: obs})
        return action, value, self.initial_state, neglogp

    def proba_step(self, obs, state=None, mask=None):
        return self.sess.run(self.policy_proba, {self.obs_ph: obs})

    def value(self, obs, state=None, mask=None):
        return self.sess.run(self.value_flat, {self.obs_ph: obs})


# Create and wrap the environment
env = DummyVecEnv([lambda: gym.make('Breakout-v0')])

model = A2C(CustomPolicy, env, verbose=1)
# Train the agent
model.learn(total_timesteps=100000)
```

1.9 Tensorboard Integration

1.9.1 Basic Usage

To use Tensorboard with the rl baselines, you simply need to define a log location for the RL agent:

```
import gym

from stable_baselines import A2C

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

Or after loading an existing model (by default the log path is not saved):

```
import gym

from stable_baselines.common.vec_env import DummyVecEnv
from stable_baselines import A2C

env = gym.make('CartPole-v1')
env = DummyVecEnv([lambda: env])  # The algorithms require a vectorized environment␣
↪to run

model = A2C.load("./a2c_cartpole.pkl", env=env, 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)

```
import gym

from stable_baselines import A2C

model = A2C('MlpPolicy', 'CartPole-v1', verbose=1, tensorboard_log="./a2c_cartpole_
↪tensorboard/")
model.learn(total_timesteps=10000, tb_log_name="first_run")
# Pass reset_num_timesteps=False to continue the training curve in tensorboard
# By default, it will create a new curve
model.learn(total_timesteps=10000, tb_log_name="second_run", reset_num_
↪timesteps=False)
model.learn(total_timesteps=10000, tb_log_name="thrid_run", reset_num_timesteps=False)
```

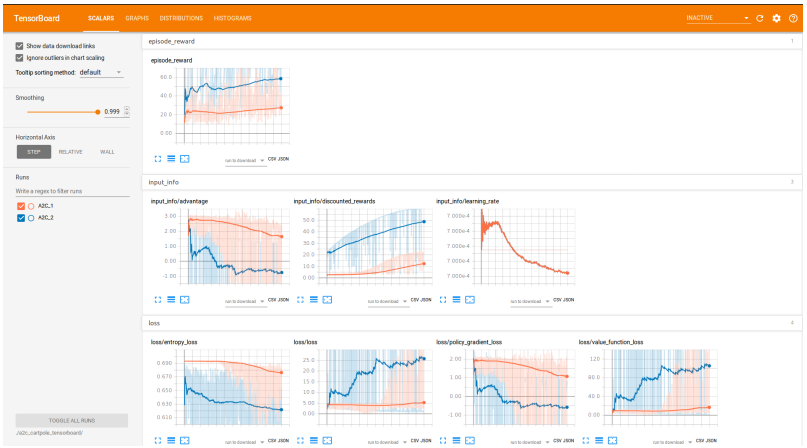
Once the learn function is called, you can monitor the RL agent during or after the training, with the following bash command:

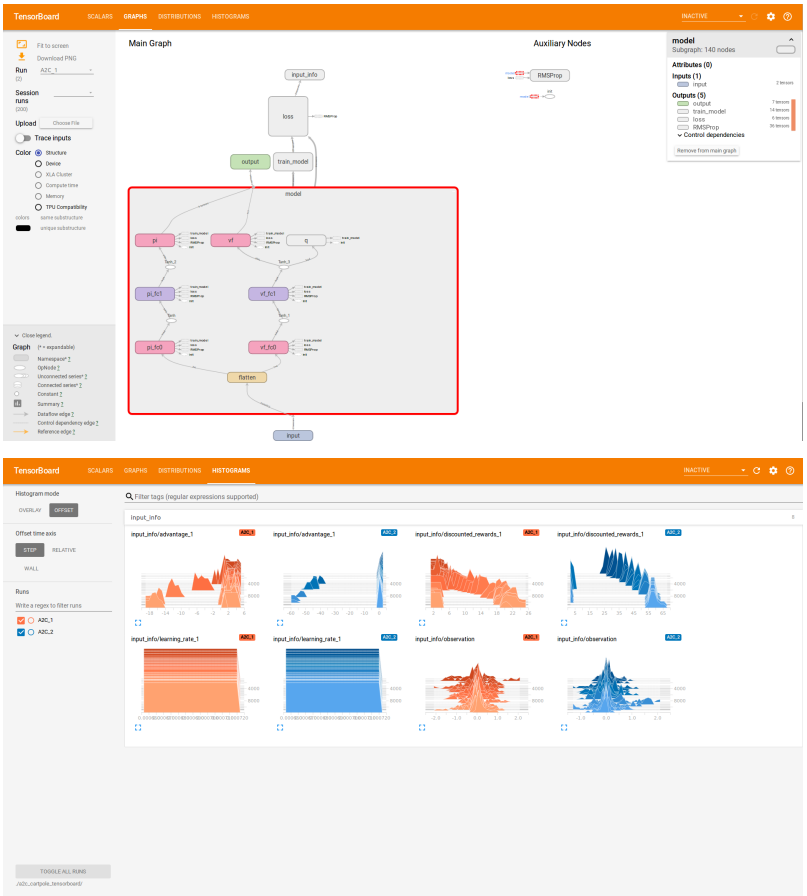
```
tensorboard --logdir ./a2c_cartpole_tensorboard/
```

you can also add past logging folders:

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

It will display information such as the model graph, the episode reward, the model losses, the observation and other parameter unique to some models.





1.9.2 Logging More Values

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

```
import tensorflow as tf
import numpy as np

from stable_baselines import SAC

model = SAC("MlpPolicy", "Pendulum-v0", tensorboard_log="/tmp/sac/", verbose=1)
# Define a new property to avoid global variable
model.is_tb_set = False


def callback(locals_, globals_):
    self_ = locals_['self']
    # Log additional tensor
    if not self_.is_tb_set:
        with self_.graph.as_default():
            tf.summary.scalar('value_target', tf.reduce_mean(self_.value_target))
            self_.summary = tf.summary.merge_all()
        self_.is_tb_set = True
    # Log scalar value (here a random variable)
    value = np.random.random()
```

(continues on next page)

(continued from previous page)

```
    summary = tf.Summary(value=[tf.Summary.Value(tag='random_value', simple_
↪value=value)])
    locals_['writer'].add_summary(summary, self_.num_timesteps)
    return True


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

1.9.3 Legacy Integration

All the information displayed in the terminal (default logging) can be also logged in tensorboard. For that, you need to define several environment variables:

```
# formats are comma-separated, but for tensorboard you only need the last one
# stdout -> terminal
export OPENAI_LOG_FORMAT='stdout,log,csv,tensorboard'
export OPENAI_LOGDIR=path/to/tensorboard/data
```

and to configure the logger using:

```
from stable_baselines.logger import configure

configure()
```

Then start tensorboard with:

```
tensorboard --logdir=$OPENAI_LOGDIR
```

1.10 RL Baselines Zoo

RL Baselines Zoo. is a collection of pre-trained Reinforcement Learning agents using Stable-Baselines. 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!

1.10.1 Installation

1. Install dependencies

```
apt-get install swig cmake libopenmpi-dev zlib1g-dev ffmpeg
pip install stable-baselines box2d box2d-kengz pyyaml pybullet optuna pytablewriter
```

2. Clone the repository:

```
git clone https://github.com/araffin/rl-baselines-zoo
```

1.10.2 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 tensorboard support):

```
python train.py --algo ppo2 --env CartPole-v1 --tensorboard-log /tmp/stable-baselines/
```

Train for multiple environments (with one call) and with tensorboard logging:

```
python train.py --algo a2c --env MountainCar-v0 CartPole-v1 --tensorboard-log /tmp/
↪stable-baselines/
```

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.pkl -n 5000
```

1.10.3 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 trained_agents/ -n␣
↪5000
```

1.10.4 Hyperparameter Optimization

We use Optuna for optimizing the hyperparameters.

Tune the hyperparameters for PPO2, 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 ppo2 --env MountainCar-v0 -n 50000 -optimize --n-trials 1000 --
↪n-jobs 2 \
  --sampler random --pruner median
```

1.10.5 Colab Notebook: Try it Online!

You can train agents online using Google colab notebook.

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

1.11 Pre-Training (Behavior Cloning)

With the `.pretrain()` method, you can pre-train RL policies using trajectories from an expert, and therefore accelerate training.

Behavior Cloning (BC) treats the problem of imitation learning, i.e., using expert demonstrations, as a supervised learning problem. That is to say, given expert trajectories (observations-actions pairs), the policy network is trained to reproduce the expert behavior: for a given observation, the action taken by the policy must be the one taken by the expert.

Expert trajectories can be human demonstrations, trajectories from another controller (e.g. a PID controller) or trajectories from a trained RL agent.

Note: Only `Box` and `Discrete` spaces are supported for now for pre-training a model.

Note: Images datasets are treated a bit differently as other datasets to avoid memory issues. The images from the expert demonstrations must be located in a folder, not in the expert numpy archive.

1.11.1 Generate Expert Trajectories

Here, we are going to train a RL model and then generate expert trajectories using this agent.

Note that in practice, generating expert trajectories usually does not require training an RL agent.

The following example is only meant to demonstrate the `pretrain()` feature.

However, we recommend users to take a look at the code of the `generate_expert_traj()` function (located in `gail/dataset/` folder) to learn about the data structure of the expert dataset (see below for an overview) and how to record trajectories.

```
from stable_baselines import DQN
from stable_baselines.gail import generate_expert_traj

model = DQN('MlpPolicy', 'CartPole-v1', verbose=1)
      # Train a DQN agent for 1e5 timesteps and generate 10 trajectories
      # data will be saved in a numpy archive named `expert_cartpole.npz`
generate_expert_traj(model, 'expert_cartpole', n_timesteps=int(1e5), n_episodes=10)
```

Here is an additional example when the expert controller is a callable, that is passed to the function instead of a RL model. The idea is that this callable can be a PID controller, asking a human player, ...

```
import gym

from stable_baselines.gail import generate_expert_traj

env = gym.make("CartPole-v1")
# Here the expert is a random agent
```

(continues on next page)

(continued from previous page)

```
# but it can be any python function, e.g. a PID controller
def dummy_expert(_obs):
    """
    Random agent. It samples actions randomly
    from the action space of the environment.

    :param _obs: (np.ndarray) Current observation
    :return: (np.ndarray) action taken by the expert
    """
    return env.action_space.sample()
# Data will be saved in a numpy archive named `expert_cartpole.npz`
# when using something different than an RL expert,
# you must pass the environment object explicitely
generate_expert_traj(dummy_expert, 'dummy_expert_cartpole', env, n_episodes=10)
```

1.11.2 Pre-Train a Model using Behavior Cloning

Using the `expert_cartpole.npz` dataset generated with the previous script.

```
from stable_baselines import PPO2
from stable_baselines.gail import ExpertDataset
# Using only one expert trajectory
# you can specify `traj_limitation=-1` for using the whole dataset
dataset = ExpertDataset(expert_path='expert_cartpole.npz',
                        traj_limitation=1, batch_size=128)

model = PPO2('MlpPolicy', 'CartPole-v1', verbose=1)
# Pretrain the PPO2 model
model.pretrain(dataset, n_epochs=1000)

# As an option, you can train the RL agent
# model.learn(int(1e5))

# Test the pre-trained model
env = model.get_env()
obs = env.reset()

reward_sum = 0.0
for _ in range(1000):
        action, _ = model.predict(obs)
        obs, reward, done, _ = env.step(action)
        reward_sum += reward
        env.render()
        if done:
                print(reward_sum)
                reward_sum = 0.0
                obs = env.reset()

env.close()
```

1.11.3 Data Structure of the Expert Dataset

The expert dataset is a `.npz` archive. The data is saved in python dictionary format with keys: `actions`, `episode_returns`, `rewards`, `obs`, `episode_starts`.

In case of images, `obs` contains the relative path to the images.

obs, actions: shape (N * L,) + S

where N = # episodes, L = episode length and S is the environment observation/action space.

S = (1,) for discrete space

class stable_baselines.gail.**ExpertDataset**(*expert_path=None*, *traj_data=None*, *train_fraction=0.7*, *batch_size=64*, *traj_limitation=-1*, *randomize=True*, *verbose=1*, *sequential_preprocessing=False*)

Dataset for using behavior cloning or GAIL.

The structure of the expert dataset is a dict, saved as an ".npz" archive. The dictionary contains the keys 'actions', 'episode_returns', 'rewards', 'obs' and 'episode_starts'. The corresponding values have data concatenated across episode: the first axis is the timestep, the remaining axes index into the data. In case of images, 'obs' contains the relative path to the images, to enable space saving from image compression.

Parameters

- **expert_path** – (str) The path to trajectory data (.npz file). Mutually exclusive with traj_data.
- **traj_data** – (dict) Trajectory data, in format described above. Mutually exclusive with expert_path.
- **train_fraction** – (float) the train validation split (0 to 1) for pre-training using behavior cloning (BC)
- **batch_size** – (int) the minibatch size for behavior cloning
- **traj_limitation** – (int) the number of trajectory to use (if -1, load all)
- **randomize** – (bool) if the dataset should be shuffled
- **verbose** – (int) Verbosity
- **sequential_preprocessing** – (bool) Do not use subprocess to preprocess the data (slower but use less memory for the CI)

get_next_batch(*split=None*)

Get the batch from the dataset.

Parameters **split** – (str) the type of data split (can be None, 'train', 'val')

Returns (np.ndarray, np.ndarray) inputs and labels

init_dataloader(*batch_size*)

Initialize the dataloader used by GAIL.

Parameters **batch_size** – (int)

log_info()

Log the information of the dataset.

plot()

Show histogram plotting of the episode returns

prepare_pickling()

Exit processes in order to pickle the dataset.

class stable_baselines.gail.**DataLoader**(*indices*, *observations*, *actions*, *batch_size*, *n_workers=1*, *infinite_loop=True*, *max_queue_len=1*, *shuffle=False*, *start_process=True*, *backend='threading'*, *sequential=False*, *partial_minibatch=True*)

A custom dataloader to preprocessing observations (including images) and feed them to the network.

Original code for the dataloader from https://github.com/araffin/robotics-rl-srl (MIT licence) Authors: Antonin Raffin, René Traoré, Ashley Hill

Parameters

- **indices** – ([int]) list of observations indices
- **observations** – (np.ndarray) observations or images path
- **actions** – (np.ndarray) actions
- **batch_size** – (int) Number of samples per minibatch
- **n_workers** – (int) number of preprocessing worker (for loading the images)
- **infinite_loop** – (bool) whether to have an iterator that can be resetted
- **max_queue_len** – (int) Max number of minibatches that can be preprocessed at the same time
- **shuffle** – (bool) Shuffle the minibatch after each epoch
- **start_process** – (bool) Start the preprocessing process (default: True)
- **backend** – (str) joblib backend (one of 'multiprocessing', 'sequential', 'threading' or 'loky' in newest versions)
- **sequential** – (bool) Do not use subprocess to preprocess the data (slower but use less memory for the CI)
- **partial_minibatch** – (bool) Allow partial minibatches (minibatches with a number of element lesser than the batch_size)

sequential_next()

Sequential version of the pre-processing.

start_process()

Start preprocessing process

stable_baselines.gail.**generate_expert_traj**(*model*, *save_path=None*, *env=None*, *n_timesteps=0*, *n_episodes=100*, *image_folder='recorded_images'*)

Train expert controller (if needed) and record expert trajectories.

Note: only Box and Discrete spaces are supported for now.

Parameters

- **model** – (RL model or callable) The expert model, if it needs to be trained, then you need to pass `n_timesteps > 0`.
- **save_path** – (str) Path without the extension where the expert dataset will be saved (ex: 'expert_cartpole' -> creates 'expert_cartpole.npz'). If not specified, it will not save, and just return the generated expert trajectories. This parameter must be specified for image-based environments.

- **env** – (gym.Env) The environment, if not defined then it tries to use the model environment.
- **n_timesteps** – (int) Number of training timesteps
- **n_episodes** – (int) Number of trajectories (episodes) to record
- **image_folder** – (str) When using images, folder that will be used to record images.

Returns (dict) the generated expert trajectories.

1.12 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.

1.12.1 How and why?

The issue arises then NaNs or infs do not crash, but simply get propagated through the training, until all the floating point number converge to NaN or inf. This is in line with the IEEE Standard for Floating-Point Arithmetic (IEEE 754) standard, as it says:

Note:

Five possible exceptions can occur:

- Invalid operation ($\sqrt{-1}$, $\inf \times 1$, $\text{NaN} \bmod 1$, ...) return NaN
- **Division by zero:**
  - if the operand is not zero ($1/0$, $-2/0$, ...) returns $\pm\inf$
  - if the operand is zero ($0/0$) returns signaling NaN
- Overflow (exponent too high to represent) returns $\pm\inf$
- Underflow (exponent too low to represent) returns 0
- Inexact (not representable exactly in base 2, eg: $1/5$) returns the rounded value (ex: `assert (1/5) * 3 == 0.6000000000000001`)

And of these, only `Division by zero` will signal an exception, the rest will propagate invalid values quietly.

In python, dividing by zero will indeed raise the exception: `ZeroDivisionError: float division by zero`, but ignores the rest.

The default in numpy, will warn: `RuntimeWarning: invalid value encountered` but will not halt the code.

And the worst of all, Tensorflow will not signal anything

```
import tensorflow as tf
import numpy as np

print("tensorflow test:")

a = tf.constant(1.0)
b = tf.constant(0.0)
c = a / b
```

(continues on next page)

(continued from previous page)

```
sess = tf.Session()
val = sess.run(c)  # this will be quiet
print(val)
sess.close()

print("\r\nnumpy test:")

a = np.float64(1.0)
b = np.float64(0.0)
val = a / b  # this will warn
print(val)

print("\r\npure python test:")

a = 1.0
b = 0.0
val = a / b  # this will raise an exception and halt.
print(val)
```

Unfortunately, most of the floating point operations are handled by Tensorflow and numpy, meaning you might get little to no warning when a invalid value occurs.

1.12.2 Numpy parameters

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

```
import numpy as np

np.seterr(all='raise')  # define before your code.

print("numpy test:")

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

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

```
import numpy as np

np.seterr(all='raise')  # define before your code.

print("numpy overflow test:")

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

but will not avoid the propagation issues:

```
import numpy as np

np.seterr(all='raise')  # define before your code.

print("numpy propagation test:")

a = np.float64('NaN')
b = np.float64(1.0)
val = a + b  # this will neither warn nor raise anything
print(val)
```

1.12.3 Tensorflow parameters

Tensorflow can add checks for detecting and dealing with invalid value: tf.add_check_numerics_ops and tf.check_numerics, however they will add operations to the Tensorflow graph and raise the computation time.

```
import tensorflow as tf

print("tensorflow test:")

a = tf.constant(1.0)
b = tf.constant(0.0)
c = a / b

check_nan = tf.add_check_numerics_ops()  # add after your graph definition.

sess = tf.Session()
val, _ = sess.run([c, check_nan])  # this will now raise an exception
print(val)
sess.close()
```

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

```
import tensorflow as tf

print("tensorflow overflow test:")

check_nan = []  # the list of check_numerics operations

a = tf.constant(10)
b = tf.constant(1000)
c = a ** b

check_nan.append(tf.check_numerics(c, ""))  # check the 'c' operations

sess = tf.Session()
val, _ = sess.run([c] + check_nan)  # this will now raise an exception
print(val)
sess.close()
```

and catch propagation issues:

```
import tensorflow as tf

print("tensorflow propagation test:")
```

(continues on next page)

(continued from previous page)

```
check_nan = []  # the list of check_numerics operations

a = tf.constant('NaN')
b = tf.constant(1.0)
c = a + b

check_nan.append(tf.check_numerics(c, ""))  # check the 'c' operations

sess = tf.Session()
val, _ = sess.run([c] + check_nan)  # this will now raise an exception
print(val)
sess.close()
```

1.12.4 VecCheckNan Wrapper

In order to find when and from where the invalid value originated from, stable-baselines comes with a `VecCheckNan` wrapper.

It will monitor the actions, observations, and rewards, indicating what action or observation caused it and from what.

```
import gym
from gym import spaces
import numpy as np

from stable_baselines import PPO2
from stable_baselines.common.vec_env import DummyVecEnv, VecCheckNan

class NanAndInfEnv(gym.Env):
    """Custom Environment that raised NaNs and Infs"""
    metadata = {'render.modes': ['human']}

    def __init__(self):
        super(NanAndInfEnv, self).__init__()
        self.action_space = spaces.Box(low=-np.inf, high=np.inf, shape=(1,), dtype=np.
↪float64)
        self.observation_space = spaces.Box(low=-np.inf, high=np.inf, shape=(1,),␣
↪dtype=np.float64)

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

    def reset(self):
        return [0.0]

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

(continues on next page)

(continued from previous page)

```
# Create environment
env = DummyVecEnv([lambda: NanAndInfEnv()])
env = VecCheckNan(env, raise_exception=True)

# Instantiate the agent
model = PPO2('MlpPolicy', env)

# Train the agent
model.learn(total_timesteps=int(2e5))  # this will crash explaining that the invalid
↪value originated from the environment.
```

1.12.5 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).

1.12.6 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

1.13 Base RL Class

Common interface for all the RL algorithms

class stable_baselines.common.base_class.**BaseRLModel**(*policy*, *env*, *verbose=0*, ***, *requires_vec_env*, *policy_base*, *policy_kwargs=None*)

The base RL model

Parameters

- **policy** – (BasePolicy) Policy object
- **env** – (Gym environment) The environment to learn from (if registered in Gym, can be str. Can be None for loading trained models)
- **verbose** – (int) the verbosity level: 0 none, 1 training information, 2 tensorflow debug
- **requires_vec_env** – (bool) Does this model require a vectorized environment
- **policy_base** – (BasePolicy) the base policy used by this method

action_probability(*observation*, *state=None*, *mask=None*, *actions=None*, *logp=False*)
If `actions` is `None`, then get the model's action probability distribution from a given observation.

Depending on the action space the output is:

- Discrete: probability for each possible action
- Box: mean and standard deviation of the action output

However if `actions` is not `None`, this function will return the probability that the given actions are taken with the given parameters (observation, state, ...) on this model. For discrete action spaces, it returns the probability mass; for continuous action spaces, the probability density. This is since the probability mass will always be zero in continuous spaces, see http://blog.christianperone.com/2019/01/ for a good explanation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (np.ndarray) The last states (can be None, used in recurrent policies)
- **mask** – (np.ndarray) The last masks (can be None, used in recurrent policies)
- **actions** – (np.ndarray) (OPTIONAL) For calculating the likelihood that the given actions are chosen by the model for each of the given parameters. Must have the same number of actions and observations. (set to None to return the complete action probability distribution)
- **logp** – (bool) (OPTIONAL) When specified with actions, returns probability in log-space. This has no effect if actions is None.

Returns (np.ndarray) the model's (log) action probability

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

Returns (Gym Environment) The current environment

get_parameter_list()
Get tensorflow Variables of model's parameters

This includes all variables necessary for continuing training (saving / loading).

Returns (list) List of tensorflow Variables

get_parameters()
Get current model parameters as dictionary of variable name -> ndarray.

Returns (OrderedDict) Dictionary of variable name -> ndarray of model's parameters.

learn(*total_timesteps*, *callback=None*, *seed=None*, *log_interval=100*, *tb_log_name='run'*, *reset_num_timesteps=True*)
Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples to train on
- **seed** – (int) The initial seed for training, if None: keep current seed
- **callback** – (function (dict, dict)) -> boolean function called at every steps with state of the algorithm. It takes the local and global variables. If it returns False, training is aborted.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for tensorboard log

- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseRLModel) the trained model

classmethod load(*load_path*, *env=None*, ***kwargs*)
Load the model from file

Parameters

- **load_path** – (str or file-like) the saved parameter location
- **env** – (Gym Envrionment) the new environment to run the loaded model on (can be None if you only need prediction from a trained model)
- **kwargs** – extra arguments to change the model when loading

load_parameters(*load_path_or_dict*, *exact_match=True*)
Load model parameters from a file or a dictionary

Dictionary keys should be tensorflow variable names, which can be obtained with `get_parameters` function. If `exact_match` is True, dictionary should contain keys for all model's parameters, otherwise RunTimeError is raised. If False, only variables included in the dictionary will be updated.

This does not load agent's hyper-parameters.

Warning: This function does not update trainer/optimizer variables (e.g. momentum). As such training after using this function may lead to less-than-optimal results.

Parameters

- **load_path_or_dict** – (str or file-like or dict) Save parameter location or dict of parameters as variable.name -> ndarrays to be loaded.
- **exact_match** – (bool) If True, expects load dictionary to contain keys for all variables in the model. If False, loads parameters only for variables mentioned in the dictionary. Defaults to True.

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

Parameters

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

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

pretrain(*dataset*, *n_epochs=10*, *learning_rate=0.0001*, *adam_epsilon=1e-08*, *val_interval=None*)
Pretrain a model using behavior cloning: supervised learning given an expert dataset.

NOTE: only Box and Discrete spaces are supported for now.

Parameters

- **dataset** – (ExpertDataset) Dataset manager

- **n_epochs** – (int) Number of iterations on the training set
- **learning_rate** – (float) Learning rate
- **adam_epsilon** – (float) the epsilon value for the adam optimizer
- **val_interval** – (int) Report training and validation losses every n epochs. By default, every 10th of the maximum number of epochs.

Returns (BaseRLModel) the pretrained model

save(*save_path*)
Save the current parameters to file

Parameters **save_path** – (str or file-like object) the save location

set_env(*env*)
Checks the validity of the environment, and if it is coherent, set it as the current environment.

Parameters **env** – (Gym Environment) The environment for learning a policy

setup_model()
Create all the functions and tensorflow graphs necessary to train the model

1.14 Policy Networks

Stable-baselines provides a set of default policies, that can be used with most action spaces. To customize the default policies, you can specify the `policy_kwargs` parameter to the model class you use. Those kwargs are then passed to the policy on instantiation (see *Custom Policy Network* for an example). If you need more control on the policy architecture, you can also create a custom policy (see *Custom Policy Network*).

Note: CnnPolicies are for images only. MlpPolicies are made for other type of features (e.g. robot joints)

Warning: For all algorithms (except DDPG, TD3 and SAC), continuous actions are clipped during training and testing (to avoid out of bound error).

Available Policies

MlpPolicy	Policy object that implements actor critic, using a MLP (2 layers of 64)
MlpLstmPolicy	Policy object that implements actor critic, using LSTMs with a MLP feature extraction
MlpLnLstmPolicy	Policy object that implements actor critic, using a layer normalized LSTMs with a MLP feature extraction
CnnPolicy	Policy object that implements actor critic, using a CNN (the nature CNN)
CnnLstmPolicy	Policy object that implements actor critic, using LSTMs with a CNN feature extraction
CnnLnLstmPolicy	Policy object that implements actor critic, using a layer normalized LSTMs with a CNN feature extraction

1.14.1 Base Classes

class stable_baselines.common.policies.**BasePolicy**(*sess*, *ob_space*, *ac_space*, *n_env*, *n_steps*, *n_batch*, *reuse=False*, *scale=False*, *obs_phs=None*, *add_action_ph=False*)

The base policy object

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batches to run (n_envs * n_steps)
- **reuse** – (bool) If the policy is reusable or not
- **scale** – (bool) whether or not to scale the input
- **obs_phs** – (TensorFlow Tensor, TensorFlow Tensor) a tuple containing an override for observation placeholder and the processed observation placeholder respectivly
- **add_action_ph** – (bool) whether or not to create an action placeholder

action_ph
tf.Tensor: placeholder for actions, shape (self.n_batch,) + self.ac_space.shape.

initial_state
The initial state of the policy. For feedforward policies, None. For a recurrent policy, a NumPy array of shape (self.n_env,) + state_shape.

is_discrete
bool: is action space discrete.

obs_ph
tf.Tensor: placeholder for observations, shape (self.n_batch,) + self.ob_space.shape.

proba_step(*obs*, *state=None*, *mask=None*)
Returns the action probability for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) the action probability

processed_obs
tf.Tensor: processed observations, shape (self.n_batch,) + self.ob_space.shape.

The form of processing depends on the type of the observation space, and the parameters whether scale is passed to the constructor; see observation_input for more information.

step(*obs*, *state=None*, *mask=None*)
Returns the policy for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float], [float], [float], [float]) actions, values, states, neglogp

class stable_baselines.common.policies.**ActorCriticPolicy**(*sess*, *ob_space*, *ac_space*, *n_env*, *n_steps*, *n_batch*, *reuse=False*, *scale=False*)

Policy object that implements actor critic

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **reuse** – (bool) If the policy is reusable or not
- **scale** – (bool) whether or not to scale the input

action
tf.Tensor: stochastic action, of shape (self.n_batch,) + self.ac_space.shape.

deterministic_action
tf.Tensor: deterministic action, of shape (self.n_batch,) + self.ac_space.shape.

neglogp
tf.Tensor: negative log likelihood of the action sampled by self.action.

pdtype
ProbabilityDistributionType: type of the distribution for stochastic actions.

policy
tf.Tensor: policy output, e.g. logits.

policy_proba
tf.Tensor: parameters of the probability distribution. Depends on pdtype.

proba_distribution
ProbabilityDistribution: distribution of stochastic actions.

step(*obs*, *state=None*, *mask=None*, *deterministic=False*)
Returns the policy for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)
- **deterministic** – (bool) Whether or not to return deterministic actions.

Returns ([float], [float], [float], [float]) actions, values, states, neglogp

value(*obs*, *state=None*, *mask=None*)
Returns the value for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) The associated value of the action

value_flat
tf.Tensor: value estimate, of shape (self.n_batch,)

value_fn
tf.Tensor: value estimate, of shape (self.n_batch, 1)

class stable_baselines.common.policies.**FeedForwardPolicy**(*sess*, *ob_space*, *ac_space*, *n_env*, *n_steps*, *n_batch*, *reuse=False*, *layers=None*, *net_arch=None*, *act_fun=<MagicMock id='139846110877232'>*, *cnn_extractor=<function nature_cnn>*, *feature_extraction='cnn'*, ***kwargs*)
Policy object that implements actor critic, using a feed forward neural network.

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **reuse** – (bool) If the policy is reusable or not
- **layers** – ([int]) (deprecated, use net_arch instead) The size of the Neural network for the policy (if None, default to [64, 64])
- **net_arch** – (list) Specification of the actor-critic policy network architecture (see mlp_extractor documentation for details).
- **act_fun** – (tf.func) the activation function to use in the neural network.
- **cnn_extractor** – (function (TensorFlow Tensor, `**kwargs`): (TensorFlow Tensor)) the CNN feature extraction
- **feature_extraction** – (str) The feature extraction type ("cnn" or "mlp")
- **kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

proba_step(*obs*, *state=None*, *mask=None*)
Returns the action probability for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) the action probability

step(*obs*, *state=None*, *mask=None*, *deterministic=False*)

Returns the policy for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)
- **deterministic** – (bool) Whether or not to return deterministic actions.

Returns ([float], [float], [float], [float]) actions, values, states, neglogp

value(*obs*, *state=None*, *mask=None*)

Returns the value for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) The associated value of the action

class stable_baselines.common.policies.**LstmPolicy**(*sess*, *ob_space*, *ac_space*, *n_env*, *n_steps*, *n_batch*, *n_lstm=256*, *reuse=False*, *layers=None*, *net_arch=None*, *act_fun=<MagicMock id='139846110936648'>*, *cnn_extractor=<function nature_cnn>*, *layer_norm=False*, *feature_extraction='cnn'*, ***kwargs*)

Policy object that implements actor critic, using LSTMs.

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **n_lstm** – (int) The number of LSTM cells (for recurrent policies)
- **reuse** – (bool) If the policy is reusable or not

- **layers** – ([int]) The size of the Neural network before the LSTM layer (if None, default to [64, 64])
- **net_arch** – (list) Specification of the actor-critic policy network architecture. Notation similar to the format described in mlp_extractor but with additional support for a 'lstm' entry in the shared network part.
- **act_fun** – (tf.func) the activation function to use in the neural network.
- **cnn_extractor** – (function (TensorFlow Tensor, `**kwargs`): (TensorFlow Tensor)) the CNN feature extraction
- **layer_norm** – (bool) Whether or not to use layer normalizing LSTMs
- **feature_extraction** – (str) The feature extraction type ("cnn" or "mlp")
- **kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

proba_step(*obs*, *state=None*, *mask=None*)
Returns the action probability for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) the action probability

step(*obs*, *state=None*, *mask=None*, *deterministic=False*)
Returns the policy for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)
- **deterministic** – (bool) Whether or not to return deterministic actions.

Returns ([float], [float], [float], [float]) actions, values, states, neglogp

value(*obs*, *state=None*, *mask=None*)
Cf base class doc.

1.14.2 MLP Policies

class stable_baselines.common.policies.**MlpPolicy**(*sess*, *ob_space*, *ac_space*, *n_env*, *n_steps*, *n_batch*, *reuse=False*, ***_kwargs*)
Policy object that implements actor critic, using a MLP (2 layers of 64)

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run

- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **reuse** – (bool) If the policy is reusable or not
- **_kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

class stable_baselines.common.policies.**MlpLstmPolicy**(*sess*, *ob_space*, *ac_space*, *n_env*, *n_steps*, *n_batch*, *n_lstm=256*, *reuse=False*, ***_kwargs*)

Policy object that implements actor critic, using LSTMs with a MLP feature extraction

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **n_lstm** – (int) The number of LSTM cells (for recurrent policies)
- **reuse** – (bool) If the policy is reusable or not
- **kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

class stable_baselines.common.policies.**MlpLnLstmPolicy**(*sess*, *ob_space*, *ac_space*, *n_env*, *n_steps*, *n_batch*, *n_lstm=256*, *reuse=False*, ***_kwargs*)

Policy object that implements actor critic, using a layer normalized LSTMs with a MLP feature extraction

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **n_lstm** – (int) The number of LSTM cells (for recurrent policies)
- **reuse** – (bool) If the policy is reusable or not
- **kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

1.14.3 CNN Policies

class stable_baselines.common.policies.**CnnPolicy**(*sess*, *ob_space*, *ac_space*, *n_env*, *n_steps*, *n_batch*, *reuse=False*, ***_kwargs*)

Policy object that implements actor critic, using a CNN (the nature CNN)

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **reuse** – (bool) If the policy is reusable or not
- **_kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

class stable_baselines.common.policies.**CnnLstmPolicy**(*sess*, *ob_space*, *ac_space*, *n_env*, *n_steps*, *n_batch*, *n_lstm=256*, *reuse=False*, ***_kwargs*)

Policy object that implements actor critic, using LSTMs with a CNN feature extraction

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **n_lstm** – (int) The number of LSTM cells (for recurrent policies)
- **reuse** – (bool) If the policy is reusable or not
- **kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

class stable_baselines.common.policies.**CnnLnLstmPolicy**(*sess*, *ob_space*, *ac_space*, *n_env*, *n_steps*, *n_batch*, *n_lstm=256*, *reuse=False*, ***_kwargs*)

Policy object that implements actor critic, using a layer normalized LSTMs with a CNN feature extraction

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **n_lstm** – (int) The number of LSTM cells (for recurrent policies)
- **reuse** – (bool) If the policy is reusable or not
- **kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

1.15 A2C

A synchronous, deterministic variant of Asynchronous Advantage Actor Critic (A3C). It uses multiple workers to avoid the use of a replay buffer.

1.15.1 Notes

- Original paper: https://arxiv.org/abs/1602.01783
- OpenAI blog post: https://openai.com/blog/baselines-acktr-a2c/
- `python -m stable_baselines.a2c.run_atari` runs the algorithm for 40M frames = 10M timesteps on an Atari game. See help (`-h`) for more options.
- `python -m stable_baselines.a2c.run_mujoco` runs the algorithm for 1M frames on a Mujoco environment.

1.15.2 Can I use?

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

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

1.15.3 Example

Train a A2C agent on *CartPole-v1* using 4 processes.

```
import gym

from stable_baselines.common.policies import MlpPolicy
from stable_baselines.common.vec_env import SubprocVecEnv
from stable_baselines import A2C

# multiprocess environment
n_cpu = 4
env = SubprocVecEnv([lambda: gym.make('CartPole-v1') for i in range(n_cpu)])

model = A2C(MlpPolicy, env, verbose=1)
model.learn(total_timesteps=25000)
model.save("a2c_cartpole")

del model # remove to demonstrate saving and loading

model = A2C.load("a2c_cartpole")
```

(continues on next page)

(continued from previous page)

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

1.15.4 Parameters

class stable_baselines.a2c.**A2C**(*policy*, *env*, *gamma=0.99*, *n_steps=5*, *vf_coef=0.25*, *ent_coef=0.01*, *max_grad_norm=0.5*, *learning_rate=0.0007*, *alpha=0.99*, *epsilon=1e-05*, *lr_schedule='constant'*, *verbose=0*, *tensorboard_log=None*, *_init_setup_model=True*, *policy_kwargs=None*, *full_tensorboard_log=False*)

The A2C (Advantage Actor Critic) model class, https://arxiv.org/abs/1602.01783

Parameters

- **policy** – (ActorCriticPolicy or str) The policy model to use (MlpPolicy, CnnPolicy, CnnLstmPolicy, ...)
- **env** – (Gym environment or str) The environment to learn from (if registered in Gym, can be str)
- **gamma** – (float) Discount factor
- **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)
- **vf_coef** – (float) Value function coefficient for the loss calculation
- **ent_coef** – (float) Entropy coefficient for the loss calculation
- **max_grad_norm** – (float) The maximum value for the gradient clipping
- **learning_rate** – (float) The learning rate
- **alpha** – (float) RMSProp decay parameter (default: 0.99)
- **epsilon** – (float) RMSProp epsilon (stabilizes square root computation in denominator of RMSProp update) (default: 1e-5)
- **lr_schedule** – (str) The type of scheduler for the learning rate update ('linear', 'constant', 'double_linear_con', 'middle_drop' or 'double_middle_drop')
- **verbose** – (int) the verbosity level: 0 none, 1 training information, 2 tensorflow debug
- **tensorboard_log** – (str) the log location for tensorboard (if None, no logging)
- **_init_setup_model** – (bool) Whether or not to build the network at the creation of the instance (used only for loading)
- **policy_kwargs** – (dict) additional arguments to be passed to the policy on creation
- **full_tensorboard_log** – (bool) enable additional logging when using tensorboard WARNING: this logging can take a lot of space quickly

action_probability(*observation*, *state=None*, *mask=None*, *actions=None*, *logp=False*)

If `actions` is `None`, then get the model's action probability distribution from a given observation.

Depending on the action space the output is:

- Discrete: probability for each possible action

- Box: mean and standard deviation of the action output

However if `actions` is not `None`, this function will return the probability that the given actions are taken with the given parameters (observation, state, ...) on this model. For discrete action spaces, it returns the probability mass; for continuous action spaces, the probability density. This is since the probability mass will always be zero in continuous spaces, see http://blog.christianperone.com/2019/01/ for a good explanation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (np.ndarray) The last states (can be None, used in recurrent policies)
- **mask** – (np.ndarray) The last masks (can be None, used in recurrent policies)
- **actions** – (np.ndarray) (OPTIONAL) For calculating the likelihood that the given actions are chosen by the model for each of the given parameters. Must have the same number of actions and observations. (set to None to return the complete action probability distribution)
- **logp** – (bool) (OPTIONAL) When specified with actions, returns probability in log-space. This has no effect if actions is None.

Returns (np.ndarray) the model's (log) action probability

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

Returns (Gym Environment) The current environment

get_parameter_list()
Get tensorflow Variables of model's parameters

This includes all variables necessary for continuing training (saving / loading).

Returns (list) List of tensorflow Variables

get_parameters()
Get current model parameters as dictionary of variable name -> ndarray.

Returns (OrderedDict) Dictionary of variable name -> ndarray of model's parameters.

learn(*total_timesteps*, *callback=None*, *seed=None*, *log_interval=100*, *tb_log_name='A2C'*, *reset_num_timesteps=True*)
Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples to train on
- **seed** – (int) The initial seed for training, if None: keep current seed
- **callback** – (function (dict, dict)) -> boolean function called at every steps with state of the algorithm. It takes the local and global variables. If it returns False, training is aborted.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for tensorboard log
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseRLModel) the trained model

classmethod load(*load_path*, *env=None*, ***kwargs*)
Load the model from file

Parameters

- **load_path** – (str or file-like) the saved parameter location
- **env** – (Gym Envrionment) the new environment to run the loaded model on (can be None if you only need prediction from a trained model)
- **kwargs** – extra arguments to change the model when loading

load_parameters(*load_path_or_dict*, *exact_match=True*)
Load model parameters from a file or a dictionary

Dictionary keys should be tensorflow variable names, which can be obtained with `get_parameters` function. If `exact_match` is True, dictionary should contain keys for all model's parameters, otherwise RunTimeError is raised. If False, only variables included in the dictionary will be updated.

This does not load agent's hyper-parameters.

Warning: This function does not update trainer/optimizer variables (e.g. momentum). As such training after using this function may lead to less-than-optimal results.

Parameters

- **load_path_or_dict** – (str or file-like or dict) Save parameter location or dict of parameters as variable.name -> ndarrays to be loaded.
- **exact_match** – (bool) If True, expects load dictionary to contain keys for all variables in the model. If False, loads parameters only for variables mentioned in the dictionary. Defaults to True.

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

Parameters

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

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

pretrain(*dataset*, *n_epochs=10*, *learning_rate=0.0001*, *adam_epsilon=1e-08*, *val_interval=None*)
Pretrain a model using behavior cloning: supervised learning given an expert dataset.

NOTE: only Box and Discrete spaces are supported for now.

Parameters

- **dataset** – (ExpertDataset) Dataset manager
- **n_epochs** – (int) Number of iterations on the training set
- **learning_rate** – (float) Learning rate
- **adam_epsilon** – (float) the epsilon value for the adam optimizer
- **val_interval** – (int) Report training and validation losses every n epochs. By default, every 10th of the maximum number of epochs.

Returns (BaseRLModel) the pretrained model

save(*save_path*)
Save the current parameters to file

Parameters **save_path** – (str or file-like object) the save location

set_env(*env*)
Checks the validity of the environment, and if it is coherent, set it as the current environment.

Parameters **env** – (Gym Environment) The environment for learning a policy

setup_model()
Create all the functions and tensorflow graphs necessary to train the model

1.16 ACER

Sample Efficient Actor-Critic with Experience Replay (ACER) combines several ideas of previous algorithms: it uses multiple workers (as A2C), implements a replay buffer (as in DQN), uses Retrace for Q-value estimation, importance sampling and a trust region.

1.16.1 Notes

- Original paper: https://arxiv.org/abs/1611.01224
- `python -m stable_baselines.acer.run_atari` runs the algorithm for 40M frames = 10M timesteps on an Atari game. See help (`-h`) for more options.

1.16.2 Can I use?

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

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

1.16.3 Example

```
import gym

from stable_baselines.common.policies import MlpPolicy, MlpLstmPolicy, MlpLnLstmPolicy
from stable_baselines.common.vec_env import SubprocVecEnv
from stable_baselines import ACER

# multiprocess environment
n_cpu = 4
env = SubprocVecEnv([lambda: gym.make('CartPole-v1') for i in range(n_cpu)])
```

(continues on next page)

(continued from previous page)

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

del model # remove to demonstrate saving and loading

model = ACER.load("acer_cartpole")

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

1.16.4 Parameters

class stable_baselines.acer.**ACER**(*policy*, *env*, *gamma=0.99*, *n_steps=20*, *num_procs=1*, *q_coef=0.5*, *ent_coef=0.01*, *max_grad_norm=10*, *learning_rate=0.0007*, *lr_schedule='linear'*, *rprop_alpha=0.99*, *rprop_epsilon=1e-05*, *buffer_size=5000*, *replay_ratio=4*, *replay_start=1000*, *correction_term=10.0*, *trust_region=True*, *alpha=0.99*, *delta=1*, *verbose=0*, *tensorboard_log=None*, *_init_setup_model=True*, *policy_kwargs=None*, *full_tensorboard_log=False*)

The ACER (Actor-Critic with Experience Replay) model class, https://arxiv.org/abs/1611.01224

Parameters

- **policy** – (ActorCriticPolicy or str) The policy model to use (MlpPolicy, CnnPolicy, CnnLstmPolicy, ...)
- **env** – (Gym environment or str) The environment to learn from (if registered in Gym, can be str)
- **gamma** – (float) The discount value
- **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)
- **num_procs** – (int) The number of threads for TensorFlow operations
- **q_coef** – (float) The weight for the loss on the Q value
- **ent_coef** – (float) The weight for the entropic loss
- **max_grad_norm** – (float) The clipping value for the maximum gradient
- **learning_rate** – (float) The initial learning rate for the RMS prop optimizer
- **lr_schedule** – (str) The type of scheduler for the learning rate update ('linear', 'constant', 'double_linear_con', 'middle_drop' or 'double_middle_drop')
- **rprop_epsilon** – (float) RMSProp epsilon (stabilizes square root computation in denominator of RMSProp update) (default: 1e-5)
- **rprop_alpha** – (float) RMSProp decay parameter (default: 0.99)
- **buffer_size** – (int) The buffer size in number of steps

- **replay_ratio** – (float) The number of replay learning per on policy learning on average, using a poisson distribution
- **replay_start** – (int) The minimum number of steps in the buffer, before learning replay
- **correction_term** – (float) Importance weight clipping factor (default: 10)
- **trust_region** – (bool) Whether or not algorithms estimates the gradient KL divergence between the old and updated policy and uses it to determine step size (default: True)
- **alpha** – (float) The decay rate for the Exponential moving average of the parameters
- **delta** – (float) max KL divergence between the old policy and updated policy (default: 1)
- **verbose** – (int) the verbosity level: 0 none, 1 training information, 2 tensorflow debug
- **tensorboard_log** – (str) the log location for tensorboard (if None, no logging)
- **_init_setup_model** – (bool) Whether or not to build the network at the creation of the instance
- **policy_kwargs** – (dict) additional arguments to be passed to the policy on creation
- **full_tensorboard_log** – (bool) enable additional logging when using tensorboard WARNING: this logging can take a lot of space quickly

action_probability(*observation*, *state=None*, *mask=None*, *actions=None*, *logp=False*)

If `actions` is `None`, then get the model's action probability distribution from a given observation.

Depending on the action space the output is:

- Discrete: probability for each possible action
- Box: mean and standard deviation of the action output

However if `actions` is not `None`, this function will return the probability that the given actions are taken with the given parameters (observation, state, ...) on this model. For discrete action spaces, it returns the probability mass; for continuous action spaces, the probability density. This is since the probability mass will always be zero in continuous spaces, see http://blog.christianperone.com/2019/01/ for a good explanation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (np.ndarray) The last states (can be None, used in recurrent policies)
- **mask** – (np.ndarray) The last masks (can be None, used in recurrent policies)
- **actions** – (np.ndarray) (OPTIONAL) For calculating the likelihood that the given actions are chosen by the model for each of the given parameters. Must have the same number of actions and observations. (set to None to return the complete action probability distribution)
- **logp** – (bool) (OPTIONAL) When specified with actions, returns probability in log-space. This has no effect if actions is None.

Returns (np.ndarray) the model's (log) action probability

get_env()

returns the current environment (can be None if not defined)

Returns (Gym Environment) The current environment

get_parameter_list()
Get tensorflow Variables of model's parameters

This includes all variables necessary for continuing training (saving / loading).

Returns (list) List of tensorflow Variables

get_parameters()
Get current model parameters as dictionary of variable name -> ndarray.

Returns (OrderedDict) Dictionary of variable name -> ndarray of model's parameters.

learn(*total_timesteps*, *callback=None*, *seed=None*, *log_interval=100*, *tb_log_name='ACER'*, *reset_num_timesteps=True*)
Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples to train on
- **seed** – (int) The initial seed for training, if None: keep current seed
- **callback** – (function (dict, dict)) -> boolean function called at every steps with state of the algorithm. It takes the local and global variables. If it returns False, training is aborted.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for tensorboard log
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseRLModel) the trained model

classmethod load(*load_path*, *env=None*, ***kwargs*)
Load the model from file

Parameters

- **load_path** – (str or file-like) the saved parameter location
- **env** – (Gym Envrionment) the new environment to run the loaded model on (can be None if you only need prediction from a trained model)
- **kwargs** – extra arguments to change the model when loading

load_parameters(*load_path_or_dict*, *exact_match=True*)
Load model parameters from a file or a dictionary

Dictionary keys should be tensorflow variable names, which can be obtained with `get_parameters` function. If `exact_match` is True, dictionary should contain keys for all model's parameters, otherwise RunTimeError is raised. If False, only variables included in the dictionary will be updated.

This does not load agent's hyper-parameters.

Warning: This function does not update trainer/optimizer variables (e.g. momentum). As such training after using this function may lead to less-than-optimal results.

Parameters

- **load_path_or_dict** – (str or file-like or dict) Save parameter location or dict of parameters as variable.name -> ndarrays to be loaded.

- **exact_match** – (bool) If True, expects load dictionary to contain keys for all variables in the model. If False, loads parameters only for variables mentioned in the dictionary. Defaults to True.

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

Parameters

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

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

pretrain(*dataset*, *n_epochs=10*, *learning_rate=0.0001*, *adam_epsilon=1e-08*, *val_interval=None*)
Pretrain a model using behavior cloning: supervised learning given an expert dataset.

NOTE: only Box and Discrete spaces are supported for now.

Parameters

- **dataset** – (ExpertDataset) Dataset manager
- **n_epochs** – (int) Number of iterations on the training set
- **learning_rate** – (float) Learning rate
- **adam_epsilon** – (float) the epsilon value for the adam optimizer
- **val_interval** – (int) Report training and validation losses every n epochs. By default, every 10th of the maximum number of epochs.

Returns (BaseRLModel) the pretrained model

save(*save_path*)
Save the current parameters to file

Parameters **save_path** – (str or file-like object) the save location

set_env(*env*)
Checks the validity of the environment, and if it is coherent, set it as the current environment.

Parameters **env** – (Gym Environment) The environment for learning a policy

setup_model()
Create all the functions and tensorflow graphs necessary to train the model

1.17 ACKTR

Actor Critic using Kronecker-Factored Trust Region (ACKTR) uses Kronecker-factored approximate curvature (K-FAC) for trust region optimization.

1.17.1 Notes

- Original paper: https://arxiv.org/abs/1708.05144
- Baselines blog post: https://blog.openai.com/baselines-acktr-a2c/
- `python -m stable_baselines.acktr.run_atari` runs the algorithm for 40M frames = 10M timesteps on an Atari game. See help (`-h`) for more options.

1.17.2 Can I use?

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

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

1.17.3 Example

```
import gym

from stable_baselines.common.policies import MlpPolicy, MlpLstmPolicy, MlpLnLstmPolicy
from stable_baselines.common.vec_env import SubprocVecEnv
from stable_baselines import ACKTR

# multiprocess environment
n_cpu = 4
env = SubprocVecEnv([lambda: gym.make('CartPole-v1') for i in range(n_cpu)])

model = ACKTR(MlpPolicy, env, verbose=1)
model.learn(total_timesteps=25000)
model.save("acktr_cartpole")

del model # remove to demonstrate saving and loading

model = ACKTR.load("acktr_cartpole")

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

1.17.4 Parameters

class stable_baselines.acktr.**ACKTR**(*policy*, *env*, *gamma=0.99*, *nprocs=1*, *n_steps=20*, *ent_coef=0.01*, *vf_coef=0.25*, *vf_fisher_coef=1.0*, *learning_rate=0.25*, *max_grad_norm=0.5*, *kfac_clip=0.001*, *lr_schedule='linear'*, *verbose=0*, *tensorboard_log=None*, *_init_setup_model=True*, *async_eigen_decomp=False*, *policy_kwargs=None*, *full_tensorboard_log=False*)

The ACKTR (Actor Critic using Kronecker-Factored Trust Region) model class, https://arxiv.org/abs/1708.05144

Parameters

- **policy** – (ActorCriticPolicy or str) The policy model to use (MlpPolicy, CnnPolicy, CnnLstmPolicy, ...)
- **env** – (Gym environment or str) The environment to learn from (if registered in Gym, can be str)
- **gamma** – (float) Discount factor
- **nprocs** – (int) The number of threads for TensorFlow operations
- **n_steps** – (int) The number of steps to run for each environment
- **ent_coef** – (float) The weight for the entropic loss
- **vf_coef** – (float) The weight for the loss on the value function
- **vf_fisher_coef** – (float) The weight for the fisher loss on the value function
- **learning_rate** – (float) The initial learning rate for the RMS prop optimizer
- **max_grad_norm** – (float) The clipping value for the maximum gradient
- **kfac_clip** – (float) gradient clipping for Kullback-Leibler
- **lr_schedule** – (str) The type of scheduler for the learning rate update ('linear', 'constant', 'double_linear_con', 'middle_drop' or 'double_middle_drop')
- **verbose** – (int) the verbosity level: 0 none, 1 training information, 2 tensorflow debug
- **tensorboard_log** – (str) the log location for tensorboard (if None, no logging)
- **_init_setup_model** – (bool) Whether or not to build the network at the creation of the instance
- **async_eigen_decomp** – (bool) Use async eigen decomposition
- **policy_kwargs** – (dict) additional arguments to be passed to the policy on creation
- **full_tensorboard_log** – (bool) enable additional logging when using tensorboard WARNING: this logging can take a lot of space quickly

action_probability(*observation*, *state=None*, *mask=None*, *actions=None*, *logp=False*)

If `actions` is `None`, then get the model's action probability distribution from a given observation.

Depending on the action space the output is:

- Discrete: probability for each possible action
- Box: mean and standard deviation of the action output

However if `actions` is not `None`, this function will return the probability that the given actions are taken with the given parameters (observation, state, ...) on this model. For discrete action spaces, it returns the probability mass; for continuous action spaces, the probability density. This is since the probability

mass will always be zero in continuous spaces, see http://blog.christianperone.com/2019/01/ for a good explanation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (np.ndarray) The last states (can be None, used in recurrent policies)
- **mask** – (np.ndarray) The last masks (can be None, used in recurrent policies)
- **actions** – (np.ndarray) (OPTIONAL) For calculating the likelihood that the given actions are chosen by the model for each of the given parameters. Must have the same number of actions and observations. (set to None to return the complete action probability distribution)
- **logp** – (bool) (OPTIONAL) When specified with actions, returns probability in log-space. This has no effect if actions is None.

Returns (np.ndarray) the model's (log) action probability

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

Returns (Gym Environment) The current environment

get_parameter_list()
Get tensorflow Variables of model's parameters

This includes all variables necessary for continuing training (saving / loading).

Returns (list) List of tensorflow Variables

get_parameters()
Get current model parameters as dictionary of variable name -> ndarray.

Returns (OrderedDict) Dictionary of variable name -> ndarray of model's parameters.

learn(*total_timesteps*, *callback=None*, *seed=None*, *log_interval=100*, *tb_log_name='ACKTR'*, *reset_num_timesteps=True*)
Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples to train on
- **seed** – (int) The initial seed for training, if None: keep current seed
- **callback** – (function (dict, dict)) -> boolean function called at every steps with state of the algorithm. It takes the local and global variables. If it returns False, training is aborted.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for tensorboard log
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseRLModel) the trained model

classmethod load(*load_path*, *env=None*, ***kwargs*)
Load the model from file

Parameters

- **load_path** – (str or file-like) the saved parameter location

- **env** – (Gym Envrionment) the new environment to run the loaded model on (can be None if you only need prediction from a trained model)
- **kwargs** – extra arguments to change the model when loading

load_parameters(*load_path_or_dict*, *exact_match=True*)
Load model parameters from a file or a dictionary

Dictionary keys should be tensorflow variable names, which can be obtained with `get_parameters` function. If `exact_match` is True, dictionary should contain keys for all model's parameters, otherwise RunTimeError is raised. If False, only variables included in the dictionary will be updated.

This does not load agent's hyper-parameters.

Warning: This function does not update trainer/optimizer variables (e.g. momentum). As such training after using this function may lead to less-than-optimal results.

Parameters

- **load_path_or_dict** – (str or file-like or dict) Save parameter location or dict of parameters as variable.name -> ndarrays to be loaded.
- **exact_match** – (bool) If True, expects load dictionary to contain keys for all variables in the model. If False, loads parameters only for variables mentioned in the dictionary. Defaults to True.

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

Parameters

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

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

pretrain(*dataset*, *n_epochs=10*, *learning_rate=0.0001*, *adam_epsilon=1e-08*, *val_interval=None*)
Pretrain a model using behavior cloning: supervised learning given an expert dataset.

NOTE: only Box and Discrete spaces are supported for now.

Parameters

- **dataset** – (ExpertDataset) Dataset manager
- **n_epochs** – (int) Number of iterations on the training set
- **learning_rate** – (float) Learning rate
- **adam_epsilon** – (float) the epsilon value for the adam optimizer
- **val_interval** – (int) Report training and validation losses every n epochs. By default, every 10th of the maximum number of epochs.

Returns (BaseRLModel) the pretrained model

save(*save_path*)
Save the current parameters to file

Parameters **save_path** – (str or file-like object) the save location

set_env(*env*)
Checks the validity of the environment, and if it is coherent, set it as the current environment.

Parameters **env** – (Gym Environment) The environment for learning a policy

setup_model()
Create all the functions and tensorflow graphs necessary to train the model

1.18 DDPG

Deep Deterministic Policy Gradient (DDPG)

Warning: The DDPG model does not support `stable_baselines.common.policies` because it uses q-value instead of value estimation, as a result it must use its own policy models (see *DDPG Policies*).

Available Policies

MlpPolicy	Policy object that implements actor critic, using a MLP (2 layers of 64)
LnMlpPolicy	Policy object that implements actor critic, using a MLP (2 layers of 64), with layer normalisation
CnnPolicy	Policy object that implements actor critic, using a CNN (the nature CNN)
LnCnnPolicy	Policy object that implements actor critic, using a CNN (the nature CNN), with layer normalisation

1.18.1 Notes

- Original paper: https://arxiv.org/abs/1509.02971
- Baselines post: https://blog.openai.com/better-exploration-with-parameter-noise/
- `python -m stable_baselines.ddpg.main` runs the algorithm for 1M frames = 10M timesteps on a Mujoco environment. See help (`-h`) for more options.

1.18.2 Can I use?

- Recurrent policies:
- Multi processing: ✓ (using MPI)
- Gym spaces:

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

1.18.3 Example

```
import gym
import numpy as np

from stable_baselines.ddpg.policies import MlpPolicy
from stable_baselines.common.vec_env import DummyVecEnv
from stable_baselines.ddpg.noise import NormalActionNoise, OrnsteinUhlenbeckActionNoise, AdaptiveParamNoiseSpec
from stable_baselines import DDPG

env = gym.make('MountainCarContinuous-v0')
env = DummyVecEnv([lambda: env])

# the noise objects for DDPG
n_actions = env.action_space.shape[-1]
param_noise = None
action_noise = OrnsteinUhlenbeckActionNoise(mean=np.zeros(n_actions), sigma=float(0.5) * np.ones(n_actions))

model = DDPG(MlpPolicy, env, verbose=1, param_noise=param_noise, action_noise=action_noise)
model.learn(total_timesteps=400000)
model.save("ddpg_mountain")

del model # remove to demonstrate saving and loading

model = DDPG.load("ddpg_mountain")

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

1.18.4 Parameters

class stable_baselines.ddpg.**DDPG**(*policy*, *env*, *gamma=0.99*, *memory_policy=None*, *eval_env=None*, *nb_train_steps=50*, *nb_rollout_steps=100*, *nb_eval_steps=100*, *param_noise=None*, *action_noise=None*, *normalize_observations=False*, *tau=0.001*, *batch_size=128*, *param_noise_adaption_interval=50*, *normalize_returns=False*, *enable_popart=False*, *observation_range=(-5.0*, *5.0)*, *critic_l2_reg=0.0*, *return_range=(-inf*, *inf)*, *actor_lr=0.0001*, *critic_lr=0.001*, *clip_norm=None*, *reward_scale=1.0*, *render=False*, *render_eval=False*, *memory_limit=None*, *buffer_size=50000*, *random_exploration=0.0*, *verbose=0*, *tensorboard_log=None*, *_init_setup_model=True*, *policy_kwargs=None*, *full_tensorboard_log=False*)

Deep Deterministic Policy Gradient (DDPG) model

DDPG: https://arxiv.org/pdf/1509.02971.pdf

Parameters

- **policy** – (DDPGPolicy or str) The policy model to use (MlpPolicy, CnnPolicy, LnMlpPolicy, ...)
- **env** – (Gym environment or str) The environment to learn from (if registered in Gym, can be str)
- **gamma** – (float) the discount factor
- **memory_policy** – (ReplayBuffer) the replay buffer (if None, default to baselines.deepq.replay_buffer.ReplayBuffer)

  Deprecated since version 2.6.0: This parameter will be removed in a future version
- **eval_env** – (Gym Environment) the evaluation environment (can be None)
- **nb_train_steps** – (int) the number of training steps
- **nb_rollout_steps** – (int) the number of rollout steps
- **nb_eval_steps** – (int) the number of evalutation steps
- **param_noise** – (AdaptiveParamNoiseSpec) the parameter noise type (can be None)
- **action_noise** – (ActionNoise) the action noise type (can be None)
- **param_noise_adaption_interval** – (int) apply param noise every N steps
- **tau** – (float) the soft update coefficient (keep old values, between 0 and 1)
- **normalize_returns** – (bool) should the critic output be normalized
- **enable_popart** – (bool) enable pop-art normalization of the critic output (https://arxiv.org/pdf/1602.07714.pdf), normalize_returns must be set to True.
- **normalize_observations** – (bool) should the observation be normalized
- **batch_size** – (int) the size of the batch for learning the policy
- **observation_range** – (tuple) the bounding values for the observation
- **return_range** – (tuple) the bounding values for the critic output
- **critic_l2_reg** – (float) l2 regularizer coefficient

- **actor_lr** – (float) the actor learning rate
- **critic_lr** – (float) the critic learning rate
- **clip_norm** – (float) clip the gradients (disabled if None)
- **reward_scale** – (float) the value the reward should be scaled by
- **render** – (bool) enable rendering of the environment
- **render_eval** – (bool) enable rendering of the evalution environment
- **memory_limit** – (int) the max number of transitions to store, size of the replay buffer

  Deprecated since version 2.6.0: Use *buffer_size* instead.
- **buffer_size** – (int) the max number of transitions to store, size of the replay buffer
- **random_exploration** – (float) Probability of taking a random action (as in an epsilon-greedy strategy) This is not needed for DDPG normally but can help exploring when using HER + DDPG. This hack was present in the original OpenAI Baselines repo (DDPG + HER)
- **verbose** – (int) the verbosity level: 0 none, 1 training information, 2 tensorflow debug
- **tensorboard_log** – (str) the log location for tensorboard (if None, no logging)
- **_init_setup_model** – (bool) Whether or not to build the network at the creation of the instance
- **policy_kwargs** – (dict) additional arguments to be passed to the policy on creation
- **full_tensorboard_log** – (bool) enable additional logging when using tensorboard WARNING: this logging can take a lot of space quickly

action_probability(*observation*, *state=None*, *mask=None*, *actions=None*, *logp=False*)

If `actions` is `None`, then get the model's action probability distribution from a given observation.

Depending on the action space the output is:

- Discrete: probability for each possible action
- Box: mean and standard deviation of the action output

However if `actions` is not `None`, this function will return the probability that the given actions are taken with the given parameters (observation, state, ...) on this model. For discrete action spaces, it returns the probability mass; for continuous action spaces, the probability density. This is since the probability mass will always be zero in continuous spaces, see http://blog.christianperone.com/2019/01/ for a good explanation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (np.ndarray) The last states (can be None, used in recurrent policies)
- **mask** – (np.ndarray) The last masks (can be None, used in recurrent policies)
- **actions** – (np.ndarray) (OPTIONAL) For calculating the likelihood that the given actions are chosen by the model for each of the given parameters. Must have the same number of actions and observations. (set to None to return the complete action probability distribution)
- **logp** – (bool) (OPTIONAL) When specified with actions, returns probability in log-space. This has no effect if actions is None.

Returns (np.ndarray) the model's (log) action probability

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

Returns (Gym Environment) The current environment

get_parameter_list()
Get tensorflow Variables of model's parameters

This includes all variables necessary for continuing training (saving / loading).

Returns (list) List of tensorflow Variables

get_parameters()
Get current model parameters as dictionary of variable name -> ndarray.

Returns (OrderedDict) Dictionary of variable name -> ndarray of model's parameters.

learn(*total_timesteps*, *callback=None*, *seed=None*, *log_interval=100*, *tb_log_name='DDPG'*, *reset_num_timesteps=True*, *replay_wrapper=None*)
Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples to train on
- **seed** – (int) The initial seed for training, if None: keep current seed
- **callback** – (function (dict, dict)) -> boolean function called at every steps with state of the algorithm. It takes the local and global variables. If it returns False, training is aborted.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for tensorboard log
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseRLModel) the trained model

classmethod load(*load_path*, *env=None*, ***kwargs*)
Load the model from file

Parameters

- **load_path** – (str or file-like) the saved parameter location
- **env** – (Gym Envrionment) the new environment to run the loaded model on (can be None if you only need prediction from a trained model)
- **kwargs** – extra arguments to change the model when loading

load_parameters(*load_path_or_dict*, *exact_match=True*)
Load model parameters from a file or a dictionary

Dictionary keys should be tensorflow variable names, which can be obtained with `get_parameters` function. If `exact_match` is True, dictionary should contain keys for all model's parameters, otherwise RunTimeError is raised. If False, only variables included in the dictionary will be updated.

This does not load agent's hyper-parameters.

Warning: This function does not update trainer/optimizer variables (e.g. momentum). As such training after using this function may lead to less-than-optimal results.

Parameters

- **load_path_or_dict** – (str or file-like or dict) Save parameter location or dict of parameters as variable.name -> ndarrays to be loaded.
- **exact_match** – (bool) If True, expects load dictionary to contain keys for all variables in the model. If False, loads parameters only for variables mentioned in the dictionary. Defaults to True.

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

Parameters

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

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

pretrain(*dataset*, *n_epochs=10*, *learning_rate=0.0001*, *adam_epsilon=1e-08*, *val_interval=None*)
Pretrain a model using behavior cloning: supervised learning given an expert dataset.

NOTE: only Box and Discrete spaces are supported for now.

Parameters

- **dataset** – (ExpertDataset) Dataset manager
- **n_epochs** – (int) Number of iterations on the training set
- **learning_rate** – (float) Learning rate
- **adam_epsilon** – (float) the epsilon value for the adam optimizer
- **val_interval** – (int) Report training and validation losses every n epochs. By default, every 10th of the maximum number of epochs.

Returns (BaseRLModel) the pretrained model

save(*save_path*)
Save the current parameters to file

Parameters **save_path** – (str or file-like object) the save location

set_env(*env*)
Checks the validity of the environment, and if it is coherent, set it as the current environment.

Parameters **env** – (Gym Environment) The environment for learning a policy

setup_model()
Create all the functions and tensorflow graphs necessary to train the model

1.18.5 DDPG Policies

class stable_baselines.ddpg.**MlpPolicy**(*sess*, *ob_space*, *ac_space*, *n_env*, *n_steps*, *n_batch*, *reuse=False*, ***_kwargs*)
Policy object that implements actor critic, using a MLP (2 layers of 64)

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session

- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **reuse** – (bool) If the policy is reusable or not
- **_kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

action_ph
tf.Tensor: placeholder for actions, shape (self.n_batch,) + self.ac_space.shape.

initial_state
The initial state of the policy. For feedforward policies, None. For a recurrent policy, a NumPy array of shape (self.n_env,) + state_shape.

is_discrete
bool: is action space discrete.

make_actor(*obs=None*, *reuse=False*, *scope='pi'*)
creates an actor object

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name of the actor

Returns (TensorFlow Tensor) the output tensor

make_critic(*obs=None*, *action=None*, *reuse=False*, *scope='qf'*)
creates a critic object

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **action** – (TensorFlow Tensor) The action placeholder (can be None for default placeholder)
- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name of the critic

Returns (TensorFlow Tensor) the output tensor

obs_ph
tf.Tensor: placeholder for observations, shape (self.n_batch,) + self.ob_space.shape.

proba_step(*obs*, *state=None*, *mask=None*)
Returns the action probability for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) the action probability

processed_obs
tf.Tensor: processed observations, shape (self.n_batch,) + self.ob_space.shape.

The form of processing depends on the type of the observation space, and the parameters whether scale is passed to the constructor; see observation_input for more information.

step(*obs*, *state=None*, *mask=None*)
Returns the policy for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) actions

value(*obs*, *action*, *state=None*, *mask=None*)
Returns the value for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **action** – ([float] or [int]) The taken action
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) The associated value of the action

class stable_baselines.ddpg.**LnMlpPolicy**(*sess*, *ob_space*, *ac_space*, *n_env*, *n_steps*, *n_batch*, *reuse=False*, ***_kwargs*)
Policy object that implements actor critic, using a MLP (2 layers of 64), with layer normalisation

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **reuse** – (bool) If the policy is reusable or not
- **_kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

action_ph
tf.Tensor: placeholder for actions, shape (self.n_batch,) + self.ac_space.shape.

initial_state
The initial state of the policy. For feedforward policies, None. For a recurrent policy, a NumPy array of shape (self.n_env,) + state_shape.

is_discrete
bool: is action space discrete.

make_actor(*obs=None*, *reuse=False*, *scope='pi'*)
creates an actor object

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name of the actor

Returns (TensorFlow Tensor) the output tensor

make_critic(*obs=None*, *action=None*, *reuse=False*, *scope='qf'*)
creates a critic object

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **action** – (TensorFlow Tensor) The action placeholder (can be None for default placeholder)
- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name of the critic

Returns (TensorFlow Tensor) the output tensor

obs_ph
tf.Tensor: placeholder for observations, shape (self.n_batch,) + self.ob_space.shape.

proba_step(*obs*, *state=None*, *mask=None*)
Returns the action probability for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) the action probability

processed_obs
tf.Tensor: processed observations, shape (self.n_batch,) + self.ob_space.shape.

The form of processing depends on the type of the observation space, and the parameters whether scale is passed to the constructor; see observation_input for more information.

step(*obs*, *state=None*, *mask=None*)
Returns the policy for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) actions

value(*obs*, *action*, *state=None*, *mask=None*)
Returns the value for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **action** – ([float] or [int]) The taken action
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) The associated value of the action

class stable_baselines.ddpg.**CnnPolicy**(*sess*, *ob_space*, *ac_space*, *n_env*, *n_steps*, *n_batch*, *reuse=False*, ***_kwargs*)

Policy object that implements actor critic, using a CNN (the nature CNN)

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **reuse** – (bool) If the policy is reusable or not
- **_kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

action_ph
tf.Tensor: placeholder for actions, shape (self.n_batch,) + self.ac_space.shape.

initial_state
The initial state of the policy. For feedforward policies, None. For a recurrent policy, a NumPy array of shape (self.n_env,) + state_shape.

is_discrete
bool: is action space discrete.

make_actor(*obs=None*, *reuse=False*, *scope='pi'*)
creates an actor object

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name of the actor

Returns (TensorFlow Tensor) the output tensor

make_critic(*obs=None*, *action=None*, *reuse=False*, *scope='qf'*)
creates a critic object

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **action** – (TensorFlow Tensor) The action placeholder (can be None for default placeholder)

- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name of the critic

Returns (TensorFlow Tensor) the output tensor

obs_ph
tf.Tensor: placeholder for observations, shape (self.n_batch,) + self.ob_space.shape.

proba_step(*obs*, *state=None*, *mask=None*)
Returns the action probability for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) the action probability

processed_obs
tf.Tensor: processed observations, shape (self.n_batch,) + self.ob_space.shape.

The form of processing depends on the type of the observation space, and the parameters whether scale is passed to the constructor; see observation_input for more information.

step(*obs*, *state=None*, *mask=None*)
Returns the policy for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) actions

value(*obs*, *action*, *state=None*, *mask=None*)
Returns the value for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **action** – ([float] or [int]) The taken action
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) The associated value of the action

class stable_baselines.ddpg.**LnCnnPolicy**(*sess*, *ob_space*, *ac_space*, *n_env*, *n_steps*, *n_batch*, *reuse=False*, ***_kwargs*)
Policy object that implements actor critic, using a CNN (the nature CNN), with layer normalisation

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run

- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **reuse** – (bool) If the policy is reusable or not
- **_kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

action_ph
tf.Tensor: placeholder for actions, shape (self.n_batch,) + self.ac_space.shape.

initial_state
The initial state of the policy. For feedforward policies, None. For a recurrent policy, a NumPy array of shape (self.n_env,) + state_shape.

is_discrete
bool: is action space discrete.

make_actor(*obs=None*, *reuse=False*, *scope='pi'*)
creates an actor object

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name of the actor

Returns (TensorFlow Tensor) the output tensor

make_critic(*obs=None*, *action=None*, *reuse=False*, *scope='qf'*)
creates a critic object

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **action** – (TensorFlow Tensor) The action placeholder (can be None for default placeholder)
- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name of the critic

Returns (TensorFlow Tensor) the output tensor

obs_ph
tf.Tensor: placeholder for observations, shape (self.n_batch,) + self.ob_space.shape.

proba_step(*obs*, *state=None*, *mask=None*)
Returns the action probability for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) the action probability

processed_obs
tf.Tensor: processed observations, shape (self.n_batch,) + self.ob_space.shape.

The form of processing depends on the type of the observation space, and the parameters whether scale is passed to the constructor; see observation_input for more information.

step(*obs*, *state=None*, *mask=None*)
Returns the policy for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) actions

value(*obs*, *action*, *state=None*, *mask=None*)
Returns the value for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **action** – ([float] or [int]) The taken action
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) The associated value of the action

1.18.6 Action and Parameters Noise

class stable_baselines.ddpg.**AdaptiveParamNoiseSpec**(*initial_stddev=0.1*, *desired_action_stddev=0.1*, *adoption_coefficient=1.01*)
Implements adaptive parameter noise

Parameters

- **initial_stddev** – (float) the initial value for the standard deviation of the noise
- **desired_action_stddev** – (float) the desired value for the standard deviation of the noise
- **adoption_coefficient** – (float) the update coefficient for the standard deviation of the noise

adapt(*distance*)
update the standard deviation for the parameter noise

Parameters **distance** – (float) the noise distance applied to the parameters

get_stats()
return the standard deviation for the parameter noise

Returns (dict) the stats of the noise

class stable_baselines.ddpg.**NormalActionNoise**(*mean*, *sigma*)
A gaussian action noise

Parameters

- **mean** – (float) the mean value of the noise
- **sigma** – (float) the scale of the noise (std here)

reset()
call end of episode reset for the noise

class stable_baselines.ddpg.**OrnsteinUhlenbeckActionNoise**(*mean*, *sigma*, *theta=0.15*, *dt=0.01*, *initial_noise=None*)
A Ornstein Uhlenbeck action noise, this is designed to aproximate brownian motion with friction.

Based on http://math.stackexchange.com/questions/1287634/implementing-ornstein-uhlenbeck-in-matlab

Parameters

- **mean** – (float) the mean of the noise
- **sigma** – (float) the scale of the noise
- **theta** – (float) the rate of mean reversion
- **dt** – (float) the timestep for the noise
- **initial_noise** – ([float]) the initial value for the noise output, (if None: 0)

reset()
reset the Ornstein Uhlenbeck noise, to the initial position

1.18.7 Custom Policy Network

Similarly to the example given in the examples page. You can easily define a custom architecture for the policy network:

```
import gym

from stable_baselines.ddpg.policies import FeedForwardPolicy
from stable_baselines.common.vec_env import DummyVecEnv
from stable_baselines import DDPG

# Custom MLP policy of two layers of size 16 each
class CustomPolicy(FeedForwardPolicy):
    def __init__(self, *args, **kwargs):
        super(CustomPolicy, self).__init__(*args, **kwargs,
                                           layers=[16, 16],
                                           layer_norm=False,
                                           feature_extraction="mlp")

# Create and wrap the environment
env = gym.make('Pendulum-v0')
env = DummyVecEnv([lambda: env])

model = DDPG(CustomPolicy, env, verbose=1)
# Train the agent
model.learn(total_timesteps=100000)
```

1.19 DQN

Deep Q Network (DQN) and its extensions (Double-DQN, Dueling-DQN, Prioritized Experience Replay).

Warning: The DQN model does not support `stable_baselines.common.policies`, as a result it must use its own policy models (see *DQN Policies*).

Available Policies

MlpPolicy	Policy object that implements DQN policy, using a MLP (2 layers of 64)
LnMlpPolicy	Policy object that implements DQN policy, using a MLP (2 layers of 64), with layer normalisation
CnnPolicy	Policy object that implements DQN policy, using a CNN (the nature CNN)
LnCnnPolicy	Policy object that implements DQN policy, using a CNN (the nature CNN), with layer normalisation

1.19.1 Notes

- Original paper: https://arxiv.org/abs/1312.5602

1.19.2 Can I use?

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

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

1.19.3 Example

```
import gym

from stable_baselines.common.vec_env import DummyVecEnv
from stable_baselines.deepq.policies import MlpPolicy
from stable_baselines import DQN

env = gym.make('CartPole-v1')
env = DummyVecEnv([lambda: env])

model = DQN(MlpPolicy, env, verbose=1)
model.learn(total_timesteps=25000)
model.save("deepq_cartpole")

del model # remove to demonstrate saving and loading
```

(continues on next page)

(continued from previous page)

```
model = DQN.load("deepq_cartpole")

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

With Atari:

```
from stable_baselines.common.atari_wrappers import make_atari
from stable_baselines.deepq.policies import MlpPolicy, CnnPolicy
from stable_baselines import DQN

env = make_atari('BreakoutNoFrameskip-v4')

model = DQN(CnnPolicy, env, verbose=1)
model.learn(total_timesteps=25000)
model.save("deepq_breakout")

del model # remove to demonstrate saving and loading

model = DQN.load("deepq_breakout")

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

1.19.4 Parameters

class stable_baselines.deepq.**DQN**(*policy*, *env*, *gamma=0.99*, *learning_rate=0.0005*, *buffer_size=50000*, *exploration_fraction=0.1*, *exploration_final_eps=0.02*, *train_freq=1*, *batch_size=32*, *checkpoint_freq=10000*, *checkpoint_path=None*, *learning_starts=1000*, *target_network_update_freq=500*, *prioritized_replay=False*, *prioritized_replay_alpha=0.6*, *prioritized_replay_beta0=0.4*, *prioritized_replay_beta_iters=None*, *prioritized_replay_eps=1e-06*, *param_noise=False*, *verbose=0*, *tensorboard_log=None*, *_init_setup_model=True*, *policy_kwargs=None*, *full_tensorboard_log=False*)

The DQN model class. DQN paper: https://arxiv.org/pdf/1312.5602.pdf

Parameters

- **policy** – (DQNPolicy or str) The policy model to use (MlpPolicy, CnnPolicy, LnMlpPolicy, ...)
- **env** – (Gym environment or str) The environment to learn from (if registered in Gym, can be str)
- **gamma** – (float) discount factor
- **learning_rate** – (float) learning rate for adam optimizer

- **buffer_size** – (int) size of the replay buffer
- **exploration_fraction** – (float) fraction of entire training period over which the exploration rate is annealed
- **exploration_final_eps** – (float) final value of random action probability
- **train_freq** – (int) update the model every *train_freq* steps. set to None to disable printing
- **batch_size** – (int) size of a batched sampled from replay buffer for training
- **checkpoint_freq** – (int) how often to save the model. This is so that the best version is restored at the end of the training. If you do not wish to restore the best version at the end of the training set this variable to None.
- **checkpoint_path** – (str) replacement path used if you need to log to somewhere else than a temporary directory.
- **learning_starts** – (int) how many steps of the model to collect transitions for before learning starts
- **target_network_update_freq** – (int) update the target network every *target_network_update_freq* steps.
- **prioritized_replay** – (bool) if True prioritized replay buffer will be used.
- **prioritized_replay_alpha** – (float)alpha parameter for prioritized replay buffer. It determines how much prioritization is used, with alpha=0 corresponding to the uniform case.
- **prioritized_replay_beta0** – (float) initial value of beta for prioritized replay buffer
- **prioritized_replay_beta_iters** – (int) number of iterations over which beta will be annealed from initial value to 1.0. If set to None equals to max_timesteps.
- **prioritized_replay_eps** – (float) epsilon to add to the TD errors when updating priorities.
- **param_noise** – (bool) Whether or not to apply noise to the parameters of the policy.
- **verbose** – (int) the verbosity level: 0 none, 1 training information, 2 tensorflow debug
- **tensorboard_log** – (str) the log location for tensorboard (if None, no logging)
- **_init_setup_model** – (bool) Whether or not to build the network at the creation of the instance
- **full_tensorboard_log** – (bool) enable additional logging when using tensorboard WARNING: this logging can take a lot of space quickly

action_probability(*observation*, *state=None*, *mask=None*, *actions=None*, *logp=False*)

If `actions` is `None`, then get the model's action probability distribution from a given observation.

Depending on the action space the output is:

- Discrete: probability for each possible action
- Box: mean and standard deviation of the action output

However if `actions` is not `None`, this function will return the probability that the given actions are taken with the given parameters (observation, state, ...) on this model. For discrete action spaces, it returns the probability mass; for continuous action spaces, the probability density. This is since the probability mass will always be zero in continuous spaces, see http://blog.christianperone.com/2019/01/ for a good explanation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (np.ndarray) The last states (can be None, used in recurrent policies)
- **mask** – (np.ndarray) The last masks (can be None, used in recurrent policies)
- **actions** – (np.ndarray) (OPTIONAL) For calculating the likelihood that the given actions are chosen by the model for each of the given parameters. Must have the same number of actions and observations. (set to None to return the complete action probability distribution)
- **logp** – (bool) (OPTIONAL) When specified with actions, returns probability in log-space. This has no effect if actions is None.

Returns (np.ndarray) the model's (log) action probability

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

Returns (Gym Environment) The current environment

get_parameter_list()
Get tensorflow Variables of model's parameters

This includes all variables necessary for continuing training (saving / loading).

Returns (list) List of tensorflow Variables

get_parameters()
Get current model parameters as dictionary of variable name -> ndarray.

Returns (OrderedDict) Dictionary of variable name -> ndarray of model's parameters.

learn(*total_timesteps*, *callback=None*, *seed=None*, *log_interval=100*, *tb_log_name='DQN'*, *reset_num_timesteps=True*, *replay_wrapper=None*)
Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples to train on
- **seed** – (int) The initial seed for training, if None: keep current seed
- **callback** – (function (dict, dict)) -> boolean function called at every steps with state of the algorithm. It takes the local and global variables. If it returns False, training is aborted.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for tensorboard log
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseRLModel) the trained model

classmethod load(*load_path*, *env=None*, ***kwargs*)
Load the model from file

Parameters

- **load_path** – (str or file-like) the saved parameter location
- **env** – (Gym Envrionment) the new environment to run the loaded model on (can be None if you only need prediction from a trained model)
- **kwargs** – extra arguments to change the model when loading

load_parameters(*load_path_or_dict*, *exact_match=True*)
Load model parameters from a file or a dictionary

Dictionary keys should be tensorflow variable names, which can be obtained with `get_parameters` function. If `exact_match` is True, dictionary should contain keys for all model's parameters, otherwise RunTimeError is raised. If False, only variables included in the dictionary will be updated.

This does not load agent's hyper-parameters.

Warning: This function does not update trainer/optimizer variables (e.g. momentum). As such training after using this function may lead to less-than-optimal results.

Parameters

- **load_path_or_dict** – (str or file-like or dict) Save parameter location or dict of parameters as variable.name -> ndarrays to be loaded.
- **exact_match** – (bool) If True, expects load dictionary to contain keys for all variables in the model. If False, loads parameters only for variables mentioned in the dictionary. Defaults to True.

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

Parameters

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

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

pretrain(*dataset*, *n_epochs=10*, *learning_rate=0.0001*, *adam_epsilon=1e-08*, *val_interval=None*)
Pretrain a model using behavior cloning: supervised learning given an expert dataset.

NOTE: only Box and Discrete spaces are supported for now.

Parameters

- **dataset** – (ExpertDataset) Dataset manager
- **n_epochs** – (int) Number of iterations on the training set
- **learning_rate** – (float) Learning rate
- **adam_epsilon** – (float) the epsilon value for the adam optimizer
- **val_interval** – (int) Report training and validation losses every n epochs. By default, every 10th of the maximum number of epochs.

Returns (BaseRLModel) the pretrained model

save(*save_path*)
Save the current parameters to file

Parameters **save_path** – (str or file-like object) the save location

set_env(*env*)
Checks the validity of the environment, and if it is coherent, set it as the current environment.

Parameters **env** – (Gym Environment) The environment for learning a policy

setup_model()
Create all the functions and tensorflow graphs necessary to train the model

1.19.5 DQN Policies

class stable_baselines.deepq.**MlpPolicy**(*sess*, *ob_space*, *ac_space*, *n_env*, *n_steps*, *n_batch*, *reuse=False*, *obs_phs=None*, *dueling=True*, ***_kwargs*)
Policy object that implements DQN policy, using a MLP (2 layers of 64)

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **reuse** – (bool) If the policy is reusable or not
- **obs_phs** – (TensorFlow Tensor, TensorFlow Tensor) a tuple containing an override for observation placeholder and the processed observation placeholder respectivly
- **dueling** – (bool) if true double the output MLP to compute a baseline for action scores
- **_kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

action_ph
tf.Tensor: placeholder for actions, shape (self.n_batch,) + self.ac_space.shape.

initial_state
The initial state of the policy. For feedforward policies, None. For a recurrent policy, a NumPy array of shape (self.n_env,) + state_shape.

is_discrete
bool: is action space discrete.

obs_ph
tf.Tensor: placeholder for observations, shape (self.n_batch,) + self.ob_space.shape.

proba_step(*obs*, *state=None*, *mask=None*)
Returns the action probability for a single step

Parameters

- **obs** – (np.ndarray float or int) The current observation of the environment
- **state** – (np.ndarray float) The last states (used in recurrent policies)
- **mask** – (np.ndarray float) The last masks (used in recurrent policies)

Returns (np.ndarray float) the action probability

processed_obs
tf.Tensor: processed observations, shape (self.n_batch,) + self.ob_space.shape.

The form of processing depends on the type of the observation space, and the parameters whether scale is passed to the constructor; see observation_input for more information.

step(*obs*, *state=None*, *mask=None*, *deterministic=True*)
Returns the q_values for a single step

Parameters

- **obs** – (np.ndarray float or int) The current observation of the environment
- **state** – (np.ndarray float) The last states (used in recurrent policies)
- **mask** – (np.ndarray float) The last masks (used in recurrent policies)
- **deterministic** – (bool) Whether or not to return deterministic actions.

Returns (np.ndarray int, np.ndarray float, np.ndarray float) actions, q_values, states

class stable_baselines.deepq.**LnMlpPolicy**(*sess*, *ob_space*, *ac_space*, *n_env*, *n_steps*, *n_batch*, *reuse=False*, *obs_phs=None*, *dueling=True*, ***_kwargs*)
Policy object that implements DQN policy, using a MLP (2 layers of 64), with layer normalisation

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **reuse** – (bool) If the policy is reusable or not
- **obs_phs** – (TensorFlow Tensor, TensorFlow Tensor) a tuple containing an override for observation placeholder and the processed observation placeholder respectivly
- **dueling** – (bool) if true double the output MLP to compute a baseline for action scores
- **_kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

action_ph
tf.Tensor: placeholder for actions, shape (self.n_batch,) + self.ac_space.shape.

initial_state
The initial state of the policy. For feedforward policies, None. For a recurrent policy, a NumPy array of shape (self.n_env,) + state_shape.

is_discrete
bool: is action space discrete.

obs_ph
tf.Tensor: placeholder for observations, shape (self.n_batch,) + self.ob_space.shape.

proba_step(*obs*, *state=None*, *mask=None*)
Returns the action probability for a single step

Parameters

- **obs** – (np.ndarray float or int) The current observation of the environment

- **state** – (np.ndarray float) The last states (used in recurrent policies)
- **mask** – (np.ndarray float) The last masks (used in recurrent policies)

Returns (np.ndarray float) the action probability

processed_obs
tf.Tensor: processed observations, shape (self.n_batch,) + self.ob_space.shape.

The form of processing depends on the type of the observation space, and the parameters whether scale is passed to the constructor; see observation_input for more information.

step(*obs*, *state=None*, *mask=None*, *deterministic=True*)
Returns the q_values for a single step

Parameters

- **obs** – (np.ndarray float or int) The current observation of the environment
- **state** – (np.ndarray float) The last states (used in recurrent policies)
- **mask** – (np.ndarray float) The last masks (used in recurrent policies)
- **deterministic** – (bool) Whether or not to return deterministic actions.

Returns (np.ndarray int, np.ndarray float, np.ndarray float) actions, q_values, states

class stable_baselines.deepq.**CnnPolicy**(*sess*, *ob_space*, *ac_space*, *n_env*, *n_steps*, *n_batch*, *reuse=False*, *obs_phs=None*, *dueling=True*, ***_kwargs*)
Policy object that implements DQN policy, using a CNN (the nature CNN)

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **reuse** – (bool) If the policy is reusable or not
- **obs_phs** – (TensorFlow Tensor, TensorFlow Tensor) a tuple containing an override for observation placeholder and the processed observation placeholder respectivly
- **dueling** – (bool) if true double the output MLP to compute a baseline for action scores
- **_kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

action_ph
tf.Tensor: placeholder for actions, shape (self.n_batch,) + self.ac_space.shape.

initial_state
The initial state of the policy. For feedforward policies, None. For a recurrent policy, a NumPy array of shape (self.n_env,) + state_shape.

is_discrete
bool: is action space discrete.

obs_ph
tf.Tensor: placeholder for observations, shape (self.n_batch,) + self.ob_space.shape.

proba_step(*obs*, *state=None*, *mask=None*)
Returns the action probability for a single step

Parameters

- **obs** – (np.ndarray float or int) The current observation of the environment
- **state** – (np.ndarray float) The last states (used in recurrent policies)
- **mask** – (np.ndarray float) The last masks (used in recurrent policies)

Returns (np.ndarray float) the action probability

processed_obs
tf.Tensor: processed observations, shape (self.n_batch,) + self.ob_space.shape.

The form of processing depends on the type of the observation space, and the parameters whether scale is passed to the constructor; see observation_input for more information.

step(*obs*, *state=None*, *mask=None*, *deterministic=True*)
Returns the q_values for a single step

Parameters

- **obs** – (np.ndarray float or int) The current observation of the environment
- **state** – (np.ndarray float) The last states (used in recurrent policies)
- **mask** – (np.ndarray float) The last masks (used in recurrent policies)
- **deterministic** – (bool) Whether or not to return deterministic actions.

Returns (np.ndarray int, np.ndarray float, np.ndarray float) actions, q_values, states

class stable_baselines.deepq.**LnCnnPolicy**(*sess*, *ob_space*, *ac_space*, *n_env*, *n_steps*, *n_batch*, *reuse=False*, *obs_phs=None*, *dueling=True*, ***_kwargs*)
Policy object that implements DQN policy, using a CNN (the nature CNN), with layer normalisation

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **reuse** – (bool) If the policy is reusable or not
- **obs_phs** – (TensorFlow Tensor, TensorFlow Tensor) a tuple containing an override for observation placeholder and the processed observation placeholder respectivly
- **dueling** – (bool) if true double the output MLP to compute a baseline for action scores
- **_kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

action_ph
tf.Tensor: placeholder for actions, shape (self.n_batch,) + self.ac_space.shape.

initial_state
The initial state of the policy. For feedforward policies, None. For a recurrent policy, a NumPy array of shape (self.n_env,) + state_shape.

is_discrete
bool: is action space discrete.

obs_ph
tf.Tensor: placeholder for observations, shape (self.n_batch,) + self.ob_space.shape.

proba_step(*obs*, *state=None*, *mask=None*)
Returns the action probability for a single step

Parameters

- **obs** – (np.ndarray float or int) The current observation of the environment
- **state** – (np.ndarray float) The last states (used in recurrent policies)
- **mask** – (np.ndarray float) The last masks (used in recurrent policies)

Returns (np.ndarray float) the action probability

processed_obs
tf.Tensor: processed observations, shape (self.n_batch,) + self.ob_space.shape.

The form of processing depends on the type of the observation space, and the parameters whether scale is passed to the constructor; see observation_input for more information.

step(*obs*, *state=None*, *mask=None*, *deterministic=True*)
Returns the q_values for a single step

Parameters

- **obs** – (np.ndarray float or int) The current observation of the environment
- **state** – (np.ndarray float) The last states (used in recurrent policies)
- **mask** – (np.ndarray float) The last masks (used in recurrent policies)
- **deterministic** – (bool) Whether or not to return deterministic actions.

Returns (np.ndarray int, np.ndarray float, np.ndarray float) actions, q_values, states

1.19.6 Custom Policy Network

Similarly to the example given in the examples page. You can easily define a custom architecture for the policy network:

```
import gym

from stable_baselines.deepq.policies import FeedForwardPolicy
from stable_baselines.common.vec_env import DummyVecEnv
from stable_baselines import DQN

# Custom MLP policy of two layers of size 32 each
class CustomPolicy(FeedForwardPolicy):
    def __init__(self, *args, **kwargs):
        super(CustomPolicy, self).__init__(*args, **kwargs,
                                           layers=[32, 32],
                                           layer_norm=False,
                                           feature_extraction="mlp")

# Create and wrap the environment
env = gym.make('LunarLander-v2')
env = DummyVecEnv([lambda: env])
```

(continues on next page)

(continued from previous page)

```
model = DQN(CustomPolicy, env, verbose=1)
# Train the agent
model.learn(total_timesteps=100000)
```

1.20 GAIL

The Generative Adversarial Imitation Learning (GAIL) uses expert trajectories to recover a cost function and then learn a policy.

Learning a cost function from expert demonstrations is called Inverse Reinforcement Learning (IRL). The connection between GAIL and Generative Adversarial Networks (GANs) is that it uses a discriminator that tries to seperate expert trajectory from trajectories of the learned policy, which has the role of the generator here.

1.20.1 Notes

- Original paper: https://arxiv.org/abs/1606.03476

Warning: Images are not yet handled properly by the current implementation

1.20.2 If you want to train an imitation learning agent

Step 1: Generate expert data

You can either train a RL algorithm in a classic setting, use another controller (e.g. a PID controller) or human demonstrations.

We recommend you to take a look at *pre-training* section or directly look at `stable_baselines/gail/dataset/` folder to learn more about the expected format for the dataset.

Here is an example of training a Soft Actor-Critic model to generate expert trajectories for GAIL:

```
from stable_baselines import SAC
from stable_baselines.gail import generate_expert_traj

# Generate expert trajectories (train expert)
model = SAC('MlpPolicy', 'Pendulum-v0', verbose=1)
# Train for 60000 timesteps and record 10 trajectories
# all the data will be saved in 'expert_pendulum.npz' file
generate_expert_traj(model, 'expert_pendulum', n_timesteps=60000, n_episodes=10)
```

Step 2: Run GAIL

In case you want to run Behavior Cloning (BC)

Use the `.pretrain()` method (cf guide).

Others

Thanks to the open source:

- @openai/imitation
- @carpedm20/deep-rl-tensorflow

1.20.3 Can I use?

- Recurrent policies:
- Multi processing: ✓ (using MPI)
- Gym spaces:

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

1.20.4 Example

```
import gym

from stable_baselines import GAIL, SAC
from stable_baselines.gail import ExpertDataset, generate_expert_traj

# Generate expert trajectories (train expert)
model = SAC('MlpPolicy', 'Pendulum-v0', verbose=1)
generate_expert_traj(model, 'expert_pendulum', n_timesteps=100, n_episodes=10)

# Load the expert dataset
dataset = ExpertDataset(expert_path='expert_pendulum.npz', traj_limitation=10,␣
↪verbose=1)

model = GAIL("MlpPolicy", 'Pendulum-v0', dataset, verbose=1)
# Note: in practice, you need to train for 1M steps to have a working policy
model.learn(total_timesteps=1000)
model.save("gail_pendulum")

del model # remove to demonstrate saving and loading

model = GAIL.load("gail_pendulum")

env = gym.make('Pendulum-v0')
obs = env.reset()
while True:
  action, _states = model.predict(obs)
  obs, rewards, dones, info = env.step(action)
  env.render()
```

1.20.5 Parameters

class stable_baselines.gail.**GAIL**(*policy*, *env*, *expert_dataset=None*, *hidden_size_adversary=100*, *adversary_entcoeff=0.001*, *g_step=3*, *d_step=1*, *d_stepsize=0.0003*, *verbose=0*, *_init_setup_model=True*, ***kwargs*)

Generative Adversarial Imitation Learning (GAIL)

Warning: Images are not yet handled properly by the current implementation

Parameters

- **policy** – (ActorCriticPolicy or str) The policy model to use (MlpPolicy, CnnPolicy, CnnLstmPolicy, ...)
- **env** – (Gym environment or str) The environment to learn from (if registered in Gym, can be str)
- **expert_dataset** – (ExpertDataset) the dataset manager
- **gamma** – (float) the discount value
- **timesteps_per_batch** – (int) the number of timesteps to run per batch (horizon)
- **max_kl** – (float) the Kullback-Leibler loss threshold
- **cg_iters** – (int) the number of iterations for the conjugate gradient calculation
- **lam** – (float) GAE factor
- **entcoeff** – (float) the weight for the entropy loss
- **cg_damping** – (float) the compute gradient dampening factor
- **vf_stepsize** – (float) the value function stepsize
- **vf_iters** – (int) the value function's number iterations for learning
- **hidden_size** – ([int]) the hidden dimension for the MLP
- **g_step** – (int) number of steps to train policy in each epoch
- **d_step** – (int) number of steps to train discriminator in each epoch
- **d_stepsize** – (float) the reward giver stepsize
- **verbose** – (int) the verbosity level: 0 none, 1 training information, 2 tensorflow debug
- **_init_setup_model** – (bool) Whether or not to build the network at the creation of the instance
- **full_tensorboard_log** – (bool) enable additional logging when using tensorboard WARNING: this logging can take a lot of space quickly

action_probability(*observation*, *state=None*, *mask=None*, *actions=None*, *logp=False*)

If `actions` is `None`, then get the model's action probability distribution from a given observation.

Depending on the action space the output is:

- Discrete: probability for each possible action
- Box: mean and standard deviation of the action output

However if `actions` is not `None`, this function will return the probability that the given actions are taken with the given parameters (observation, state, ...) on this model. For discrete action spaces, it returns the probability mass; for continuous action spaces, the probability density. This is since the probability mass will always be zero in continuous spaces, see http://blog.christianperone.com/2019/01/ for a good explanation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (np.ndarray) The last states (can be None, used in recurrent policies)
- **mask** – (np.ndarray) The last masks (can be None, used in recurrent policies)
- **actions** – (np.ndarray) (OPTIONAL) For calculating the likelihood that the given actions are chosen by the model for each of the given parameters. Must have the same number of actions and observations. (set to None to return the complete action probability distribution)
- **logp** – (bool) (OPTIONAL) When specified with actions, returns probability in log-space. This has no effect if actions is None.

Returns (np.ndarray) the model's (log) action probability

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

Returns (Gym Environment) The current environment

get_parameter_list()
Get tensorflow Variables of model's parameters

This includes all variables necessary for continuing training (saving / loading).

Returns (list) List of tensorflow Variables

get_parameters()
Get current model parameters as dictionary of variable name -> ndarray.

Returns (OrderedDict) Dictionary of variable name -> ndarray of model's parameters.

learn(*total_timesteps*, *callback=None*, *seed=None*, *log_interval=100*, *tb_log_name='GAIL'*, *reset_num_timesteps=True*)
Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples to train on
- **seed** – (int) The initial seed for training, if None: keep current seed
- **callback** – (function (dict, dict)) -> boolean function called at every steps with state of the algorithm. It takes the local and global variables. If it returns False, training is aborted.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for tensorboard log
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseRLModel) the trained model

classmethod load(*load_path*, *env=None*, ***kwargs*)
Load the model from file

Parameters

- **load_path** – (str or file-like) the saved parameter location
- **env** – (Gym Envrionment) the new environment to run the loaded model on (can be None if you only need prediction from a trained model)
- **kwargs** – extra arguments to change the model when loading

load_parameters(*load_path_or_dict*, *exact_match=True*)
Load model parameters from a file or a dictionary

Dictionary keys should be tensorflow variable names, which can be obtained with `get_parameters` function. If `exact_match` is True, dictionary should contain keys for all model's parameters, otherwise RunTimeError is raised. If False, only variables included in the dictionary will be updated.

This does not load agent's hyper-parameters.

Warning: This function does not update trainer/optimizer variables (e.g. momentum). As such training after using this function may lead to less-than-optimal results.

Parameters

- **load_path_or_dict** – (str or file-like or dict) Save parameter location or dict of parameters as variable.name -> ndarrays to be loaded.
- **exact_match** – (bool) If True, expects load dictionary to contain keys for all variables in the model. If False, loads parameters only for variables mentioned in the dictionary. Defaults to True.

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

Parameters

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

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

pretrain(*dataset*, *n_epochs=10*, *learning_rate=0.0001*, *adam_epsilon=1e-08*, *val_interval=None*)
Pretrain a model using behavior cloning: supervised learning given an expert dataset.

NOTE: only Box and Discrete spaces are supported for now.

Parameters

- **dataset** – (ExpertDataset) Dataset manager
- **n_epochs** – (int) Number of iterations on the training set
- **learning_rate** – (float) Learning rate
- **adam_epsilon** – (float) the epsilon value for the adam optimizer
- **val_interval** – (int) Report training and validation losses every n epochs. By default, every 10th of the maximum number of epochs.

Returns (BaseRLModel) the pretrained model

save(*save_path*)
Save the current parameters to file

Parameters **save_path** – (str or file-like object) the save location

set_env(*env*)
Checks the validity of the environment, and if it is coherent, set it as the current environment.

Parameters **env** – (Gym Environment) The environment for learning a policy

setup_model()
Create all the functions and tensorflow graphs necessary to train the model

1.21 HER

Hindsight Experience Replay (HER)

HER is a method wrapper that works with Off policy methods (DQN, SAC, TD3 and DDPG for example).

Note: HER was re-implemented from scratch in Stable-Baselines compared to the original OpenAI baselines. If you want to reproduce results from the paper, please use the rl baselines zoo in order to have the correct hyperparameters and at least 8 MPI workers with DDPG.

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

Warning: you must pass an environment or wrap it with `HERGoalEnvWrapper` in order to use the predict method

1.21.1 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/

1.21.2 Can I use?

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

1.21.3 Example

```
from stable_baselines import HER, DQN, SAC, DDPG, TD3
from stable_baselines.her import GoalSelectionStrategy, HERGoalEnvWrapper
from stable_baselines.common.bit_flipping_env import BitFlippingEnv

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

(continues on next page)

(continued from previous page)

```
env = BitFlippingEnv(N_BITS, continuous=model_class in [DDPG, SAC, TD3], max_steps=N_
↪BITS)

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

# Wrap the model
model = HER('MlpPolicy', env, model_class, n_sampled_goal=4, goal_selection_
↪strategy=goal_selection_strategy,
                                                    verbose=1)
# Train the model
model.learn(1000)

model.save("./her_bit_env")

# WARNING: you must pass an env
# or wrap your environment with HERGoalEnvWrapper to use the predict method
model = HER.load('./her_bit_env', env=env)

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

    if done:
        obs = env.reset()
```

1.21.4 Parameters

class stable_baselines.her.**HER**(*policy*, *env*, *model_class*, *n_sampled_goal=4*, *goal_selection_strategy='future'*, **args*, ***kwargs*)

Hindsight Experience Replay (HER) https://arxiv.org/abs/1707.01495

Parameters

- **policy** – (BasePolicy or str) The policy model to use (MlpPolicy, CnnPolicy, CnnLstmPolicy, ...)
- **env** – (Gym environment or str) The environment to learn from (if registered in Gym, can be str)
- **model_class** – (OffPolicyRLModel) The off policy RL model to apply Hindsight Experience Replay currently supported: DQN, DDPG, SAC
- **n_sampled_goal** – (int)
- **goal_selection_strategy** – (GoalSelectionStrategy or str)

action_probability(*observation*, *state=None*, *mask=None*, *actions=None*, *logp=False*)

If `actions` is `None`, then get the model's action probability distribution from a given observation.

Depending on the action space the output is:

- Discrete: probability for each possible action
- Box: mean and standard deviation of the action output

However if `actions` is not `None`, this function will return the probability that the given actions are taken with the given parameters (observation, state, ...) on this model. For discrete action spaces, it returns

the probability mass; for continuous action spaces, the probability density. This is since the probability mass will always be zero in continuous spaces, see http://blog.christianperone.com/2019/01/ for a good explanation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (np.ndarray) The last states (can be None, used in recurrent policies)
- **mask** – (np.ndarray) The last masks (can be None, used in recurrent policies)
- **actions** – (np.ndarray) (OPTIONAL) For calculating the likelihood that the given actions are chosen by the model for each of the given parameters. Must have the same number of actions and observations. (set to None to return the complete action probability distribution)
- **logp** – (bool) (OPTIONAL) When specified with actions, returns probability in log-space. This has no effect if actions is None.

Returns (np.ndarray) the model's (log) action probability

get_env()

returns the current environment (can be None if not defined)

Returns (Gym Environment) The current environment

get_parameter_list()

Get tensorflow Variables of model's parameters

This includes all variables necessary for continuing training (saving / loading).

Returns (list) List of tensorflow Variables

learn(*total_timesteps*, *callback=None*, *seed=None*, *log_interval=100*, *tb_log_name='HER'*, *reset_num_timesteps=True*)

Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples to train on
- **seed** – (int) The initial seed for training, if None: keep current seed
- **callback** – (function (dict, dict)) -> boolean function called at every steps with state of the algorithm. It takes the local and global variables. If it returns False, training is aborted.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for tensorboard log
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseRLModel) the trained model

classmethod load(*load_path*, *env=None*, ***kwargs*)

Load the model from file

Parameters

- **load_path** – (str or file-like) the saved parameter location
- **env** – (Gym Envrionment) the new environment to run the loaded model on (can be None if you only need prediction from a trained model)
- **kwargs** – extra arguments to change the model when loading

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

Parameters

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

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

save(*save_path*)
Save the current parameters to file

Parameters **save_path** – (str or file-like object) the save location

set_env(*env*)
Checks the validity of the environment, and if it is coherent, set it as the current environment.

Parameters **env** – (Gym Environment) The environment for learning a policy

setup_model()
Create all the functions and tensorflow graphs necessary to train the model

1.21.5 Goal Selection Strategies

class stable_baselines.her.**GoalSelectionStrategy**
The strategies for selecting new goals when creating artificial transitions.

1.21.6 Gaol Env Wrapper

class stable_baselines.her.**HERGoalEnvWrapper**(*env*)
A wrapper that allow to use dict observation space (coming from GoalEnv) with the RL algorithms. It assumes that all the spaces of the dict space are of the same type.

Parameters **env** – (gym.GoalEnv)

convert_dict_to_obs(*obs_dict*)

Parameters **obs_dict** – (dict<np.ndarray>)

Returns (np.ndarray)

convert_obs_to_dict(*observations*)
Inverse operation of convert_dict_to_obs

Parameters **observations** – (np.ndarray)

Returns (OrderedDict<np.ndarray>)

1.21.7 Replay Wrapper

class stable_baselines.her.**HindsightExperienceReplayWrapper**(*replay_buffer*, *n_sampled_goal*, *goal_selection_strategy*, *wrapped_env*)

Wrapper around a replay buffer in order to use HER. This implementation is inspired by to the one found in https://github.com/NervanaSystems/coach/.

Parameters

- **replay_buffer** – (ReplayBuffer)
- **n_sampled_goal** – (int) The number of artificial transitions to generate for each actual transition
- **goal_selection_strategy** – (GoalSelectionStrategy) The method that will be used to generate the goals for the artificial transitions.
- **wrapped_env** – (HERGoalEnvWrapper) the GoalEnv wrapped using HERGoalEnvWrapper, that enables to convert observation to dict, and vice versa

add(*obs_t*, *action*, *reward*, *obs_tp1*, *done*)

add a new transition to the buffer

Parameters

- **obs_t** – (np.ndarray) the last observation
- **action** – ([float]) the action
- **reward** – (float) the reward of the transition
- **obs_tp1** – (np.ndarray) the new observation
- **done** – (bool) is the episode done

can_sample(*n_samples*)

Check if n_samples samples can be sampled from the buffer.

Parameters **n_samples** – (int)

Returns (bool)

1.22 PPO1

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

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

Note: PPO2 is the implementation of OpenAI made for GPU. For multiprocessing, it uses vectorized environments compared to PPO1 which uses MPI.

1.22.1 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/
- `mpirun -np 8 python -m stable_baselines.ppo1.run_atari` runs the algorithm for 40M frames = 10M timesteps on an Atari game. See help (`-h`) for more options.
- `python -m stable_baselines.ppo1.run_mujoco` runs the algorithm for 1M frames on a Mujoco environment.
- Train mujoco 3d humanoid (with optimal-ish hyperparameters): `mpirun -np 16 python -m stable_baselines.ppo1.run_humanoid --model-path=/path/to/model`
- Render the 3d humanoid: `python -m stable_baselines.ppo1.run_humanoid --play --model-path=/path/to/model`

1.22.2 Can I use?

- Recurrent policies:
- Multi processing: ✓ (using MPI)
- Gym spaces:

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

1.22.3 Example

```
import gym

from stable_baselines.common.policies import MlpPolicy
from stable_baselines.common.vec_env import DummyVecEnv
from stable_baselines import PPO1

env = gym.make('CartPole-v1')
env = DummyVecEnv([lambda: env])

model = PPO1(MlpPolicy, env, verbose=1)
model.learn(total_timesteps=25000)
model.save("ppo1_cartpole")

del model # remove to demonstrate saving and loading

model = PPO1.load("ppo1_cartpole")

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

1.22.4 Parameters

class stable_baselines.ppo1.**PPO1**(*policy*, *env*, *gamma=0.99*, *timesteps_per_actorbatch=256*, *clip_param=0.2*, *entcoeff=0.01*, *optim_epochs=4*, *optim_stepsize=0.001*, *optim_batchsize=64*, *lam=0.95*, *adam_epsilon=1e-05*, *schedule='linear'*, *verbose=0*, *tensorboard_log=None*, *_init_setup_model=True*, *policy_kwargs=None*, *full_tensorboard_log=False*)

Proximal Policy Optimization algorithm (MPI version). Paper: https://arxiv.org/abs/1707.06347

Parameters

- **env** – (Gym environment or str) The environment to learn from (if registered in Gym, can be str)
- **policy** – (ActorCriticPolicy or str) The policy model to use (MlpPolicy, CnnPolicy, CnnLstmPolicy, ...)
- **timesteps_per_actorbatch** – (int) timesteps per actor per update
- **clip_param** – (float) clipping parameter epsilon
- **entcoeff** – (float) the entropy loss weight
- **optim_epochs** – (float) the optimizer's number of epochs
- **optim_stepsize** – (float) the optimizer's stepsize
- **optim_batchsize** – (int) the optimizer's the batch size
- **gamma** – (float) discount factor
- **lam** – (float) advantage estimation
- **adam_epsilon** – (float) the epsilon value for the adam optimizer
- **schedule** – (str) The type of scheduler for the learning rate update ('linear', 'constant', 'double_linear_con', 'middle_drop' or 'double_middle_drop')
- **verbose** – (int) the verbosity level: 0 none, 1 training information, 2 tensorflow debug
- **tensorboard_log** – (str) the log location for tensorboard (if None, no logging)
- **_init_setup_model** – (bool) Whether or not to build the network at the creation of the instance
- **policy_kwargs** – (dict) additional arguments to be passed to the policy on creation
- **full_tensorboard_log** – (bool) enable additional logging when using tensorboard WARNING: this logging can take a lot of space quickly

action_probability(*observation*, *state=None*, *mask=None*, *actions=None*, *logp=False*)

If `actions` is `None`, then get the model's action probability distribution from a given observation.

Depending on the action space the output is:

- Discrete: probability for each possible action
- Box: mean and standard deviation of the action output

However if `actions` is not `None`, this function will return the probability that the given actions are taken with the given parameters (observation, state, ...) on this model. For discrete action spaces, it returns the probability mass; for continuous action spaces, the probability density. This is since the probability mass will always be zero in continuous spaces, see http://blog.christianperone.com/2019/01/ for a good explanation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (np.ndarray) The last states (can be None, used in recurrent policies)
- **mask** – (np.ndarray) The last masks (can be None, used in recurrent policies)
- **actions** – (np.ndarray) (OPTIONAL) For calculating the likelihood that the given actions are chosen by the model for each of the given parameters. Must have the same number of actions and observations. (set to None to return the complete action probability distribution)
- **logp** – (bool) (OPTIONAL) When specified with actions, returns probability in log-space. This has no effect if actions is None.

Returns (np.ndarray) the model's (log) action probability

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

Returns (Gym Environment) The current environment

get_parameter_list()
Get tensorflow Variables of model's parameters

This includes all variables necessary for continuing training (saving / loading).

Returns (list) List of tensorflow Variables

get_parameters()
Get current model parameters as dictionary of variable name -> ndarray.

Returns (OrderedDict) Dictionary of variable name -> ndarray of model's parameters.

learn(*total_timesteps*, *callback=None*, *seed=None*, *log_interval=100*, *tb_log_name='PPO1'*, *reset_num_timesteps=True*)
Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples to train on
- **seed** – (int) The initial seed for training, if None: keep current seed
- **callback** – (function (dict, dict)) -> boolean function called at every steps with state of the algorithm. It takes the local and global variables. If it returns False, training is aborted.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for tensorboard log
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseRLModel) the trained model

classmethod load(*load_path*, *env=None*, ***kwargs*)
Load the model from file

Parameters

- **load_path** – (str or file-like) the saved parameter location
- **env** – (Gym Envrionment) the new environment to run the loaded model on (can be None if you only need prediction from a trained model)
- **kwargs** – extra arguments to change the model when loading

load_parameters(*load_path_or_dict*, *exact_match=True*)
Load model parameters from a file or a dictionary

Dictionary keys should be tensorflow variable names, which can be obtained with `get_parameters` function. If `exact_match` is True, dictionary should contain keys for all model's parameters, otherwise RunTimeError is raised. If False, only variables included in the dictionary will be updated.

This does not load agent's hyper-parameters.

Warning: This function does not update trainer/optimizer variables (e.g. momentum). As such training after using this function may lead to less-than-optimal results.

Parameters

- **load_path_or_dict** – (str or file-like or dict) Save parameter location or dict of parameters as variable.name -> ndarrays to be loaded.
- **exact_match** – (bool) If True, expects load dictionary to contain keys for all variables in the model. If False, loads parameters only for variables mentioned in the dictionary. Defaults to True.

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

Parameters

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

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

pretrain(*dataset*, *n_epochs=10*, *learning_rate=0.0001*, *adam_epsilon=1e-08*, *val_interval=None*)
Pretrain a model using behavior cloning: supervised learning given an expert dataset.

NOTE: only Box and Discrete spaces are supported for now.

Parameters

- **dataset** – (ExpertDataset) Dataset manager
- **n_epochs** – (int) Number of iterations on the training set
- **learning_rate** – (float) Learning rate
- **adam_epsilon** – (float) the epsilon value for the adam optimizer
- **val_interval** – (int) Report training and validation losses every n epochs. By default, every 10th of the maximum number of epochs.

Returns (BaseRLModel) the pretrained model

save(*save_path*)
Save the current parameters to file

Parameters **save_path** – (str or file-like object) the save location

set_env(*env*)
Checks the validity of the environment, and if it is coherent, set it as the current environment.

Parameters **env** – (Gym Environment) The environment for learning a policy

setup_model()
Create all the functions and tensorflow graphs necessary to train the model

1.23 PPO2

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

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

Note: PPO2 is the implementation of OpenAI made for GPU. For multiprocessing, it uses vectorized environments compared to PPO1 which uses MPI.

Note: PPO2 contains several modifications from the original algorithm not documented by OpenAI: value function is also clipped and advantages are normalized.

1.23.1 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/
- **python -m stable_baselines.ppo2.run_atari runs the algorithm for 40M** frames = 10M timesteps on an Atari game. See help (`-h`) for more options.
- **python -m stable_baselines.ppo2.run_mujoco runs the algorithm for 1M** frames on a Mujoco environment.

1.23.2 Can I use?

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

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

1.23.3 Example

Train a PPO agent on *CartPole-v1* using 4 processes.

```
import gym

from stable_baselines.common.policies import MlpPolicy
from stable_baselines.common.vec_env import SubprocVecEnv
from stable_baselines import PPO2

# multiprocess environment
n_cpu = 4
env = SubprocVecEnv([lambda: gym.make('CartPole-v1') for i in range(n_cpu)])

model = PPO2(MlpPolicy, env, verbose=1)
model.learn(total_timesteps=25000)
model.save("ppo2_cartpole")

del model # remove to demonstrate saving and loading

model = PPO2.load("ppo2_cartpole")

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

1.23.4 Parameters

class stable_baselines.ppo2.**PPO2**(*policy*, *env*, *gamma=0.99*, *n_steps=128*, *ent_coef=0.01*, *learning_rate=0.00025*, *vf_coef=0.5*, *max_grad_norm=0.5*, *lam=0.95*, *nminibatches=4*, *noptepochs=4*, *cliprange=0.2*, *cliprange_vf=None*, *verbose=0*, *tensorboard_log=None*, *_init_setup_model=True*, *policy_kwargs=None*, *full_tensorboard_log=False*)

Proximal Policy Optimization algorithm (GPU version). Paper: https://arxiv.org/abs/1707.06347

Parameters

- **policy** – (ActorCriticPolicy or str) The policy model to use (MlpPolicy, CnnPolicy, CnnLstmPolicy, ...)
- **env** – (Gym environment or str) The environment to learn from (if registered in Gym, can be str)
- **gamma** – (float) Discount factor
- **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)
- **ent_coef** – (float) Entropy coefficient for the loss calculation
- **learning_rate** – (float or callable) The learning rate, it can be a function
- **vf_coef** – (float) Value function coefficient for the loss calculation
- **max_grad_norm** – (float) The maximum value for the gradient clipping

- **lam** – (float) Factor for trade-off of bias vs variance for Generalized Advantage Estimator
- **nminibatches** – (int) Number of training minibatches per update. For recurrent policies, the number of environments run in parallel should be a multiple of nminibatches.
- **noptepochs** – (int) Number of epoch when optimizing the surrogate
- **cliprange** – (float or callable) Clipping parameter, it can be a function
- **cliprange_vf** – (float or callable) Clipping parameter for the value function, it can be a function. This is a parameter specific to the OpenAI implementation. If None is passed (default), then *cliprange* (that is used for the policy) will be used. IMPORTANT: this clipping depends on the reward scaling. To deactivate value function clipping (and recover the original PPO implementation), you have to pass a negative value (e.g. -1).
- **verbose** – (int) the verbosity level: 0 none, 1 training information, 2 tensorflow debug
- **tensorboard_log** – (str) the log location for tensorboard (if None, no logging)
- **_init_setup_model** – (bool) Whether or not to build the network at the creation of the instance
- **policy_kwargs** – (dict) additional arguments to be passed to the policy on creation
- **full_tensorboard_log** – (bool) enable additional logging when using tensorboard WARNING: this logging can take a lot of space quickly

action_probability(*observation*, *state=None*, *mask=None*, *actions=None*, *logp=False*)

If `actions` is `None`, then get the model's action probability distribution from a given observation.

Depending on the action space the output is:

- Discrete: probability for each possible action
- Box: mean and standard deviation of the action output

However if `actions` is not `None`, this function will return the probability that the given actions are taken with the given parameters (observation, state, ...) on this model. For discrete action spaces, it returns the probability mass; for continuous action spaces, the probability density. This is since the probability mass will always be zero in continuous spaces, see http://blog.christianperone.com/2019/01/ for a good explanation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (np.ndarray) The last states (can be None, used in recurrent policies)
- **mask** – (np.ndarray) The last masks (can be None, used in recurrent policies)
- **actions** – (np.ndarray) (OPTIONAL) For calculating the likelihood that the given actions are chosen by the model for each of the given parameters. Must have the same number of actions and observations. (set to None to return the complete action probability distribution)
- **logp** – (bool) (OPTIONAL) When specified with actions, returns probability in log-space. This has no effect if actions is None.

Returns (np.ndarray) the model's (log) action probability

get_env()

returns the current environment (can be None if not defined)

Returns (Gym Environment) The current environment

get_parameter_list()
Get tensorflow Variables of model's parameters

This includes all variables necessary for continuing training (saving / loading).

Returns (list) List of tensorflow Variables

get_parameters()
Get current model parameters as dictionary of variable name -> ndarray.

Returns (OrderedDict) Dictionary of variable name -> ndarray of model's parameters.

learn(*total_timesteps*, *callback=None*, *seed=None*, *log_interval=1*, *tb_log_name='PPO2'*, *reset_num_timesteps=True*)
Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples to train on
- **seed** – (int) The initial seed for training, if None: keep current seed
- **callback** – (function (dict, dict)) -> boolean function called at every steps with state of the algorithm. It takes the local and global variables. If it returns False, training is aborted.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for tensorboard log
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseRLModel) the trained model

classmethod load(*load_path*, *env=None*, ***kwargs*)
Load the model from file

Parameters

- **load_path** – (str or file-like) the saved parameter location
- **env** – (Gym Envrionment) the new environment to run the loaded model on (can be None if you only need prediction from a trained model)
- **kwargs** – extra arguments to change the model when loading

load_parameters(*load_path_or_dict*, *exact_match=True*)
Load model parameters from a file or a dictionary

Dictionary keys should be tensorflow variable names, which can be obtained with `get_parameters` function. If `exact_match` is True, dictionary should contain keys for all model's parameters, otherwise RunTimeError is raised. If False, only variables included in the dictionary will be updated.

This does not load agent's hyper-parameters.

Warning: This function does not update trainer/optimizer variables (e.g. momentum). As such training after using this function may lead to less-than-optimal results.

Parameters

- **load_path_or_dict** – (str or file-like or dict) Save parameter location or dict of parameters as variable.name -> ndarrays to be loaded.

- **exact_match** – (bool) If True, expects load dictionary to contain keys for all variables in the model. If False, loads parameters only for variables mentioned in the dictionary. Defaults to True.

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

Parameters

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

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

pretrain(*dataset*, *n_epochs=10*, *learning_rate=0.0001*, *adam_epsilon=1e-08*, *val_interval=None*)
Pretrain a model using behavior cloning: supervised learning given an expert dataset.

NOTE: only Box and Discrete spaces are supported for now.

Parameters

- **dataset** – (ExpertDataset) Dataset manager
- **n_epochs** – (int) Number of iterations on the training set
- **learning_rate** – (float) Learning rate
- **adam_epsilon** – (float) the epsilon value for the adam optimizer
- **val_interval** – (int) Report training and validation losses every n epochs. By default, every 10th of the maximum number of epochs.

Returns (BaseRLModel) the pretrained model

save(*save_path*)
Save the current parameters to file

Parameters **save_path** – (str or file-like object) the save location

set_env(*env*)
Checks the validity of the environment, and if it is coherent, set it as the current environment.

Parameters **env** – (Gym Environment) The environment for learning a policy

setup_model()
Create all the functions and tensorflow graphs necessary to train the model

1.24 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.

Warning: The SAC model does not support `stable_baselines.common.policies` because it uses double q-values and value estimation, as a result it must use its own policy models (see *SAC Policies*).

Available Policies

MlpPolicy	Policy object that implements actor critic, using a MLP (2 layers of 64)
LnMlpPolicy	Policy object that implements actor critic, using a MLP (2 layers of 64), with layer normalisation
CnnPolicy	Policy object that implements actor critic, using a CNN (the nature CNN)
LnCnnPolicy	Policy object that implements actor critic, using a CNN (the nature CNN), with layer normalisation

1.24.1 Notes

- Original paper: https://arxiv.org/abs/1801.01290
- OpenAI Spinning Guide for SAC: https://spinningup.openai.com/en/latest/algorithms/sac.html
- Original Implementation: https://github.com/haarnoja/sac
- Blog post on using SAC with real robots: https://bair.berkeley.edu/blog/2018/12/14/sac/

Note: In our implementation, we use an entropy coefficient (as in OpenAI Spinning or Facebook Horizon), which is the equivalent to the inverse of reward scale in the original SAC paper. The main reason is that it avoids having too high errors when updating the Q functions.

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

1.24.2 Can I use?

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

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

1.24.3 Example

```
import gym
import numpy as np

from stable_baselines.sac.policies import MlpPolicy
from stable_baselines.common.vec_env import DummyVecEnv
from stable_baselines import SAC

env = gym.make('Pendulum-v0')
env = DummyVecEnv([lambda: env])

model = SAC(MlpPolicy, env, verbose=1)
model.learn(total_timesteps=50000, log_interval=10)
model.save("sac_pendulum")

del model # remove to demonstrate saving and loading

model = SAC.load("sac_pendulum")

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

1.24.4 Parameters

class stable_baselines.sac.**SAC**(*policy*, *env*, *gamma=0.99*, *learning_rate=0.0003*, *buffer_size=50000*, *learning_starts=100*, *train_freq=1*, *batch_size=64*, *tau=0.005*, *ent_coef='auto'*, *target_update_interval=1*, *gradient_steps=1*, *target_entropy='auto'*, *action_noise=None*, *random_exploration=0.0*, *verbose=0*, *tensorboard_log=None*, *_init_setup_model=True*, *policy_kwargs=None*, *full_tensorboard_log=False*)

Soft Actor-Critic (SAC) Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor, This implementation borrows code from original implementation (https://github.com/haarnoja/sac) from OpenAI Spinning Up (https://github.com/openai/spinningup) and from the Softlearning repo (https://github.com/rail-berkeley/softlearning/) Paper: https://arxiv.org/abs/1801.01290 Introduction to SAC: https://spinningup.openai.com/en/latest/algorithms/sac.html

Parameters

- **policy** – (SACPolicy or str) The policy model to use (MlpPolicy, CnnPolicy, LnMlpPolicy, ...)
- **env** – (Gym environment or str) The environment to learn from (if registered in Gym, can be str)
- **gamma** – (float) the discount factor
- **learning_rate** – (float or callable) learning rate for adam optimizer, the same learning rate will be used for all networks (Q-Values, Actor and Value function) it can be a function of the current progress (from 1 to 0)
- **buffer_size** – (int) size of the replay buffer

- **batch_size** – (int) Minibatch size for each gradient update
- **tau** – (float) the soft update coefficient ("polyak update", between 0 and 1)
- **ent_coef** – (str or float) Entropy regularization coefficient. (Equivalent to inverse of reward scale in the original SAC paper.) Controlling exploration/exploitation trade-off. Set it to 'auto' to learn it automatically (and 'auto_0.1' for using 0.1 as initial value)
- **train_freq** – (int) Update the model every *train_freq* steps.
- **learning_starts** – (int) how many steps of the model to collect transitions for before learning starts
- **target_update_interval** – (int) update the target network every *target_network_update_freq* steps.
- **gradient_steps** – (int) How many gradient update after each step
- **target_entropy** – (str or float) target entropy when learning ent_coef (ent_coef = 'auto')
- **action_noise** – (ActionNoise) the action noise type (None by default), this can help for hard exploration problem. Cf DDPG for the different action noise type.
- **random_exploration** – (float) Probability of taking a random action (as in an epsilon-greedy strategy) This is not needed for SAC normally but can help exploring when using HER + SAC. This hack was present in the original OpenAI Baselines repo (DDPG + HER)
- **verbose** – (int) the verbosity level: 0 none, 1 training information, 2 tensorflow debug
- **tensorboard_log** – (str) the log location for tensorboard (if None, no logging)
- **_init_setup_model** – (bool) Whether or not to build the network at the creation of the instance
- **policy_kwargs** – (dict) additional arguments to be passed to the policy on creation
- **full_tensorboard_log** – (bool) enable additional logging when using tensorboard Note: this has no effect on SAC logging for now

action_probability(*observation*, *state=None*, *mask=None*, *actions=None*, *logp=False*)

If `actions` is `None`, then get the model's action probability distribution from a given observation.

Depending on the action space the output is:

- Discrete: probability for each possible action
- Box: mean and standard deviation of the action output

However if `actions` is not `None`, this function will return the probability that the given actions are taken with the given parameters (observation, state, ...) on this model. For discrete action spaces, it returns the probability mass; for continuous action spaces, the probability density. This is since the probability mass will always be zero in continuous spaces, see http://blog.christianperone.com/2019/01/ for a good explanation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (np.ndarray) The last states (can be None, used in recurrent policies)
- **mask** – (np.ndarray) The last masks (can be None, used in recurrent policies)
- **actions** – (np.ndarray) (OPTIONAL) For calculating the likelihood that the given actions are chosen by the model for each of the given parameters. Must have the same

number of actions and observations. (set to None to return the complete action probability distribution)

- **logp** – (bool) (OPTIONAL) When specified with actions, returns probability in log-space. This has no effect if actions is None.

Returns (np.ndarray) the model's (log) action probability

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

Returns (Gym Environment) The current environment

get_parameter_list()
Get tensorflow Variables of model's parameters

This includes all variables necessary for continuing training (saving / loading).

Returns (list) List of tensorflow Variables

get_parameters()
Get current model parameters as dictionary of variable name -> ndarray.

Returns (OrderedDict) Dictionary of variable name -> ndarray of model's parameters.

learn(*total_timesteps*, *callback=None*, *seed=None*, *log_interval=4*, *tb_log_name='SAC'*, *reset_num_timesteps=True*, *replay_wrapper=None*)
Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples to train on
- **seed** – (int) The initial seed for training, if None: keep current seed
- **callback** – (function (dict, dict)) -> boolean function called at every steps with state of the algorithm. It takes the local and global variables. If it returns False, training is aborted.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for tensorboard log
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseRLModel) the trained model

classmethod load(*load_path*, *env=None*, ***kwargs*)
Load the model from file

Parameters

- **load_path** – (str or file-like) the saved parameter location
- **env** – (Gym Envrionment) the new environment to run the loaded model on (can be None if you only need prediction from a trained model)
- **kwargs** – extra arguments to change the model when loading

load_parameters(*load_path_or_dict*, *exact_match=True*)
Load model parameters from a file or a dictionary

Dictionary keys should be tensorflow variable names, which can be obtained with `get_parameters` function. If `exact_match` is True, dictionary should contain keys for all model's parameters, otherwise RunTimeError is raised. If False, only variables included in the dictionary will be updated.

This does not load agent's hyper-parameters.

Warning: This function does not update trainer/optimizer variables (e.g. momentum). As such training after using this function may lead to less-than-optimal results.

Parameters

- **load_path_or_dict** – (str or file-like or dict) Save parameter location or dict of parameters as variable.name -> ndarrays to be loaded.
- **exact_match** – (bool) If True, expects load dictionary to contain keys for all variables in the model. If False, loads parameters only for variables mentioned in the dictionary. Defaults to True.

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

Parameters

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

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

pretrain(*dataset*, *n_epochs=10*, *learning_rate=0.0001*, *adam_epsilon=1e-08*, *val_interval=None*)
Pretrain a model using behavior cloning: supervised learning given an expert dataset.

NOTE: only Box and Discrete spaces are supported for now.

Parameters

- **dataset** – (ExpertDataset) Dataset manager
- **n_epochs** – (int) Number of iterations on the training set
- **learning_rate** – (float) Learning rate
- **adam_epsilon** – (float) the epsilon value for the adam optimizer
- **val_interval** – (int) Report training and validation losses every n epochs. By default, every 10th of the maximum number of epochs.

Returns (BaseRLModel) the pretrained model

save(*save_path*)
Save the current parameters to file

Parameters **save_path** – (str or file-like object) the save location

set_env(*env*)
Checks the validity of the environment, and if it is coherent, set it as the current environment.

Parameters **env** – (Gym Environment) The environment for learning a policy

setup_model()
Create all the functions and tensorflow graphs necessary to train the model

1.24.5 SAC Policies

class stable_baselines.sac.**MlpPolicy**(*sess*, *ob_space*, *ac_space*, *n_env=1*, *n_steps=1*, *n_batch=None*, *reuse=False*, ***_kwargs*)

Policy object that implements actor critic, using a MLP (2 layers of 64)

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **reuse** – (bool) If the policy is reusable or not
- **_kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

action_ph

tf.Tensor: placeholder for actions, shape (self.n_batch,) + self.ac_space.shape.

initial_state

The initial state of the policy. For feedforward policies, None. For a recurrent policy, a NumPy array of shape (self.n_env,) + state_shape.

is_discrete

bool: is action space discrete.

make_actor(*obs=None*, *reuse=False*, *scope='pi'*)

Creates an actor object

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name of the actor

Returns (TensorFlow Tensor) the output tensor

make_critics(*obs=None*, *action=None*, *reuse=False*, *scope='values_fn'*, *create_vf=True*, *create_qf=True*)

Creates the two Q-Values approximator along with the Value function

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **action** – (TensorFlow Tensor) The action placeholder
- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name
- **create_vf** – (bool) Whether to create Value fn or not
- **create_qf** – (bool) Whether to create Q-Values fn or not

Returns ([tf.Tensor]) Mean, action and log probability

obs_ph
tf.Tensor: placeholder for observations, shape (self.n_batch,) + self.ob_space.shape.

proba_step(*obs*, *state=None*, *mask=None*)
Returns the action probability params (mean, std) for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float], [float])

processed_obs
tf.Tensor: processed observations, shape (self.n_batch,) + self.ob_space.shape.

The form of processing depends on the type of the observation space, and the parameters whether scale is passed to the constructor; see observation_input for more information.

step(*obs*, *state=None*, *mask=None*, *deterministic=False*)
Returns the policy for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)
- **deterministic** – (bool) Whether or not to return deterministic actions.

Returns ([float]) actions

class stable_baselines.sac.**LnMlpPolicy**(*sess*, *ob_space*, *ac_space*, *n_env=1*, *n_steps=1*, *n_batch=None*, *reuse=False*, ***_kwargs*)
Policy object that implements actor critic, using a MLP (2 layers of 64), with layer normalisation

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **reuse** – (bool) If the policy is reusable or not
- **_kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

action_ph
tf.Tensor: placeholder for actions, shape (self.n_batch,) + self.ac_space.shape.

initial_state
The initial state of the policy. For feedforward policies, None. For a recurrent policy, a NumPy array of shape (self.n_env,) + state_shape.

is_discrete
bool: is action space discrete.

make_actor(*obs=None*, *reuse=False*, *scope='pi'*)
Creates an actor object

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name of the actor

Returns (TensorFlow Tensor) the output tensor

make_critics(*obs=None*, *action=None*, *reuse=False*, *scope='values_fn'*, *create_vf=True*, *create_qf=True*)
Creates the two Q-Values approximator along with the Value function

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **action** – (TensorFlow Tensor) The action placeholder
- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name
- **create_vf** – (bool) Whether to create Value fn or not
- **create_qf** – (bool) Whether to create Q-Values fn or not

Returns ([tf.Tensor]) Mean, action and log probability

obs_ph
tf.Tensor: placeholder for observations, shape (self.n_batch,) + self.ob_space.shape.

proba_step(*obs*, *state=None*, *mask=None*)
Returns the action probability params (mean, std) for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float], [float])

processed_obs
tf.Tensor: processed observations, shape (self.n_batch,) + self.ob_space.shape.

The form of processing depends on the type of the observation space, and the parameters whether scale is passed to the constructor; see observation_input for more information.

step(*obs*, *state=None*, *mask=None*, *deterministic=False*)
Returns the policy for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)
- **deterministic** – (bool) Whether or not to return deterministic actions.

Returns ([float]) actions

class stable_baselines.sac.**CnnPolicy**(*sess*, *ob_space*, *ac_space*, *n_env=1*, *n_steps=1*, *n_batch=None*, *reuse=False*, ***_kwargs*)

Policy object that implements actor critic, using a CNN (the nature CNN)

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **reuse** – (bool) If the policy is reusable or not
- **_kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

action_ph

tf.Tensor: placeholder for actions, shape (self.n_batch,) + self.ac_space.shape.

initial_state

The initial state of the policy. For feedforward policies, None. For a recurrent policy, a NumPy array of shape (self.n_env,) + state_shape.

is_discrete

bool: is action space discrete.

make_actor(*obs=None*, *reuse=False*, *scope='pi'*)

Creates an actor object

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name of the actor

Returns (TensorFlow Tensor) the output tensor

make_critics(*obs=None*, *action=None*, *reuse=False*, *scope='values_fn'*, *create_vf=True*, *create_qf=True*)

Creates the two Q-Values approximator along with the Value function

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **action** – (TensorFlow Tensor) The action placeholder
- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name
- **create_vf** – (bool) Whether to create Value fn or not
- **create_qf** – (bool) Whether to create Q-Values fn or not

Returns ([tf.Tensor]) Mean, action and log probability

obs_ph
tf.Tensor: placeholder for observations, shape (self.n_batch,) + self.ob_space.shape.

proba_step(*obs*, *state=None*, *mask=None*)
Returns the action probability params (mean, std) for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float], [float])

processed_obs
tf.Tensor: processed observations, shape (self.n_batch,) + self.ob_space.shape.

The form of processing depends on the type of the observation space, and the parameters whether scale is passed to the constructor; see observation_input for more information.

step(*obs*, *state=None*, *mask=None*, *deterministic=False*)
Returns the policy for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)
- **deterministic** – (bool) Whether or not to return deterministic actions.

Returns ([float]) actions

class stable_baselines.sac.**LnCnnPolicy**(*sess*, *ob_space*, *ac_space*, *n_env=1*, *n_steps=1*, *n_batch=None*, *reuse=False*, ***_kwargs*)
Policy object that implements actor critic, using a CNN (the nature CNN), with layer normalisation

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **reuse** – (bool) If the policy is reusable or not
- **_kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

action_ph
tf.Tensor: placeholder for actions, shape (self.n_batch,) + self.ac_space.shape.

initial_state
The initial state of the policy. For feedforward policies, None. For a recurrent policy, a NumPy array of shape (self.n_env,) + state_shape.

is_discrete
bool: is action space discrete.

make_actor(*obs=None*, *reuse=False*, *scope='pi'*)
Creates an actor object

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name of the actor

Returns (TensorFlow Tensor) the output tensor

make_critics(*obs=None*, *action=None*, *reuse=False*, *scope='values_fn'*, *create_vf=True*, *create_qf=True*)
Creates the two Q-Values approximator along with the Value function

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **action** – (TensorFlow Tensor) The action placeholder
- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name
- **create_vf** – (bool) Whether to create Value fn or not
- **create_qf** – (bool) Whether to create Q-Values fn or not

Returns ([tf.Tensor]) Mean, action and log probability

obs_ph
tf.Tensor: placeholder for observations, shape (self.n_batch,) + self.ob_space.shape.

proba_step(*obs*, *state=None*, *mask=None*)
Returns the action probability params (mean, std) for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float], [float])

processed_obs
tf.Tensor: processed observations, shape (self.n_batch,) + self.ob_space.shape.

The form of processing depends on the type of the observation space, and the parameters whether scale is passed to the constructor; see observation_input for more information.

step(*obs*, *state=None*, *mask=None*, *deterministic=False*)
Returns the policy for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)
- **deterministic** – (bool) Whether or not to return deterministic actions.

Returns ([float]) actions

1.24.6 Custom Policy Network

Similarly to the example given in the examples page. You can easily define a custom architecture for the policy network:

```
import gym

from stable_baselines.sac.policies import FeedForwardPolicy
from stable_baselines.common.vec_env import DummyVecEnv
from stable_baselines import SAC

# Custom MLP policy of three layers of size 128 each
class CustomSACPolicy(FeedForwardPolicy):
    def __init__(self, *args, **kwargs):
        super(CustomPolicy, self).__init__(*args, **kwargs,
                                           layers=[128, 128, 128],
                                           layer_norm=False,
                                           feature_extraction="mlp")

# Create and wrap the environment
env = gym.make('Pendulum-v0')
env = DummyVecEnv([lambda: env])

model = SAC(CustomSACPolicy, env, verbose=1)
# Train the agent
model.learn(total_timesteps=100000)
```

1.25 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.

Warning: The TD3 model does not support `stable_baselines.common.policies` because it uses double q-values estimation, as a result it must use its own policy models (see *TD3 Policies*).

Available Policies

MlpPolicy	Policy object that implements actor critic, using a MLP (2 layers of 64)
LnMlpPolicy	Policy object that implements actor critic, using a MLP (2 layers of 64), with layer normalisation
CnnPolicy	Policy object that implements actor critic, using a CNN (the nature CNN)

Continued on next page

Table 5 – continued from previous page

LnCnnPolicy	Policy object that implements actor critic, using a CNN (the nature CNN), with layer normalisation

1.25.1 Notes

- Original paper: https://arxiv.org/pdf/1802.09477.pdf
- OpenAI Spinning Guide for TD3: https://spinningup.openai.com/en/latest/algorithms/td3.html
- Original Implementation: https://github.com/sfujim/TD3

Note: The default policies for TD3 differ a bit from others MlpPolicy: it uses ReLU instead of tanh activation, to match the original paper

1.25.2 Can I use?

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

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

1.25.3 Example

```
import gym
import numpy as np

from stable_baselines import TD3
from stable_baselines.td3.policies import MlpPolicy
from stable_baselines.common.vec_env import DummyVecEnv
from stable_baselines.ddpg.noise import NormalActionNoise,␣
↪OrnsteinUhlenbeckActionNoise

env = gym.make('Pendulum-v0')
env = DummyVecEnv([lambda: env])

# 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=50000, log_interval=10)
model.save("td3_pendulum")
```

(continues on next page)

(continued from previous page)

```
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()
```

1.25.4 Parameters

class stable_baselines.td3.**TD3**(*policy*, *env*, *gamma=0.99*, *learning_rate=0.0003*, *buffer_size=50000*, *learning_starts=100*, *train_freq=100*, *gradient_steps=100*, *batch_size=128*, *tau=0.005*, *policy_delay=2*, *action_noise=None*, *target_policy_noise=0.2*, *target_noise_clip=0.5*, *random_exploration=0.0*, *verbose=0*, *tensorboard_log=None*, *_init_setup_model=True*, *policy_kwargs=None*, *full_tensorboard_log=False*)

Twin Delayed DDPG (TD3) Addressing Function Approximation Error in Actor-Critic Methods.

Original implementation: https://github.com/sfujim/TD3 Paper: https://arxiv.org/pdf/1802.09477.pdf Introduction to TD3: https://spinningup.openai.com/en/latest/algorithms/td3.html

Parameters

- **policy** – (TD3Policy or str) The policy model to use (MlpPolicy, CnnPolicy, LnMlpPolicy, ...)
- **env** – (Gym environment or str) The environment to learn from (if registered in Gym, can be str)
- **gamma** – (float) the discount factor
- **learning_rate** – (float or callable) learning rate for adam optimizer, the same learning rate will be used for all networks (Q-Values and Actor networks) it can be a function of the current progress (from 1 to 0)
- **buffer_size** – (int) size of the replay buffer
- **batch_size** – (int) Minibatch size for each gradient update
- **tau** – (float) the soft update coefficient ("polyak update" of the target networks, between 0 and 1)
- **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).
- **action_noise** – (ActionNoise) the action noise type. Cf DDPG for the different action noise type.
- **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.
- **train_freq** – (int) Update the model every *train_freq* steps.

- **learning_starts** – (int) how many steps of the model to collect transitions for before learning starts
- **gradient_steps** – (int) How many gradient update after each step
- **random_exploration** – (float) Probability of taking a random action (as in an epsilon-greedy strategy) This is not needed for TD3 normally but can help exploring when using HER + TD3. This hack was present in the original OpenAI Baselines repo (DDPG + HER)
- **verbose** – (int) the verbosity level: 0 none, 1 training information, 2 tensorflow debug
- **tensorboard_log** – (str) the log location for tensorboard (if None, no logging)
- **_init_setup_model** – (bool) Whether or not to build the network at the creation of the instance
- **policy_kwargs** – (dict) additional arguments to be passed to the policy on creation
- **full_tensorboard_log** – (bool) enable additional logging when using tensorboard Note: this has no effect on TD3 logging for now

action_probability(*observation*, *state=None*, *mask=None*, *actions=None*, *logp=False*)

If `actions` is `None`, then get the model's action probability distribution from a given observation.

Depending on the action space the output is:

- Discrete: probability for each possible action
- Box: mean and standard deviation of the action output

However if `actions` is not `None`, this function will return the probability that the given actions are taken with the given parameters (observation, state, ...) on this model. For discrete action spaces, it returns the probability mass; for continuous action spaces, the probability density. This is since the probability mass will always be zero in continuous spaces, see http://blog.christianperone.com/2019/01/ for a good explanation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (np.ndarray) The last states (can be None, used in recurrent policies)
- **mask** – (np.ndarray) The last masks (can be None, used in recurrent policies)
- **actions** – (np.ndarray) (OPTIONAL) For calculating the likelihood that the given actions are chosen by the model for each of the given parameters. Must have the same number of actions and observations. (set to None to return the complete action probability distribution)
- **logp** – (bool) (OPTIONAL) When specified with actions, returns probability in log-space. This has no effect if actions is None.

Returns (np.ndarray) the model's (log) action probability

get_env()

returns the current environment (can be None if not defined)

Returns (Gym Environment) The current environment

get_parameter_list()

Get tensorflow Variables of model's parameters

This includes all variables necessary for continuing training (saving / loading).

Returns (list) List of tensorflow Variables

get_parameters()
Get current model parameters as dictionary of variable name -> ndarray.

Returns (OrderedDict) Dictionary of variable name -> ndarray of model's parameters.

learn(*total_timesteps*, *callback=None*, *seed=None*, *log_interval=4*, *tb_log_name='TD3'*, *reset_num_timesteps=True*, *replay_wrapper=None*)
Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples to train on
- **seed** – (int) The initial seed for training, if None: keep current seed
- **callback** – (function (dict, dict)) -> boolean function called at every steps with state of the algorithm. It takes the local and global variables. If it returns False, training is aborted.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for tensorboard log
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseRLModel) the trained model

classmethod load(*load_path*, *env=None*, ***kwargs*)
Load the model from file

Parameters

- **load_path** – (str or file-like) the saved parameter location
- **env** – (Gym Envrionment) the new environment to run the loaded model on (can be None if you only need prediction from a trained model)
- **kwargs** – extra arguments to change the model when loading

load_parameters(*load_path_or_dict*, *exact_match=True*)
Load model parameters from a file or a dictionary

Dictionary keys should be tensorflow variable names, which can be obtained with `get_parameters` function. If `exact_match` is True, dictionary should contain keys for all model's parameters, otherwise RunTimeError is raised. If False, only variables included in the dictionary will be updated.

This does not load agent's hyper-parameters.

Warning: This function does not update trainer/optimizer variables (e.g. momentum). As such training after using this function may lead to less-than-optimal results.

Parameters

- **load_path_or_dict** – (str or file-like or dict) Save parameter location or dict of parameters as variable.name -> ndarrays to be loaded.
- **exact_match** – (bool) If True, expects load dictionary to contain keys for all variables in the model. If False, loads parameters only for variables mentioned in the dictionary. Defaults to True.

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

Parameters

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

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

pretrain(*dataset*, *n_epochs=10*, *learning_rate=0.0001*, *adam_epsilon=1e-08*, *val_interval=None*)
Pretrain a model using behavior cloning: supervised learning given an expert dataset.

NOTE: only Box and Discrete spaces are supported for now.

Parameters

- **dataset** – (ExpertDataset) Dataset manager
- **n_epochs** – (int) Number of iterations on the training set
- **learning_rate** – (float) Learning rate
- **adam_epsilon** – (float) the epsilon value for the adam optimizer
- **val_interval** – (int) Report training and validation losses every n epochs. By default, every 10th of the maximum number of epochs.

Returns (BaseRLModel) the pretrained model

save(*save_path*)
Save the current parameters to file

Parameters **save_path** – (str or file-like object) the save location

set_env(*env*)
Checks the validity of the environment, and if it is coherent, set it as the current environment.

Parameters **env** – (Gym Environment) The environment for learning a policy

setup_model()
Create all the functions and tensorflow graphs necessary to train the model

1.25.5 TD3 Policies

class stable_baselines.td3.**MlpPolicy**(*sess*, *ob_space*, *ac_space*, *n_env=1*, *n_steps=1*, *n_batch=None*, *reuse=False*, ***_kwargs*)
Policy object that implements actor critic, using a MLP (2 layers of 64)

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **reuse** – (bool) If the policy is reusable or not

- **_kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

action_ph
tf.Tensor: placeholder for actions, shape (self.n_batch,) + self.ac_space.shape.

initial_state
The initial state of the policy. For feedforward policies, None. For a recurrent policy, a NumPy array of shape (self.n_env,) + state_shape.

is_discrete
bool: is action space discrete.

make_actor(*obs=None*, *reuse=False*, *scope='pi'*)
Creates an actor object

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name of the actor

Returns (TensorFlow Tensor) the output tensor

make_critics(*obs=None*, *action=None*, *reuse=False*, *scope='values_fn'*)
Creates the two Q-Values approximator

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **action** – (TensorFlow Tensor) The action placeholder
- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name

Returns ([tf.Tensor]) Mean, action and log probability

obs_ph
tf.Tensor: placeholder for observations, shape (self.n_batch,) + self.ob_space.shape.

proba_step(*obs*, *state=None*, *mask=None*)
Returns the policy for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) actions

processed_obs
tf.Tensor: processed observations, shape (self.n_batch,) + self.ob_space.shape.

The form of processing depends on the type of the observation space, and the parameters whether scale is passed to the constructor; see observation_input for more information.

step(*obs*, *state=None*, *mask=None*)
Returns the policy for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) actions

class stable_baselines.td3.**LnMlpPolicy**(*sess*, *ob_space*, *ac_space*, *n_env=1*, *n_steps=1*, *n_batch=None*, *reuse=False*, ***_kwargs*)

Policy object that implements actor critic, using a MLP (2 layers of 64), with layer normalisation

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **reuse** – (bool) If the policy is reusable or not
- **_kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

action_ph
tf.Tensor: placeholder for actions, shape (self.n_batch,) + self.ac_space.shape.

initial_state
The initial state of the policy. For feedforward policies, None. For a recurrent policy, a NumPy array of shape (self.n_env,) + state_shape.

is_discrete
bool: is action space discrete.

make_actor(*obs=None*, *reuse=False*, *scope='pi'*)
Creates an actor object

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name of the actor

Returns (TensorFlow Tensor) the output tensor

make_critics(*obs=None*, *action=None*, *reuse=False*, *scope='values_fn'*)
Creates the two Q-Values approximator

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **action** – (TensorFlow Tensor) The action placeholder
- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name

Returns ([tf.Tensor]) Mean, action and log probability

obs_ph
tf.Tensor: placeholder for observations, shape (self.n_batch,) + self.ob_space.shape.

proba_step(*obs*, *state=None*, *mask=None*)
Returns the policy for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) actions

processed_obs
tf.Tensor: processed observations, shape (self.n_batch,) + self.ob_space.shape.

The form of processing depends on the type of the observation space, and the parameters whether scale is passed to the constructor; see observation_input for more information.

step(*obs*, *state=None*, *mask=None*)
Returns the policy for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) actions

class stable_baselines.td3.**CnnPolicy**(*sess*, *ob_space*, *ac_space*, *n_env=1*, *n_steps=1*, *n_batch=None*, *reuse=False*, ***_kwargs*)
Policy object that implements actor critic, using a CNN (the nature CNN)

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **reuse** – (bool) If the policy is reusable or not
- **_kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

action_ph
tf.Tensor: placeholder for actions, shape (self.n_batch,) + self.ac_space.shape.

initial_state
The initial state of the policy. For feedforward policies, None. For a recurrent policy, a NumPy array of shape (self.n_env,) + state_shape.

is_discrete
bool: is action space discrete.

make_actor(*obs=None*, *reuse=False*, *scope='pi'*)
Creates an actor object

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name of the actor

Returns (TensorFlow Tensor) the output tensor

make_critics(*obs=None*, *action=None*, *reuse=False*, *scope='values_fn'*)
Creates the two Q-Values approximator

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **action** – (TensorFlow Tensor) The action placeholder
- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name

Returns ([tf.Tensor]) Mean, action and log probability

obs_ph
tf.Tensor: placeholder for observations, shape (self.n_batch,) + self.ob_space.shape.

proba_step(*obs*, *state=None*, *mask=None*)
Returns the policy for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) actions

processed_obs
tf.Tensor: processed observations, shape (self.n_batch,) + self.ob_space.shape.

The form of processing depends on the type of the observation space, and the parameters whether scale is passed to the constructor; see observation_input for more information.

step(*obs*, *state=None*, *mask=None*)
Returns the policy for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) actions

class stable_baselines.td3.**LnCnnPolicy**(*sess*, *ob_space*, *ac_space*, *n_env=1*, *n_steps=1*, *n_batch=None*, *reuse=False*, ***_kwargs*)
Policy object that implements actor critic, using a CNN (the nature CNN), with layer normalisation

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **reuse** – (bool) If the policy is reusable or not
- **_kwargs** – (dict) Extra keyword arguments for the nature CNN feature extraction

action_ph
tf.Tensor: placeholder for actions, shape (self.n_batch,) + self.ac_space.shape.

initial_state
The initial state of the policy. For feedforward policies, None. For a recurrent policy, a NumPy array of shape (self.n_env,) + state_shape.

is_discrete
bool: is action space discrete.

make_actor(*obs=None*, *reuse=False*, *scope='pi'*)
Creates an actor object

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name of the actor

Returns (TensorFlow Tensor) the output tensor

make_critics(*obs=None*, *action=None*, *reuse=False*, *scope='values_fn'*)
Creates the two Q-Values approximator

Parameters

- **obs** – (TensorFlow Tensor) The observation placeholder (can be None for default placeholder)
- **action** – (TensorFlow Tensor) The action placeholder
- **reuse** – (bool) whether or not to resue parameters
- **scope** – (str) the scope name

Returns ([tf.Tensor]) Mean, action and log probability

obs_ph
tf.Tensor: placeholder for observations, shape (self.n_batch,) + self.ob_space.shape.

proba_step(*obs*, *state=None*, *mask=None*)
Returns the policy for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment

- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) actions

processed_obs
tf.Tensor: processed observations, shape (self.n_batch,) + self.ob_space.shape.

The form of processing depends on the type of the observation space, and the parameters whether scale is passed to the constructor; see observation_input for more information.

step(*obs*, *state=None*, *mask=None*)
Returns the policy for a single step

Parameters

- **obs** – ([float] or [int]) The current observation of the environment
- **state** – ([float]) The last states (used in recurrent policies)
- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) actions

1.25.6 Custom Policy Network

Similarly to the example given in the examples page. You can easily define a custom architecture for the policy network:

```
import gym
import numpy as np

from stable_baselines import TD3
from stable_baselines.td3.policies import FeedForwardPolicy
from stable_baselines.common.vec_env import DummyVecEnv
from stable_baselines.ddpg.noise import NormalActionNoise,␣
↪OrnsteinUhlenbeckActionNoise

# Custom MLP policy with two layers
class CustomTD3Policy(FeedForwardPolicy):
    def __init__(self, *args, **kwargs):
        super(CustomTD3Policy, self).__init__(*args, **kwargs,
                                              layers=[400, 300],
                                              layer_norm=False,
                                              feature_extraction="mlp")

# Create and wrap the environment
env = gym.make('Pendulum-v0')
env = DummyVecEnv([lambda: env])

# 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(CustomTD3Policy, env, action_noise=action_noise, verbose=1)
# Train the agent
model.learn(total_timesteps=80000)
```

1.26 TRPO

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

1.26.1 Notes

- Original paper: https://arxiv.org/abs/1502.05477
- OpenAI blog post: https://blog.openai.com/openai-baselines-ppo/
- `mpirun -np 16 python -m stable_baselines.trpo_mpi.run_atari` runs the algorithm for 40M frames = 10M timesteps on an Atari game. See help (`-h`) for more options.
- `python -m stable_baselines.trpo_mpi.run_mujoco` runs the algorithm for 1M timesteps on a Mujoco environment.

1.26.2 Can I use?

- Recurrent policies:
- Multi processing: ✓ (using MPI)
- Gym spaces:

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

1.26.3 Example

```
import gym

from stable_baselines.common.policies import MlpPolicy
from stable_baselines.common.vec_env import DummyVecEnv
from stable_baselines import TRPO

env = gym.make('CartPole-v1')
env = DummyVecEnv([lambda: env])

model = TRPO(MlpPolicy, env, verbose=1)
model.learn(total_timesteps=25000)
model.save("trpo_cartpole")

del model # remove to demonstrate saving and loading

model = TRPO.load("trpo_cartpole")

obs = env.reset()
while True:
    action, _states = model.predict(obs)
```

(continues on next page)

(continued from previous page)

```
    obs, rewards, dones, info = env.step(action)
    env.render()
```

1.26.4 Parameters

class stable_baselines.trpo_mpi.**TRPO**(*policy*, *env*, *gamma=0.99*, *timesteps_per_batch=1024*, *max_kl=0.01*, *cg_iters=10*, *lam=0.98*, *entcoeff=0.0*, *cg_damping=0.01*, *vf_stepsize=0.0003*, *vf_iters=3*, *verbose=0*, *tensorboard_log=None*, *_init_setup_model=True*, *policy_kwargs=None*, *full_tensorboard_log=False*)

Trust Region Policy Optimization (https://arxiv.org/abs/1502.05477)

Parameters

- **policy** – (ActorCriticPolicy or str) The policy model to use (MlpPolicy, CnnPolicy, CnnLstmPolicy, ...)
- **env** – (Gym environment or str) The environment to learn from (if registered in Gym, can be str)
- **gamma** – (float) the discount value
- **timesteps_per_batch** – (int) the number of timesteps to run per batch (horizon)
- **max_kl** – (float) the Kullback-Leibler loss threshold
- **cg_iters** – (int) the number of iterations for the conjugate gradient calculation
- **lam** – (float) GAE factor
- **entcoeff** – (float) the weight for the entropy loss
- **cg_damping** – (float) the compute gradient dampening factor
- **vf_stepsize** – (float) the value function stepsize
- **vf_iters** – (int) the value function's number iterations for learning
- **verbose** – (int) the verbosity level: 0 none, 1 training information, 2 tensorflow debug
- **tensorboard_log** – (str) the log location for tensorboard (if None, no logging)
- **_init_setup_model** – (bool) Whether or not to build the network at the creation of the instance
- **policy_kwargs** – (dict) additional arguments to be passed to the policy on creation
- **full_tensorboard_log** – (bool) enable additional logging when using tensorboard WARNING: this logging can take a lot of space quickly

action_probability(*observation*, *state=None*, *mask=None*, *actions=None*, *logp=False*)

If `actions` is `None`, then get the model's action probability distribution from a given observation.

Depending on the action space the output is:

- Discrete: probability for each possible action
- Box: mean and standard deviation of the action output

However if `actions` is not `None`, this function will return the probability that the given actions are taken with the given parameters (observation, state, ...) on this model. For discrete action spaces, it returns the probability mass; for continuous action spaces, the probability density. This is since the probability

mass will always be zero in continuous spaces, see http://blog.christianperone.com/2019/01/ for a good explanation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (np.ndarray) The last states (can be None, used in recurrent policies)
- **mask** – (np.ndarray) The last masks (can be None, used in recurrent policies)
- **actions** – (np.ndarray) (OPTIONAL) For calculating the likelihood that the given actions are chosen by the model for each of the given parameters. Must have the same number of actions and observations. (set to None to return the complete action probability distribution)
- **logp** – (bool) (OPTIONAL) When specified with actions, returns probability in log-space. This has no effect if actions is None.

Returns (np.ndarray) the model's (log) action probability

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

Returns (Gym Environment) The current environment

get_parameter_list()
Get tensorflow Variables of model's parameters

This includes all variables necessary for continuing training (saving / loading).

Returns (list) List of tensorflow Variables

get_parameters()
Get current model parameters as dictionary of variable name -> ndarray.

Returns (OrderedDict) Dictionary of variable name -> ndarray of model's parameters.

learn(*total_timesteps*, *callback=None*, *seed=None*, *log_interval=100*, *tb_log_name='TRPO'*, *reset_num_timesteps=True*)
Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples to train on
- **seed** – (int) The initial seed for training, if None: keep current seed
- **callback** – (function (dict, dict)) -> boolean function called at every steps with state of the algorithm. It takes the local and global variables. If it returns False, training is aborted.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for tensorboard log
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseRLModel) the trained model

classmethod load(*load_path*, *env=None*, ***kwargs*)
Load the model from file

Parameters

- **load_path** – (str or file-like) the saved parameter location

- **env** – (Gym Envrionment) the new environment to run the loaded model on (can be None if you only need prediction from a trained model)
- **kwargs** – extra arguments to change the model when loading

load_parameters(*load_path_or_dict*, *exact_match=True*)
Load model parameters from a file or a dictionary

Dictionary keys should be tensorflow variable names, which can be obtained with `get_parameters` function. If `exact_match` is True, dictionary should contain keys for all model's parameters, otherwise RunTimeError is raised. If False, only variables included in the dictionary will be updated.

This does not load agent's hyper-parameters.

Warning: This function does not update trainer/optimizer variables (e.g. momentum). As such training after using this function may lead to less-than-optimal results.

Parameters

- **load_path_or_dict** – (str or file-like or dict) Save parameter location or dict of parameters as variable.name -> ndarrays to be loaded.
- **exact_match** – (bool) If True, expects load dictionary to contain keys for all variables in the model. If False, loads parameters only for variables mentioned in the dictionary. Defaults to True.

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

Parameters

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

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

pretrain(*dataset*, *n_epochs=10*, *learning_rate=0.0001*, *adam_epsilon=1e-08*, *val_interval=None*)
Pretrain a model using behavior cloning: supervised learning given an expert dataset.

NOTE: only Box and Discrete spaces are supported for now.

Parameters

- **dataset** – (ExpertDataset) Dataset manager
- **n_epochs** – (int) Number of iterations on the training set
- **learning_rate** – (float) Learning rate
- **adam_epsilon** – (float) the epsilon value for the adam optimizer
- **val_interval** – (int) Report training and validation losses every n epochs. By default, every 10th of the maximum number of epochs.

Returns (BaseRLModel) the pretrained model

save(*save_path*)
Save the current parameters to file

Parameters **save_path** – (str or file-like object) the save location

set_env(*env*)
Checks the validity of the environment, and if it is coherent, set it as the current environment.

Parameters **env** – (Gym Environment) The environment for learning a policy

setup_model()
Create all the functions and tensorflow graphs necessary to train the model

1.27 Probability Distributions

Probability distributions used for the different action spaces:

- `CategoricalProbabilityDistribution` -> Discrete
- `DiagGaussianProbabilityDistribution` -> Box (continuous actions)
- `MultiCategoricalProbabilityDistribution` -> MultiDiscrete
- `BernoulliProbabilityDistribution` -> MultiBinary

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 `CategoricalProbabilityDistribution` allows to sample from it, computes the entropy, the negative log probability (`neglogp`) 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 `DiagGaussianProbabilityDistribution`).

class stable_baselines.common.distributions.**BernoulliProbabilityDistribution**(*logits*)

entropy()
Returns shannon's entropy of the probability

Returns (float) the entropy

flatparam()
Return the direct probabilities

Returns ([float]) the probabilites

classmethod fromflat(*flat*)
Create an instance of this from new bernoulli input

Parameters **flat** – ([float]) the bernoulli input data

Returns (ProbabilityDistribution) the instance from the given bernoulli input data

kl(*other*)
Calculates the Kullback-Leibler divergence from the given probabilty distribution

Parameters **other** – ([float]) the distibution to compare with

Returns (float) the KL divergence of the two distributions

mode()
Returns the probability

Returns (Tensorflow Tensor) the deterministic action

neglogp(*x*)
returns the of the negative log likelihood

Parameters **x** – (str) the labels of each index

Returns ([float]) The negative log likelihood of the distribution

sample()
returns a sample from the probabilty distribution

Returns (Tensorflow Tensor) the stochastic action

class stable_baselines.common.distributions.**BernoulliProbabilityDistributionType**(*size*)

param_shape()
returns the shape of the input parameters

Returns ([int]) the shape

proba_distribution_from_latent(*pi_latent_vector*, *vf_latent_vector*, *init_scale=1.0*, *init_bias=0.0*)
returns the probability distribution from latent values

Parameters

- **pi_latent_vector** – ([float]) the latent pi values
- **vf_latent_vector** – ([float]) the latent vf values
- **init_scale** – (float) the inital scale of the distribution
- **init_bias** – (float) the inital bias of the distribution

Returns (ProbabilityDistribution) the instance of the ProbabilityDistribution associated

probability_distribution_class()
returns the ProbabilityDistribution class of this type

Returns (Type ProbabilityDistribution) the probability distribution class associated

sample_dtype()
returns the type of the sampling

Returns (type) the type

sample_shape()
returns the shape of the sampling

Returns ([int]) the shape

class stable_baselines.common.distributions.**CategoricalProbabilityDistribution**(*logits*)

entropy()
Returns shannon's entropy of the probability

Returns (float) the entropy

flatparam()
Return the direct probabilities

Returns ([float]) the probabilites

classmethod fromflat(*flat*)
Create an instance of this from new logits values

Parameters **flat** – ([float]) the categorical logits input

Returns (ProbabilityDistribution) the instance from the given categorical input

kl(*other*)
Calculates the Kullback-Leibler divergence from the given probabilty distribution

Parameters **other** – ([float]) the distibution to compare with

Returns (float) the KL divergence of the two distributions

mode()
Returns the probability

Returns (Tensorflow Tensor) the deterministic action

neglogp(*x*)
returns the of the negative log likelihood

Parameters **x** – (str) the labels of each index

Returns ([float]) The negative log likelihood of the distribution

sample()
returns a sample from the probabilty distribution

Returns (Tensorflow Tensor) the stochastic action

class stable_baselines.common.distributions.**CategoricalProbabilityDistributionType**(*n_cat*)

param_shape()
returns the shape of the input parameters

Returns ([int]) the shape

proba_distribution_from_latent(*pi_latent_vector*, *vf_latent_vector*, *init_scale=1.0*, *init_bias=0.0*)
returns the probability distribution from latent values

Parameters

- **pi_latent_vector** – ([float]) the latent pi values
- **vf_latent_vector** – ([float]) the latent vf values
- **init_scale** – (float) the inital scale of the distribution
- **init_bias** – (float) the inital bias of the distribution

Returns (ProbabilityDistribution) the instance of the ProbabilityDistribution associated

probability_distribution_class()
returns the ProbabilityDistribution class of this type

Returns (Type ProbabilityDistribution) the probability distribution class associated

sample_dtype()
returns the type of the sampling

Returns (type) the type

sample_shape()
returns the shape of the sampling

Returns ([int]) the shape

class stable_baselines.common.distributions.**DiagGaussianProbabilityDistribution**(*flat*)

entropy()
Returns shannon's entropy of the probability

Returns (float) the entropy

flatparam()
Return the direct probabilities

Returns ([float]) the probabilites

classmethod fromflat(*flat*)
Create an instance of this from new multivariate gaussian input

Parameters **flat** – ([float]) the multivariate gaussian input data

Returns (ProbabilityDistribution) the instance from the given multivariate gaussian input data

kl(*other*)
Calculates the Kullback-Leibler divergence from the given probabilty distribution

Parameters **other** – ([float]) the distibution to compare with

Returns (float) the KL divergence of the two distributions

mode()
Returns the probability

Returns (Tensorflow Tensor) the deterministic action

neglogp(*x*)
returns the of the negative log likelihood

Parameters **x** – (str) the labels of each index

Returns ([float]) The negative log likelihood of the distribution

sample()
returns a sample from the probabilty distribution

Returns (Tensorflow Tensor) the stochastic action

class stable_baselines.common.distributions.**DiagGaussianProbabilityDistributionType**(*size*)

param_shape()
returns the shape of the input parameters

Returns ([int]) the shape

proba_distribution_from_flat(*flat*)
returns the probability distribution from flat probabilities

Parameters **flat** – ([float]) the flat probabilities

Returns (ProbabilityDistribution) the instance of the ProbabilityDistribution associated

proba_distribution_from_latent(*pi_latent_vector*, *vf_latent_vector*, *init_scale=1.0*, *init_bias=0.0*)
returns the probability distribution from latent values

Parameters

- **pi_latent_vector** – ([float]) the latent pi values
- **vf_latent_vector** – ([float]) the latent vf values

- **init_scale** – (float) the inital scale of the distribution
- **init_bias** – (float) the inital bias of the distribution

Returns (ProbabilityDistribution) the instance of the ProbabilityDistribution associated

probability_distribution_class()
returns the ProbabilityDistribution class of this type

Returns (Type ProbabilityDistribution) the probability distribution class associated

sample_dtype()
returns the type of the sampling

Returns (type) the type

sample_shape()
returns the shape of the sampling

Returns ([int]) the shape

class stable_baselines.common.distributions.**MultiCategoricalProbabilityDistribution**(*nvec*, *flat*)

entropy()
Returns shannon's entropy of the probability

Returns (float) the entropy

flatparam()
Return the direct probabilities

Returns ([float]) the probabilites

classmethod fromflat(*flat*)
Create an instance of this from new logits values

Parameters **flat** – ([float]) the multi categorical logits input

Returns (ProbabilityDistribution) the instance from the given multi categorical input

kl(*other*)
Calculates the Kullback-Leibler divergence from the given probabilty distribution

Parameters **other** – ([float]) the distibution to compare with

Returns (float) the KL divergence of the two distributions

mode()
Returns the probability

Returns (Tensorflow Tensor) the deterministic action

neglogp(*x*)
returns the of the negative log likelihood

Parameters **x** – (str) the labels of each index

Returns ([float]) The negative log likelihood of the distribution

sample()
returns a sample from the probabilty distribution

Returns (Tensorflow Tensor) the stochastic action

class stable_baselines.common.distributions.**MultiCategoricalProbabilityDistributionType**(*n_v*

param_shape()
returns the shape of the input parameters

Returns ([int]) the shape

proba_distribution_from_flat(*flat*)
Returns the probability distribution from flat probabilities flat: flattened vector of parameters of probability distribution

Parameters **flat** – ([float]) the flat probabilities

Returns (ProbabilityDistribution) the instance of the ProbabilityDistribution associated

proba_distribution_from_latent(*pi_latent_vector*, *vf_latent_vector*, *init_scale=1.0*, *init_bias=0.0*)
returns the probability distribution from latent values

Parameters

- **pi_latent_vector** – ([float]) the latent pi values
- **vf_latent_vector** – ([float]) the latent vf values
- **init_scale** – (float) the inital scale of the distribution
- **init_bias** – (float) the inital bias of the distribution

Returns (ProbabilityDistribution) the instance of the ProbabilityDistribution associated

probability_distribution_class()
returns the ProbabilityDistribution class of this type

Returns (Type ProbabilityDistribution) the probability distribution class associated

sample_dtype()
returns the type of the sampling

Returns (type) the type

sample_shape()
returns the shape of the sampling

Returns ([int]) the shape

class stable_baselines.common.distributions.**ProbabilityDistribution**
A particular probability distribution

entropy()
Returns shannon's entropy of the probability

Returns (float) the entropy

flatparam()
Return the direct probabilities

Returns ([float]) the probabilites

kl(*other*)
Calculates the Kullback-Leibler divergence from the given probabilty distribution

Parameters **other** – ([float]) the distibution to compare with

Returns (float) the KL divergence of the two distributions

logp(*x*)
returns the of the log likelihood

Parameters **x** – (str) the labels of each index

Returns ([float]) The log likelihood of the distribution

mode()
Returns the probability

Returns (Tensorflow Tensor) the deterministic action

neglogp(*x*)
returns the of the negative log likelihood

Parameters **x** – (str) the labels of each index

Returns ([float]) The negative log likelihood of the distribution

sample()
returns a sample from the probabilty distribution

Returns (Tensorflow Tensor) the stochastic action

class stable_baselines.common.distributions.**ProbabilityDistributionType**
Parametrized family of probability distributions

param_placeholder(*prepend_shape*, *name=None*)
returns the TensorFlow placeholder for the input parameters

Parameters

- **prepend_shape** – ([int]) the prepend shape
- **name** – (str) the placeholder name

Returns (TensorFlow Tensor) the placeholder

param_shape()
returns the shape of the input parameters

Returns ([int]) the shape

proba_distribution_from_flat(*flat*)
Returns the probability distribution from flat probabilities flat: flattened vector of parameters of probability distribution

Parameters **flat** – ([float]) the flat probabilities

Returns (ProbabilityDistribution) the instance of the ProbabilityDistribution associated

proba_distribution_from_latent(*pi_latent_vector*, *vf_latent_vector*, *init_scale=1.0*, *init_bias=0.0*)
returns the probability distribution from latent values

Parameters

- **pi_latent_vector** – ([float]) the latent pi values
- **vf_latent_vector** – ([float]) the latent vf values
- **init_scale** – (float) the inital scale of the distribution
- **init_bias** – (float) the inital bias of the distribution

Returns (ProbabilityDistribution) the instance of the ProbabilityDistribution associated

probability_distribution_class()
returns the ProbabilityDistribution class of this type

Returns (Type ProbabilityDistribution) the probability distribution class associated

sample_dtype()
returns the type of the sampling

Returns (type) the type

sample_placeholder(*prepend_shape*, *name=None*)
returns the TensorFlow placeholder for the sampling

Parameters

- **prepend_shape** – ([int]) the prepend shape
- **name** – (str) the placeholder name

Returns (TensorFlow Tensor) the placeholder

sample_shape()
returns the shape of the sampling

Returns ([int]) the shape

stable_baselines.common.distributions.**make_proba_dist_type**(*ac_space*)
return an instance of ProbabilityDistributionType for the correct type of action space

Parameters **ac_space** – (Gym Space) the input action space

Returns (ProbabilityDistributionType) the approriate instance of a ProbabilityDistributionType

stable_baselines.common.distributions.**shape_el**(*tensor*, *index*)
get the shape of a TensorFlow Tensor element

Parameters

- **tensor** – (TensorFlow Tensor) the input tensor
- **index** – (int) the element

Returns ([int]) the shape

1.28 Tensorflow Utils

stable_baselines.common.tf_util.**conv2d**(*input_tensor*, *num_filters*, *name*, *filter_size=(3, 3)*, *stride=(1, 1)*, *pad='SAME'*, *dtype=<MagicMock id='139846110431104'>*, *collections=None*, *summary_tag=None*)
Creates a 2d convolutional layer for TensorFlow

Parameters

- **input_tensor** – (TensorFlow Tensor) The input tensor for the convolution
- **num_filters** – (int) The number of filters
- **name** – (str) The TensorFlow variable scope
- **filter_size** – (tuple) The filter size
- **stride** – (tuple) The stride of the convolution
- **pad** – (str) The padding type ('VALID' or 'SAME')
- **dtype** – (type) The data type for the Tensors
- **collections** – (list) List of graph collections keys to add the Variable to

- **summary_tag** – (str) image summary name, can be None for no image summary

Returns (TensorFlow Tensor) 2d convolutional layer

stable_baselines.common.tf_util.**display_var_info**(*_vars*)
log variable information, for debug purposes

Parameters **_vars** – ([TensorFlow Tensor]) the variables

stable_baselines.common.tf_util.**flatgrad**(*loss*, *var_list*, *clip_norm=None*)
calculates the gradient and flattens it

Parameters

- **loss** – (float) the loss value
- **var_list** – ([TensorFlow Tensor]) the variables
- **clip_norm** – (float) clip the gradients (disabled if None)

Returns ([TensorFlow Tensor]) flattend gradient

stable_baselines.common.tf_util.**flattenallbut0**(*tensor*)
flatten all the dimension, except from the first one

Parameters **tensor** – (TensorFlow Tensor) the input tensor

Returns (TensorFlow Tensor) the flattened tensor

stable_baselines.common.tf_util.**function**(*inputs*, *outputs*, *updates=None*, *givens=None*)
Take a bunch of tensorflow placeholders and expressions computed based on those placeholders and produces f(inputs) -> outputs. Function f takes values to be fed to the input's placeholders and produces the values of the expressions in outputs. Just like a Theano function.

Input values can be passed in the same order as inputs or can be provided as kwargs based on placeholder name (passed to constructor or accessible via placeholder.op.name).

Example:

```
>>> x = tf.placeholder(tf.int32, (), name="x")
>>> y = tf.placeholder(tf.int32, (), name="y")
>>> z = 3 * x + 2 * y
>>> lin = function([x, y], z, givens={y: 0})
>>> with single_threaded_session():
>>>     initialize()
>>>     assert lin(2) == 6
>>>     assert lin(x=3) == 9
>>>     assert lin(2, 2) == 10
```

Parameters

- **inputs** – (TensorFlow Tensor or Object with make_feed_dict) list of input arguments
- **outputs** – (TensorFlow Tensor) list of outputs or a single output to be returned from function. Returned value will also have the same shape.
- **updates** – ([tf.Operation] or tf.Operation) list of update functions or single update function that will be run whenever the function is called. The return is ignored.
- **givens** – (dict) the values known for the output

stable_baselines.common.tf_util.**get_globals_vars**(*name*)
returns the trainable variables

Parameters **name** – (str) the scope

Returns ([TensorFlow Variable])

stable_baselines.common.tf_util.**get_trainable_vars**(*name*)
returns the trainable variables

Parameters **name** – (str) the scope

Returns ([TensorFlow Variable])

stable_baselines.common.tf_util.**huber_loss**(*tensor*, *delta=1.0*)
Reference: https://en.wikipedia.org/wiki/Huber_loss

Parameters

- **tensor** – (TensorFlow Tensor) the input value
- **delta** – (float) huber loss delta value

Returns (TensorFlow Tensor) huber loss output

stable_baselines.common.tf_util.**in_session**(*func*)
wrappes a function so that it is in a TensorFlow Session

Parameters **func** – (function) the function to wrap

Returns (function)

stable_baselines.common.tf_util.**initialize**(*sess=None*)
Initialize all the uninitialized variables in the global scope.

Parameters **sess** – (TensorFlow Session)

stable_baselines.common.tf_util.**intprod**(*tensor*)
calculates the product of all the elements in a list

Parameters **tensor** – ([Number]) the list of elements

Returns (int) the product truncated

stable_baselines.common.tf_util.**is_image**(*tensor*)
Check if a tensor has the shape of a valid image for tensorboard logging. Valid image: RGB, RGBD, GrayScale

Parameters **tensor** – (np.ndarray or tf.placeholder)

Returns (bool)

stable_baselines.common.tf_util.**leaky_relu**(*tensor*, *leak=0.2*)
Leaky ReLU http://web.stanford.edu/~awni/papers/relu_hybrid_icml2013_final.pdf

Parameters

- **tensor** – (float) the input value
- **leak** – (float) the leaking coeficient when the function is saturated

Returns (float) Leaky ReLU output

stable_baselines.common.tf_util.**load_state**(*fname*, *sess=None*, *var_list=None*)
Load a TensorFlow saved model

Parameters

- **fname** – (str) the graph name
- **sess** – (TensorFlow Session) the session, if None: get_default_session()

- **var_list** – ([TensorFlow Tensor] or dict(str: TensorFlow Tensor)) A list of Variable/SaveableObject, or a dictionary mapping names to SaveableObject's. If `None`, defaults to the list of all saveable objects.

`stable_baselines.common.tf_util.`**`make_session`**(*num_cpu=None*, *make_default=False*, *graph=None*)

Returns a session that will use <num_cpu> CPU's only

Parameters

- **num_cpu** – (int) number of CPUs to use for TensorFlow
- **make_default** – (bool) if this should return an InteractiveSession or a normal Session
- **graph** – (TensorFlow Graph) the graph of the session

Returns (TensorFlow session)

`stable_baselines.common.tf_util.`**`normc_initializer`**(*std=1.0*, *axis=0*)

Return a parameter initializer for TensorFlow

Parameters

- **std** – (float) standard deviation
- **axis** – (int) the axis to normalize on

Returns (function)

`stable_baselines.common.tf_util.`**`numel`**(*tensor*)

get TensorFlow Tensor's number of elements

Parameters **tensor** – (TensorFlow Tensor) the input tensor

Returns (int) the number of elements

`stable_baselines.common.tf_util.`**`outer_scope_getter`**(*scope*, *new_scope=''*)

remove a scope layer for the getter

Parameters

- **scope** – (str) the layer to remove
- **new_scope** – (str) optional replacement name

Returns (function (function, str, `*args`, `**kwargs`): Tensorflow Tensor)

`stable_baselines.common.tf_util.`**`save_state`**(*fname*, *sess=None*, *var_list=None*)

Save a TensorFlow model

Parameters

- **fname** – (str) the graph name
- **sess** – (TensorFlow Session) The tf session, if None, get_default_session()
- **var_list** – ([TensorFlow Tensor] or dict(str: TensorFlow Tensor)) A list of Variable/SaveableObject, or a dictionary mapping names to SaveableObject's. If `None`, defaults to the list of all saveable objects.

`stable_baselines.common.tf_util.`**`single_threaded_session`**(*make_default=False*, *graph=None*)

Returns a session which will only use a single CPU

Parameters

- **make_default** – (bool) if this should return an InteractiveSession or a normal Session

- **graph** – (TensorFlow Graph) the graph of the session

Returns (TensorFlow session)

stable_baselines.common.tf_util.**switch**(*condition*, *then_expression*, *else_expression*)
Switches between two operations depending on a scalar value (int or bool). Note that both *then_expression* and *else_expression* should be symbolic tensors of the *same shape*.

Parameters

- **condition** – (TensorFlow Tensor) scalar tensor.
- **then_expression** – (TensorFlow Operation)
- **else_expression** – (TensorFlow Operation)

Returns (TensorFlow Operation) the switch output

stable_baselines.common.tf_util.**var_shape**(*tensor*)
get TensorFlow Tensor shape

Parameters **tensor** – (TensorFlow Tensor) the input tensor

Returns ([int]) the shape

1.29 Command Utils

Helpers for scripts like run_atari.py.

stable_baselines.common.cmd_util.**arg_parser**()
Create an empty argparse.ArgumentParser.

Returns (ArgumentParser)

stable_baselines.common.cmd_util.**atari_arg_parser**()
Create an argparse.ArgumentParser for run_atari.py.

Returns (ArgumentParser) parser {'–env': 'BreakoutNoFrameskip-v4', '–seed': 0, '–num-timesteps': int(1e7)}

stable_baselines.common.cmd_util.**make_atari_env**(*env_id*, *num_env*, *seed*, *wrapper_kwargs=None*, *start_index=0*, *allow_early_resets=True*, *start_method=None*)
Create a wrapped, monitored SubprocVecEnv for Atari.

Parameters

- **env_id** – (str) the environment ID
- **num_env** – (int) the number of environment you wish to have in subprocesses
- **seed** – (int) the inital seed for RNG
- **wrapper_kwargs** – (dict) the parameters for wrap_deepmind function
- **start_index** – (int) start rank index
- **allow_early_resets** – (bool) allows early reset of the environment
- **start_method** – (str) method used to start the subprocesses. See SubprocVecEnv doc for more information

Returns (Gym Environment) The atari environment

stable_baselines.common.cmd_util.**make_mujoco_env**(*env_id*, *seed*, *allow_early_resets=True*)

Create a wrapped, monitored gym.Env for MuJoCo.

Parameters

- **env_id** – (str) the environment ID
- **seed** – (int) the inital seed for RNG
- **allow_early_resets** – (bool) allows early reset of the environment

Returns (Gym Environment) The mujoco environment

stable_baselines.common.cmd_util.**make_robotics_env**(*env_id*, *seed*, *rank=0*, *allow_early_resets=True*)

Create a wrapped, monitored gym.Env for MuJoCo.

Parameters

- **env_id** – (str) the environment ID
- **seed** – (int) the inital seed for RNG
- **rank** – (int) the rank of the environment (for logging)
- **allow_early_resets** – (bool) allows early reset of the environment

Returns (Gym Environment) The robotic environment

stable_baselines.common.cmd_util.**mujoco_arg_parser**()

Create an argparse.ArgumentParser for run_mujoco.py.

Returns (ArgumentParser) parser {'–env': 'Reacher-v2', '–seed': 0, '–num-timesteps': int(1e6), '–play': False}

stable_baselines.common.cmd_util.**robotics_arg_parser**()

Create an argparse.ArgumentParser for run_mujoco.py.

Returns (ArgumentParser) parser {'–env': 'FetchReach-v0', '–seed': 0, '–num-timesteps': int(1e6)}

1.30 Schedules

Schedules are used as hyperparameter for most of the algortihms, in order to change value of a parameter over time (usuallly the learning rate).

This file is used for specifying various schedules that evolve over time throughout the execution of the algorithm, such as:

- learning rate for the optimizer
- exploration epsilon for the epsilon greedy exploration strategy
- beta parameter for beta parameter in prioritized replay

Each schedule has a function *value(t)* which returns the current value of the parameter given the timestep t of the optimization procedure.

class stable_baselines.common.schedules.**ConstantSchedule**(*value*)

Value remains constant over time.

Parameters **value** – (float) Constant value of the schedule

value(*step*)
Value of the schedule for a given timestep

Parameters **step** – (int) the timestep

Returns (float) the output value for the given timestep

class stable_baselines.common.schedules.**LinearSchedule**(*schedule_timesteps*, *final_p*, *initial_p=1.0*)
Linear interpolation between initial_p and final_p over schedule_timesteps. After this many timesteps pass final_p is returned.

Parameters

- **schedule_timesteps** – (int) Number of timesteps for which to linearly anneal initial_p to final_p
- **initial_p** – (float) initial output value
- **final_p** – (float) final output value

value(*step*)
Value of the schedule for a given timestep

Parameters **step** – (int) the timestep

Returns (float) the output value for the given timestep

class stable_baselines.common.schedules.**PiecewiseSchedule**(*endpoints*, *interpolation=<function linear_interpolation>*, *outside_value=None*)
Piecewise schedule.

Parameters

- **endpoints** – ([(int, int)]) list of pairs *(time, value)* meanining that schedule should output *value* when *t==time*. All the values for time must be sorted in an increasing order. When t is between two times, e.g. *(time_a, value_a)* and *(time_b, value_b)*, such that *time_a <= t < time_b* then value outputs *interpolation(value_a, value_b, alpha)* where alpha is a fraction of time passed between *time_a* and *time_b* for time *t*.
- **interpolation** – (lambda (float, float, float): float) a function that takes value to the left and to the right of t according to the *endpoints*. Alpha is the fraction of distance from left endpoint to right endpoint that t has covered. See linear_interpolation for example.
- **outside_value** – (float) if the value is requested outside of all the intervals sepecified in *endpoints* this value is returned. If None then AssertionError is raised when outside value is requested.

value(*step*)
Value of the schedule for a given timestep

Parameters **step** – (int) the timestep

Returns (float) the output value for the given timestep

stable_baselines.common.schedules.**linear_interpolation**(*left*, *right*, *alpha*)
Linear interpolation between *left* and *right*.

Parameters

- **left** – (float) left boundary
- **right** – (float) right boundary

- **alpha** – (float) coeff in [0, 1]

Returns (float)

1.31 Changelog

For download links, please look at Github release page.

1.31.1 Release 2.7.0 (2019-07-31)

Twin Delayed DDPG (TD3) and GAE bug fix (TRPO, PPO1, GAIL)

Breaking Changes:

New Features:

- added Twin Delayed DDPG (TD3) algorithm, with HER support
- added support for continuous action spaces to *action_probability*, computing the PDF of a Gaussian policy in addition to the existing support for categorical stochastic policies.
- added flag to *action_probability* to return log-probabilities.
- added support for python lists and numpy arrays in `logger.writekvs`. (@dwiel)
- the info dict returned by VecEnvs now include a `terminal_observation` key providing access to the last observation in a trajectory. (@qxcv)

Bug Fixes:

- fixed a bug in `traj_segment_generator` where the `episode_starts` was wrongly recorded, resulting in wrong calculation of Generalized Advantage Estimation (GAE), this affects TRPO, PPO1 and GAIL (thanks to @miguelrass for spotting the bug)
- added missing property *n_batch* in *BasePolicy*.

Deprecations:

Others:

- renamed some keys in `traj_segment_generator` to be more meaningful
- retrieve unnormalized reward when using Monitor wrapper with TRPO, PPO1 and GAIL to display them in the logs (mean episode reward)
- clean up DDPG code (renamed variables)

Documentation:

- doc fix for the hyperparameter tuning command in the rl zoo
- added an example on how to log additional variable with tensorboard and a callback

1.31.2 Release 2.6.0 (2019-06-12)

Hindsight Experience Replay (HER) - Reloaded | get/load parameters

Breaking Changes:

- **breaking change** removed `stable_baselines.ddpg.memory` in favor of `stable_baselines.deepq.replay_buffer` (see fix below)

Breaking Change: DDPG replay buffer was unified with DQN/SAC replay buffer. As a result, when loading a DDPG model trained with stable_baselines<2.6.0, it throws an import error. You can fix that using:

```
import sys
import pkg_resources

import stable_baselines

# Fix for breaking change for DDPG buffer in v2.6.0
if pkg_resources.get_distribution("stable_baselines").version >= "2.6.0":
    sys.modules['stable_baselines.ddpg.memory'] = stable_baselines.deepq.replay_buffer
    stable_baselines.deepq.replay_buffer.Memory = stable_baselines.deepq.replay_
↪buffer.ReplayBuffer
```

We recommend you to save again the model afterward, so the fix won't be needed the next time the trained agent is loaded.

New Features:

- **revamped HER implementation**: clean re-implementation from scratch, now supports DQN, SAC and DDPG
- add `action_noise` param for SAC, it helps exploration for problem with deceptive reward
- The parameter `filter_size` of the function `conv` in A2C utils now supports passing a list/tuple of two integers (height and width), in order to have non-squared kernel matrix. (@yutingsz)
- add `random_exploration` parameter for DDPG and SAC, it may be useful when using HER + DDPG/SAC. This hack was present in the original OpenAI Baselines DDPG + HER implementation.
- added `load_parameters` and `get_parameters` to base RL class. With these methods, users are able to load and get parameters to/from existing model, without touching tensorflow. (@Miffyli)
- added specific hyperparameter for PPO2 to clip the value function (`cliprange_vf`)
- added `VecCheckNan` wrapper

Bug Fixes:

- bugfix for `VecEnvWrapper.__getattr__` which enables access to class attributes inherited from parent classes.
- fixed path splitting in `TensorboardWriter._get_latest_run_id()` on Windows machines (@PatrickWalter214)
- fixed a bug where initial learning rate is logged instead of its placeholder in `A2C.setup_model` (@sc420)
- fixed a bug where number of timesteps is incorrectly updated and logged in `A2C.learn` and `A2C._train_step` (@sc420)

- fixed `num_timesteps` (total_timesteps) variable in PPO2 that was wrongly computed.
- fixed a bug in DDPG/DQN/SAC, when there were the number of samples in the replay buffer was lesser than the batch size (thanks to @dwiel for spotting the bug)
- **removed** `a2c.utils.find_trainable_params` please use `common.tf_util.get_trainable_vars` instead. `find_trainable_params` was returning all trainable variables, discarding the scope argument. This bug was causing the model to save duplicated parameters (for DDPG and SAC) but did not affect the performance.

Deprecations:

- **deprecated** `memory_limit` and `memory_policy` in DDPG, please use `buffer_size` instead. (will be removed in v3.x.x)

Others:

- **important change** switched to using dictionaries rather than lists when storing parameters, with tensorflow Variable names being the keys. (@Miffyli)
- removed unused dependencies (tdqm, dill, progressbar2, seaborn, glob2, click)
- removed `get_available_gpus` function which hadn't been used anywhere (@Pastafarianist)

Documentation:

- added guide for managing `NaN` and `inf`
- updated ven_env doc
- misc doc updates

1.31.3 Release 2.5.1 (2019-05-04)

Bug fixes + improvements in the VecEnv

Warning: breaking changes when using custom policies

- doc update (fix example of result plotter + improve doc)
- fixed logger issues when stdout lacks `read` function
- fixed a bug in `common.dataset.Dataset` where shuffling was not disabled properly (it affects only PPO1 with recurrent policies)
- fixed output layer name for DDPG q function, used in pop-art normalization and l2 regularization of the critic
- added support for multi env recording to `generate_expert_traj` (@XMaster96)
- added support for LSTM model recording to `generate_expert_traj` (@XMaster96)
- `GAIL`: remove mandatory matplotlib dependency and refactor as subclass of `TRPO` (@kantneel and @AdamGleave)
- added `get_attr()`, `env_method()` and `set_attr()` methods for all VecEnv. Those methods now all accept `indices` keyword to select a subset of envs. `set_attr` now returns `None` rather than a list of `None`. (@kantneel)

- GAIL: `gail.dataset.ExpertDataset` supports loading from memory rather than file, and `gail.dataset.record_expert` supports returning in-memory rather than saving to file.
- added support in `VecEnvWrapper` for accessing attributes of arbitrarily deeply nested instances of `VecEnvWrapper` and `VecEnv`. This is allowed as long as the attribute belongs to exactly one of the nested instances i.e. it must be unambiguous. (@kantneel)
- fixed bug where result plotter would crash on very short runs (@Pastafarianist)
- added option to not trim output of result plotter by number of timesteps (@Pastafarianist)
- clarified the public interface of `BasePolicy` and `ActorCriticPolicy`. **Breaking change** when using custom policies: `masks_ph` is now called `dones_ph`, and most placeholders were made private: e.g. `self.value_fn` is now `self._value_fn`
- support for custom stateful policies.
- fixed episode length recording in `trpo_mpi.utils.traj_segment_generator` (@GerardMaggiolino)

1.31.4 Release 2.5.0 (2019-03-28)

Working GAIL, pretrain RL models and hotfix for A2C with continuous actions

- fixed various bugs in GAIL
- added scripts to generate dataset for gail
- added tests for GAIL + data for Pendulum-v0
- removed unused `utils` file in DQN folder
- fixed a bug in A2C where actions were cast to `int32` even in the continuous case
- added addional logging to A2C when Monitor wrapper is used
- changed logging for PPO2: do not display NaN when reward info is not present
- change default value of A2C lr schedule
- removed behavior cloning script
- added `pretrain` method to base class, in order to use behavior cloning on all models
- fixed `close()` method for DummyVecEnv.
- added support for Dict spaces in DummyVecEnv and SubprocVecEnv. (@AdamGleave)
- added support for arbitrary multiprocessing start methods and added a warning about SubprocVecEnv that are not thread-safe by default. (@AdamGleave)
- added support for Discrete actions for GAIL
- fixed deprecation warning for tf: replaces `tf.to_float()` by `tf.cast()`
- fixed bug in saving and loading ddpg model when using normalization of obs or returns (@tperol)
- changed DDPG default buffer size from 100 to 50000.
- fixed a bug in `ddpg.py` in `combined_stats` for eval. Computed mean on `eval_episode_rewards` and `eval_qs` (@keshaviyengar)
- fixed a bug in `setup.py` that would error on non-GPU systems without TensorFlow installed

1.31.5 Release 2.4.1 (2019-02-11)

Bug fixes and improvements

- fixed computation of training metrics in TRPO and PPO1
- added `reset_num_timesteps` keyword when calling train() to continue tensorboard learning curves
- reduced the size taken by tensorboard logs (added a `full_tensorboard_log` to enable full logging, which was the previous behavior)
- fixed image detection for tensorboard logging
- fixed ACKTR for recurrent policies
- fixed gym breaking changes
- fixed custom policy examples in the doc for DQN and DDPG
- remove gym spaces patch for equality functions
- fixed tensorflow dependency: cpu version was installed overwritting tensorflow-gpu when present.
- fixed a bug in `traj_segment_generator` (used in ppo1 and trpo) where `new` was not updated. (spotted by @junhyeokahn)

1.31.6 Release 2.4.0 (2019-01-17)

Soft Actor-Critic (SAC) and policy kwargs

- added Soft Actor-Critic (SAC) model
- fixed a bug in DQN where prioritized_replay_beta_iters param was not used
- fixed DDPG that did not save target network parameters
- fixed bug related to shape of true_reward (@abhiskk)
- fixed example code in documentation of tf_util:Function (@JohannesAck)
- added learning rate schedule for SAC
- fixed action probability for continuous actions with actor-critic models
- added optional parameter to action_probability for likelihood calculation of given action being taken.
- added more flexible custom LSTM policies
- added auto entropy coefficient optimization for SAC
- clip continuous actions at test time too for all algorithms (except SAC/DDPG where it is not needed)
- added a mean to pass kwargs to policy when creating a model (+ save those kwargs)
- fixed DQN examples in DQN folder
- added possibility to pass activation function for DDPG, DQN and SAC

1.31.7 Release 2.3.0 (2018-12-05)

- added support for storing model in file like object. (thanks to @erniejunior)
- fixed wrong image detection when using tensorboard logging with DQN
- fixed bug in ppo2 when passing non callable lr after loading

- fixed tensorboard logging in ppo2 when nminibatches=1
- added early stoppping via callback return value (@erniejunior)
- added more flexible custom mlp policies (@erniejunior)

1.31.8 Release 2.2.1 (2018-11-18)

- added VecVideoRecorder to record mp4 videos from environment.

1.31.9 Release 2.2.0 (2018-11-07)

- Hotfix for ppo2, the wrong placeholder was used for the value function

1.31.10 Release 2.1.2 (2018-11-06)

- added `async_eigen_decomp` parameter for ACKTR and set it to `False` by default (remove deprecation warnings)
- added methods for calling env methods/setting attributes inside a VecEnv (thanks to @bjmuld)
- updated gym minimum version

1.31.11 Release 2.1.1 (2018-10-20)

- fixed MpiAdam synchronization issue in PPO1 (thanks to @brendenpetersen) issue #50
- fixed dependency issues (new mujoco-py requires a mujoco licence + gym broke MultiDiscrete space shape)

1.31.12 Release 2.1.0 (2018-10-2)

Warning: This version contains breaking changes for DQN policies, please read the full details

Bug fixes + doc update

- added patch fix for equal function using *gym.spaces.MultiDiscrete* and *gym.spaces.MultiBinary*
- fixes for DQN action_probability
- re-added double DQN + refactored DQN policies **breaking changes**
- replaced *async* with *async_eigen_decomp* in ACKTR/KFAC for python 3.7 compatibility
- removed action clipping for prediction of continuous actions (see issue #36)
- fixed NaN issue due to clipping the continuous action in the wrong place (issue #36)
- documentation was updated (policy + DDPG example hyperparameters)

1.31.13 Release 2.0.0 (2018-09-18)

Warning: This version contains breaking changes, please read the full details

Tensorboard, refactoring and bug fixes

- Renamed DeepQ to DQN **breaking changes**
- Renamed DeepQPolicy to DQNPolicy **breaking changes**
- fixed DDPG behavior **breaking changes**
- changed default policies for DDPG, so that DDPG now works correctly **breaking changes**
- added more documentation (some modules from common).
- added doc about using custom env
- added Tensorboard support for A2C, ACER, ACKTR, DDPG, DeepQ, PPO1, PPO2 and TRPO
- added episode reward to Tensorboard
- added documentation for Tensorboard usage
- added Identity for Box action space
- fixed render function ignoring parameters when using wrapped environments
- fixed PPO1 and TRPO done values for recurrent policies
- fixed image normalization not occurring when using images
- updated VecEnv objects for the new Gym version
- added test for DDPG
- refactored DQN policies
- added registry for policies, can be passed as string to the agent
- added documentation for custom policies + policy registration
- fixed numpy warning when using DDPG Memory
- fixed DummyVecEnv not copying the observation array when stepping and resetting
- added pre-built docker images + installation instructions
- added `deterministic` argument in the predict function
- added assert in PPO2 for recurrent policies
- fixed predict function to handle both vectorized and unwrapped environment
- added input check to the predict function
- refactored ActorCritic models to reduce code duplication
- refactored Off Policy models (to begin HER and replay_buffer refactoring)
- added tests for auto vectorization detection
- fixed render function, to handle positional arguments

1.31.14 Release 1.0.7 (2018-08-29)

Bug fixes and documentation

- added html documentation using sphinx + integration with read the docs
- cleaned up README + typos
- fixed normalization for DQN with images
- fixed DQN identity test

1.31.15 Release 1.0.1 (2018-08-20)

Refactored Stable Baselines

- refactored A2C, ACER, ACTKR, DDPG, DeepQ, GAIL, TRPO, PPO1 and PPO2 under a single constant class
- added callback to refactored algorithm training
- added saving and loading to refactored algorithms
- refactored ACER, DDPG, GAIL, PPO1 and TRPO to fit with A2C, PPO2 and ACKTR policies
- added new policies for most algorithms (Mlp, MlpLstm, MlpLnLstm, Cnn, CnnLstm and CnnLnLstm)
- added dynamic environment switching (so continual RL learning is now feasible)
- added prediction from observation and action probability from observation for all the algorithms
- fixed graphs issues, so models wont collide in names
- fixed behavior_clone weight loading for GAIL
- fixed Tensorflow using all the GPU VRAM
- fixed models so that they are all compatible with vectorized environments
- fixed `set_global_seed` to update `gym.spaces`'s random seed
- fixed PPO1 and TRPO performance issues when learning identity function
- added new tests for loading, saving, continuous actions and learning the identity function
- fixed DQN wrapping for atari
- added saving and loading for Vecnormalize wrapper
- added automatic detection of action space (for the policy network)
- fixed ACER buffer with constant values assuming n_stack=4
- fixed some RL algorithms not clipping the action to be in the action_space, when using `gym.spaces.Box`
- refactored algorithms can take either a `gym.Environment` or a `str` ([if the environment name is registered](https://github.com/openai/gym/wiki/Environments))
- Hoftix in ACER (compared to v1.0.0)

Future Work :

- Finish refactoring HER
- Refactor ACKTR and ACER for continuous implementation

1.31.16 Release 0.1.6 (2018-07-27)

Deobfuscation of the code base + pep8 and fixes

- Fixed `tf.session().__enter__()` being used, rather than `sess = tf.session()` and passing the session to the objects
- Fixed uneven scoping of TensorFlow Sessions throughout the code
- Fixed rolling vecwrapper to handle observations that are not only grayscale images
- Fixed deepq saving the environment when trying to save itself
- Fixed `ValueError: Cannot take the length of Shape with unknown rank.` in `acktr`, when running `run_atari.py` script.
- Fixed calling baselines sequentially no longer creates graph conflicts
- Fixed mean on empty array warning with deepq
- Fixed kfac eigen decomposition not cast to float64, when the parameter use_float64 is set to True
- Fixed Dataset data loader, not correctly resetting id position if shuffling is disabled
- Fixed `EOFError` when reading from connection in the `worker` in `subproc_vec_env.py`
- Fixed `behavior_clone` weight loading and saving for GAIL
- Avoid taking root square of negative number in `trpo_mpi.py`
- Removed some duplicated code (a2cpolicy, trpo_mpi)
- Removed unused, undocumented and crashing function `reset_task` in `subproc_vec_env.py`
- Reformated code to PEP8 style
- Documented all the codebase
- Added atari tests
- Added logger tests

Missing: tests for acktr continuous (+ HER, rely on mujoco...)

1.31.17 Maintainers

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

1.31.18 Contributors (since v2.0.0):

In random order...

Thanks to @bjmuld @iambenzo @iandanforth @r7vme @brendenpetersen @huvar @abhiskk @JohannesAck @EliasHasle @mrakgr @Bleyddyn @antoine-galataud @junhyeokahn @AdamGleave @keshaviyengar @tperol @XMaster96 @kantneel @Pastafarianist @GerardMaggiolino @PatrickWalter214 @yutingsz @sc420 @Aaahh @billtubbs @Miffyli @dwiel @miguelrass @qxcv

1.32 Projects

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

1.32.1 Learning to drive in a day

Implementation of reinforcement learning approach to make a donkey car learn to drive. Uses DDPG on VAE features (reproducing paper from wayve.ai)

Author: Roma Sokolkov (@r7vme)
Github repo: https://github.com/r7vme/learning-to-drive-in-a-day

1.32.2 Donkey Gym

OpenAI gym environment for donkeycar simulator.

Author: Tawn Kramer (@tawnkramer)
Github repo: https://github.com/tawnkramer/donkey_gym

1.32.3 Self-driving FZERO Artificial Intelligence

Series of videos on how to make a self-driving FZERO artificial intelligence using reinforcement learning algorithms PPO2 and A2C.

Author: Lucas Thompson
Video Link

1.32.4 S-RL Toolbox

S-RL Toolbox: Reinforcement Learning (RL) and State Representation Learning (SRL) for Robotics. Stable-Baselines was originally developped for this project.

Authors: Antonin Raffin, Ashley Hill, René Traoré, Timothée Lesort, Natalia Díaz-Rodríguez, David Filliat
Github repo: https://github.com/araffin/robotics-rl-srl

1.32.5 Roboschool simulations training on Amazon SageMaker

"In this notebook example, we will make HalfCheetah learn to walk using the stable-baselines [. . .]"

Author: Amazon AWS
Repo Link

1.32.6 MarathonEnvs + OpenAi.Baselines

Experimental - using OpenAI baselines with MarathonEnvs (ML-Agents)

Author: Joe Booth (@Sohojoe)
Github repo: https://github.com/Sohojoe/MarathonEnvsBaselines

1.32.7 Learning to drive smoothly in minutes

Implementation of reinforcement learning approach to make a car learn to drive smoothly in minutes. Uses SAC on VAE features.

Author: Antonin Raffin (@araffin)
Blog post: https://towardsdatascience.com/learning-to-drive-smoothly-in-minutes-450a7cdb35f4
Github repo: https://github.com/araffin/learning-to-drive-in-5-minutes

1.32.8 Making Roboy move with elegance

Project around Roboy, a tendon-driven robot, that enabled it to move its shoulder in simulation to reach a pre-defined point in 3D space. The agent used Proximal Policy Optimization (PPO) or Soft Actor-Critic (SAC) and was tested on the real hardware.

Authors: Alexander Pakakis, Baris Yazici, Tomas Ruiz
Email: FirstName.LastName@tum.de
GitHub repo: https://github.com/Roboy/DeepAndReinforced
DockerHub image: deepandreinforced/rl:latest
Presentation: https://tinyurl.com/DeepRoboyControl
Video: https://tinyurl.com/DeepRoboyControlVideo
Blog post: https://tinyurl.com/mediumDRC
Website: https://roboy.org/

1.32.9 Train a ROS-integrated mobile robot (differential drive) to avoid dynamic objects

The RL-agent serves as local planner and is trained in a simulator, fusion of the Flatland Simulator and the crowd simulator Pedsim. This was tested on a real mobile robot. The Proximal Policy Optimization (PPO) algorithm is applied.

Author: Ronja Güldenring
Email: 6guelden@informatik.uni-hamburg.de
Video: https://www.youtube.com/watch?v=laGrLaMaeT4
GitHub: https://github.com/RGring/drl_local_planner_ros_stable_baselines

1.32.10 Adversarial Policies: Attacking Deep Reinforcement Learning

Uses Stable Baselines to train *adversarial policies* that attack pre-trained victim policies in a zero-sum multi-agent environments. May be useful as an example of how to integrate Stable Baselines with Ray to perform distributed experiments and Sacred for experiment configuration and monitoring.

Authors: Adam Gleave, Michael Dennis, Neel Kant, Cody Wild
Email: adam@gleave.me
GitHub: https://github.com/HumanCompatibleAI/adversarial-policies
Paper: https://arxiv.org/abs/1905.10615
Website: https://adversarialpolicies.github.io

1.33 Plotting Results

stable_baselines.results_plotter.**main**()
Example usage in jupyter-notebook

```
from stable_baselines import results_plotter
%matplotlib inline
results_plotter.plot_results(["./log"], 10e6, log_viewer.X_TIMESTEPS, "Breakout")
```

Here ./log is a directory containing the monitor.csv files

stable_baselines.results_plotter.**plot_curves**(*xy_list*, *xaxis*, *title*)
plot the curves

Parameters

- **xy_list** – ([(np.ndarray, np.ndarray)]) the x and y coordinates to plot
- **xaxis** – (str) the axis for the x and y output (can be X_TIMESTEPS='timesteps', X_EPISODES='episodes' or X_WALLTIME='walltime_hrs')
- **title** – (str) the title of the plot

stable_baselines.results_plotter.**plot_results**(*dirs*, *num_timesteps*, *xaxis*, *task_name*)
plot the results

Parameters

- **dirs** – ([str]) the save location of the results to plot
- **num_timesteps** – (int or None) only plot the points below this value
- **xaxis** – (str) the axis for the x and y output (can be X_TIMESTEPS='timesteps', X_EPISODES='episodes' or X_WALLTIME='walltime_hrs')
- **task_name** – (str) the title of the task to plot

stable_baselines.results_plotter.**rolling_window**(*array*, *window*)
apply a rolling window to a np.ndarray

Parameters

- **array** – (np.ndarray) the input Array
- **window** – (int) length of the rolling window

Returns (np.ndarray) rolling window on the input array

stable_baselines.results_plotter.**ts2xy**(*timesteps*, *xaxis*)
Decompose a timesteps variable to x ans ys

Parameters

- **timesteps** – (Pandas DataFrame) the input data

- **xaxis** – (str) the axis for the x and y output (can be X_TIMESTEPS='timesteps', X_EPISODES='episodes' or X_WALLTIME='walltime_hrs')

Returns (np.ndarray, np.ndarray) the x and y output

stable_baselines.results_plotter.**window_func**(*var_1*, *var_2*, *window*, *func*)

apply a function to the rolling window of 2 arrays

Parameters

- **var_1** – (np.ndarray) variable 1
- **var_2** – (np.ndarray) variable 2
- **window** – (int) length of the rolling window
- **func** – (numpy function) function to apply on the rolling window on variable 2 (such as np.mean)

Returns (np.ndarray, np.ndarray) the rolling output with applied function

CHAPTER 2

Citing Stable Baselines

To cite this project in publications:

```
@misc{stable-baselines,
  author = {Hill, Ashley and Raffin, Antonin and Ernestus, Maximilian and Gleave,␣
↪Adam and Traore, Rene and Dhariwal, Prafulla and Hesse, Christopher and Klimov,␣
↪Oleg and Nichol, Alex and Plappert, Matthias and Radford, Alec and Schulman, John␣
↪and Sidor, Szymon and Wu, Yuhuai},
  title = {Stable Baselines},
  year = {2018},
  publisher = {GitHub},
  journal = {GitHub repository},
  howpublished = {\url{https://github.com/hill-a/stable-baselines}},
}
```

CHAPTER 3

Contributing

To any interested in making the rl baselines better, there is still some improvements that needs to be done. A full TODO list is available in the roadmap.

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

CHAPTER 4

Indices and tables

- genindex
- search
- modindex

Python Module Index

S

Index

A

G

H

I

K

L

M

N

O

P

R

S

T

V

W