Stable Baselines Documentation

Release 2.5.0

Stable Baselines Contributors

May 04, 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 and evaluate agents.

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

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 follwing:

```
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
- 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	✓	✓	[5]	✓	✓
ACKTR	✓	✓	[5]	✓	✓
DDPG	✓		✓		
DQN	✓			✓	
GAIL[2]	✓	✓	✓	✓	✓[4]
PPO1	✓		✓	✓	✓[4]
PPO2	✓	✓	✓	✓	✓
SAC	✓		✓		
TRPO	✓		✓	✓	✓[4]

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

Actions `gym.spaces`:

- `Box`: A N-dimensional box that 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.

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)

1.5.2 Basic Usage: Training, Saving, Loading

In the following example, we will train, save and load an A2C model on the Lunar Lander environment.

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

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`

```
import gym

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

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

model = A2C(MlpPolicy, env, ent_coef=0.1, verbose=1)
# Train the agent
model.learn(total_timesteps=100000)
# Save the agent
model.save("a2c_lunar")
del model  # delete trained model to demonstrate loading

# Load the trained agent
model = A2C.load("a2c_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

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
```

(continues on next page)

(continued from previous page)

```
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.


```
import os

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

from stable_baselines.ddpg.policies import MlpPolicy
from stable_baselines.common.vec_env.dummy_vec_env import DummyVecEnv
from stable_baselines.bench import Monitor
from stable_baselines.results_plotter import load_results, ts2xy
from stable_baselines import DDPG
```

(continues on next page)

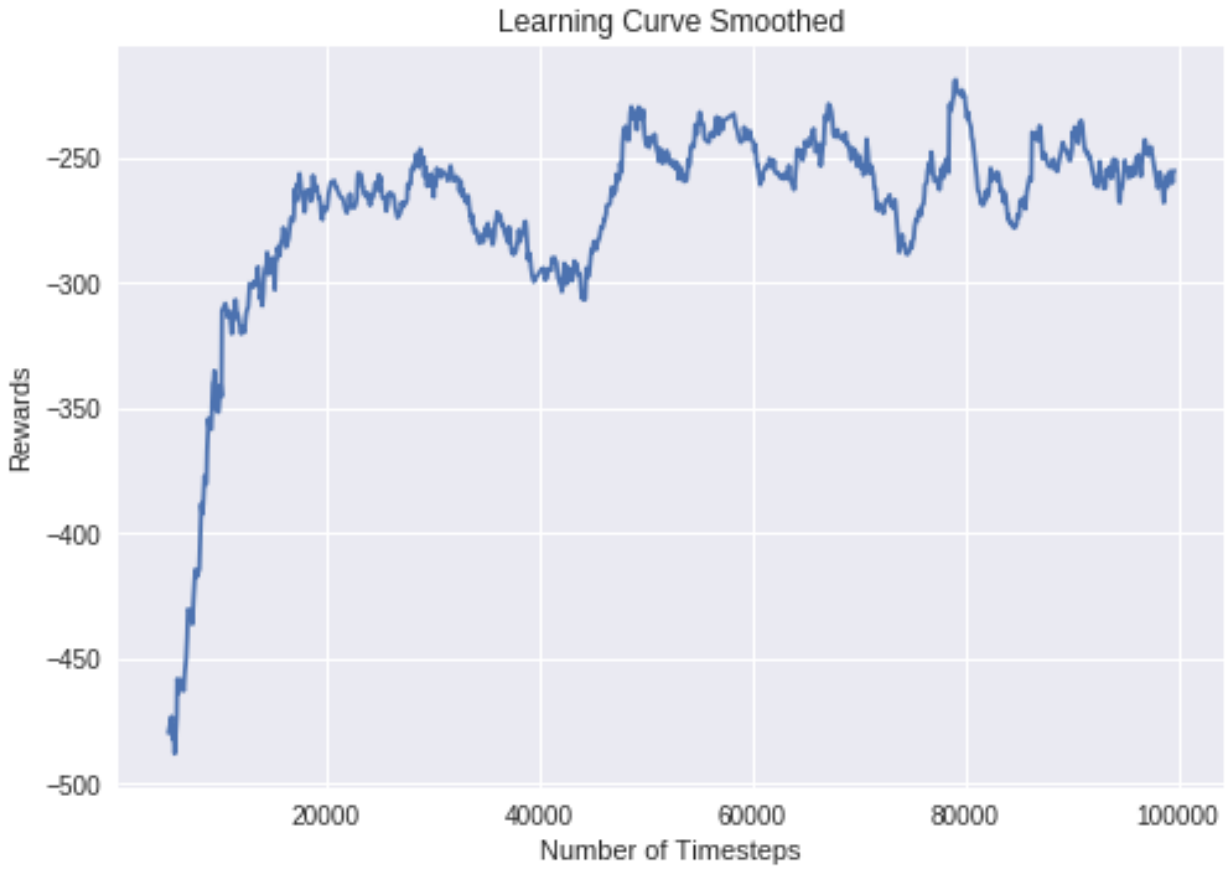

Fig. 4: Learning curve of DDPG on LunarLanderContinuous environment

(continued from previous page)

```
from stable_baselines.ddpg.noise 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)
  :param _globals: (dict)
  """
  global n_steps, best_mean_reward
  # Print stats every 1000 calls
  if (n_steps + 1) % 1000 == 0:
      # Evaluate policy 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)
env = DummyVecEnv([lambda: env])

# Add some param noise for exploration
param_noise = AdaptiveParamNoiseSpec(initial_stddev=0.2, desired_action_stddev=0.2)
model = DDPG(MlpPolicy, env, param_noise=param_noise, memory_limit=int(1e6),␣
↪verbose=0)
# Train the agent
model.learn(total_timesteps=200000, callback=callback)
```

1.5.5 Atari Games

Fig. 5: Trained A2C agent on Breakout

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.

Fig. 6: Pong Environment

Try it in a CO notebook

```
from stable_baselines.common.cmd_util import make_atari_env
from stable_baselines.common.policies import CnnPolicy
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/"
```

(continues on next page)

(continued from previous page)

```
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")

# 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)
```

1.5.8 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.common.policies import CnnPolicy
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)

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
```

(continues on next page)

(continued from previous page)

```
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()
```

1.5.9 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()
```

1.5.10 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
```

(continues on next page)

(continued from previous page)

```
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 multiprocess training. Instead of training an RL agent on 1 environment, it allows us to train it on *n* environments using *n* processes. 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.

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 DummyVecEnv

class stable_baselines.common.vec_env.**DummyVecEnv**(*env_fns*)

Creates a simple vectorized wrapper for multiple environments

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

close()

Clean up the environment's resources.

env_method(*method_name*, **method_args*, ***method_kwargs*)
Provides an interface to call arbitrary class methods of vectorized environments

Parameters

- **method_name** – (str) The name of the env class method to invoke
- **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 retured by the environment's method call

get_attr(*attr_name*)
Provides a mechanism for getting class attribues from vectorized environments

Parameters **attr_name** – (str) The name of the attribute whose value to return

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

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*)
Provides a mechanism for setting arbitrary class attributes inside vectorized environments

Parameters

- **attr_name** – (str) 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 (list) in case env access methods might return something, they will be returned in a list

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 SubprocVecEnv

class stable_baselines.common.vec_env.**SubprocVecEnv**(*env_fns*, *start_method=None*)
Creates a multiprocess vectorized wrapper for multiple environments

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*, ***method_kwargs*)
Provides an interface to call arbitrary class methods of vectorized environments

Parameters

- **method_name** – (str) The name of the env class method to invoke
- **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 retured by each environment's method call

get_attr(*attr_name*)
Provides a mechanism for getting class attribues from vectorized environments (note: attribute value returned must be picklable)

Parameters **attr_name** – (str) The name of the attribute whose value to return

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

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*)
Provides a mechanism for setting arbitrary class attributes inside vectorized environments (note: this is a broadcast of a single value to all instances) (note: the value must be picklable)

Parameters

- **attr_name** – (str) Name of attribute to assign new value
- **value** – (obj) Value to assign to 'attr_name'

- **indices** – (list,tuple) Iterable containing indices of envs whose attr to set

Returns (list) in case env access methods might return something, they will be returned in a list

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 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

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):

```
import gym
from gym import spaces

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

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

  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
```

(continues on next page)

(continued from previous page)

```
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
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
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>
                     |
                   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)
```

(continues on next page)

(continued from previous page)

```
            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
            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.initial_state = None
        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, self.neglogp],
                                                   {self.obs_ph: obs})
        else:
            action, value, neglogp = self.sess.run([self.action, self._value, 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, {self.obs_ph: obs})


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

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.common.policies import MlpPolicy
```

(continues on next page)

(continued from previous page)

```
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(MlpPolicy, env, 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.common.vec_env import DummyVecEnv
from stable_baselines.common.policies import MlpPolicy
from stable_baselines import A2C

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

model = A2C(MlpPolicy, env, verbose=1, tensorboard_log="./a2c_cartpole_tensorboard/")
model.learn(total_timesteps=10000, tb_log_name="first_run")
model.learn(total_timesteps=10000, tb_log_name="second_run")
model.learn(total_timesteps=10000, tb_log_name="thrid_run")
```

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 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
```

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 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>=2.2.1 box2d box2d-kengz pyyaml pybullet==2.1.0␣
↪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
```

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
```

(continues on next page)

(continued from previous page)

```
      # 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
# 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()
```

(continues on next page)

(continued from previous page)

```
    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*, *train_fraction=0.7*, *batch_size=64*, *traj_limitation=-1*, *randomize=True*, *verbose=1*, *sequential_preprocessing=False*)

Dataset for using behavior cloning or GAIL.

Data structure of the expert dataset, an ".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.

Parameters

- **expert_path** – (str) the path to trajectory data (.npz file)
- **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*, *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')
- **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.

1.12 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*)

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.

> **Warning:** When working with continuous probability distribution (e.g. Gaussian distribution for continuous action) the probability of taking a particular action is exactly zero. 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)

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

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

Returns (Gym Environment) The current environment

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

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

1.13 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 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

1.13.1 Base Classes

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

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.**FeedForwardPolicy**(*sess*, *ob_space*, *ac_space*, *n_env*, *n_steps*, *n_batch*, *reuse=False*, *layers=None*, *net_arch=None*, *act_fun=<MagicMock id='140384425661776'>*, *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='140384425598032'>*, *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*)

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

1.13.2 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

1.13.3 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

1.14 A2C

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

1.14.1 Notes

- Original paper: https://arxiv.org/abs/1602.01783
- 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.

1.14.2 Can I use?

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

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

1.14.3 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")

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

1.14.4 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 caculation
- **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*)

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.

Warning: When working with continuous probability distribution (e.g. Gaussian distribution for continuous action) the probability of taking a particular action is exactly zero. 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)

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

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

Returns (Gym Environment) The current environment

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

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

1.15 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.

1.15.1 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.

1.15.2 Can I use?

- Recurrent policies: ✓
- Multi processing: ✓

- Gym spaces:

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

1.15.3 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)])

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()
```

1.15.4 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*)

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.

> **Warning:** When working with continuous probability distribution (e.g. Gaussian distribution for continuous action) the probability of taking a particular action is exactly zero. 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)

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

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

Returns (Gym Environment) The current environment

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

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 ACKTR

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

1.16.1 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.

1.16.2 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 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()
```

1.16.4 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 leiber
- **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*)

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.

> **Warning:** When working with continuous probability distribution (e.g. Gaussian distribution for continuous action) the probability of taking a particular action is exactly zero. 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)

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

get_env()

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

Returns (Gym Environment) The current environment

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

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 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

1.17.1 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.

1.17.2 Can I use?

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

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

1.17.3 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()
```

1.17.4 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=50000*, *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** – (Memory) the replay buffer (if None, default to baselines.ddpg.memory.Memory)
- **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_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
- **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*)

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.

Warning: When working with continuous probability distribution (e.g. Gaussian distribution for continuous action) the probability of taking a particular action is exactly zero. 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)

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

get_env()

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

Returns (Gym Environment) The current environment

learn(*total_timesteps*, *callback=None*, *seed=None*, *log_interval=100*, *tb_log_name='DDPG'*, *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)

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

1.17.5 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

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

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*)
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

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

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*)
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

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

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*)
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

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

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*)
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

1.17.6 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

1.17.7 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)
```

1.18 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

1.18.1 Notes

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

1.18.2 Can I use?

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

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

1.18.3 Example

```
import gym

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)
```

(continues on next page)

(continued from previous page)

```
model.learn(total_timesteps=25000)
model.save("deepq_cartpole")

del model # remove to demonstrate saving and loading

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

DQN.load("deepq_breakout")

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.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*)

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.

> **Warning:** When working with continuous probability distribution (e.g. Gaussian distribution for continuous action) the probability of taking a particular action is exactly zero. 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)

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

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

Returns (Gym Environment) The current environment

learn(*total_timesteps*, *callback=None*, *seed=None*, *log_interval=100*, *tb_log_name='DQN'*, *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)

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 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

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

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

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

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

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

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

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

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

1.18.6 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])

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

1.19 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.

1.19.1 Notes

- Original paper: https://arxiv.org/abs/1606.03476

Warning: Images are not yet handled properly by the current implementation

1.19.2 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

1.19.3 Can I use?

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

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

1.19.4 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)
```

(continues on next page)

(continued from previous page)

```
# 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()
```

1.19.5 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 leiber loss threashold
- **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*)

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.

Warning: When working with continuous probability distribution (e.g. Gaussian distribution for continuous action) the probability of taking a particular action is exactly zero. 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)

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

get_env()

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

Returns (Gym Environment) The current environment

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

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

1.20 HER

Hindsight Experience Replay (HER)

Warning: HER is not refactored yet. We are looking for contributors to help us.

1.20.1 How to use Hindsight Experience Replay

Getting started

Training an agent is very simple:

```
python -m stable_baselines.her.experiment.train
```

This will train a DDPG+HER agent on the `FetchReach` environment. You should see the success rate go up quickly to `1.0`, which means that the agent achieves the desired goal in 100% of the cases. The training script logs other diagnostics as well and pickles the best policy so far (w.r.t. to its test success rate), the latest policy, and, if enabled, a history of policies every K epochs.

To inspect what the agent has learned, use the play script:

```
python -m stable_baselines.her.experiment.play /path/to/an/experiment/policy_best.pkl
```

You can try it right now with the results of the training step (the script prints out the path for you). This should visualize the current policy for 10 episodes and will also print statistics.

Reproducing results

In order to reproduce the results from Plappert et al. (2018), run the following command:

```
python -m stable_baselines.her.experiment.train --num_cpu 19
```

This will require a machine with sufficient amount of physical CPU cores. In our experiments, we used Azure's D15v2 instances, which have 20 physical cores. We only scheduled the experiment on 19 of those to leave some head-room on the system.

1.20.2 Parameters

class stable_baselines.her.**HER**(*policy*, *env*, *verbose=0*, *_init_setup_model=True*)

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

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.

> **Warning:** When working with continuous probability distribution (e.g. Gaussian distribution for continuous action) the probability of taking a particular action is exactly zero. 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)

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

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=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)

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

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

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

1.21 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.

1.21.1 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`

1.21.2 Can I use?

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

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

1.21.3 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()
```

1.21.4 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*)

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.

> **Warning:** When working with continuous probability distribution (e.g. Gaussian distribution for continuous action) the probability of taking a particular action is exactly zero. 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)

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

get_env()

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

Returns (Gym Environment) The current environment

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

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

1.22 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.

1.22.1 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/
- **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.

1.22.2 Can I use?

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

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

1.22.3 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()
```

1.22.4 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*, *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 caculation
- **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
- **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*)

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.

> **Warning:** When working with continuous probability distribution (e.g. Gaussian distribution for continuous action) the probability of taking a particular action is exactly zero. 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)

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

get_env()

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

Returns (Gym Environment) The current environment

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

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 SAC

Soft Actor Critic (SAC) Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor.

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

1.23.1 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

1.23.2 Can I use?

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

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

1.23.3 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()
```

1.23.4 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'*, *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')
- **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*)

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.

> **Warning:** When working with continuous probability distribution (e.g. Gaussian distribution for continuous action) the probability of taking a particular action is exactly zero. 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)

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

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

Returns (Gym Environment) The current environment

learn(*total_timesteps*, *callback=None*, *seed=None*, *log_interval=4*, *tb_log_name='SAC'*, *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)

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

1.23.5 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

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

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])

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

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

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])

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

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

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])

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

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

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])

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

1.23.6 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)
```

1.24 TRPO

Trust Region Policy Optimization (TRPO) is an iterative approach for optimizing policies with guaranteed monotonic improvement.

1.24.1 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.

1.24.2 Can I use?

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

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

1.24.3 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)
    obs, rewards, dones, info = env.step(action)
    env.render()
```

1.24.4 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 leiber 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*)

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.

> **Warning:** When working with continuous probability distribution (e.g. Gaussian distribution for continuous action) the probability of taking a particular action is exactly zero. 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)

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

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

Returns (Gym Environment) The current environment

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

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

1.25 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-Leiber 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-Leiber 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-Leiber 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-Leiber 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-Leiber 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

1.26 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='140384425360744'>*, *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_available_gpus**()
Return a list of all the available GPUs

Returns ([str]) the GPUs available

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

1.27 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)}

1.28 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)

1.29 Changelog

For download links, please look at Github release page.

1.29.1 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

1.29.2 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)

1.29.3 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

1.29.4 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)

1.29.5 Release 2.2.1 (2018-11-18)

- added VecVideoRecorder to record mp4 videos from environment.

1.29.6 Release 2.2.0 (2018-11-07)

- Hotfix for ppo2, the wrong placeholder was used for the value function

1.29.7 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

1.29.8 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)

1.29.9 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)

1.29.10 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

1.29.11 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

1.29.12 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

1.29.13 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...)

1.29.14 Maintainers

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

1.29.15 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

1.30 Projects

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

1.30.1 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

1.30.2 Donkey Gym

OpenAI gym environment for donkeycar simulator.

Author: Tawn Kramer (@tawnkramer)
Github repo: https://github.com/tawnkramer/donkey_gym

1.30.3 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

1.30.4 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

1.30.5 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

1.30.6 MarathonEnvs + OpenAi.Baselines

Experimental - using OpenAI baselines with MarathonEnvs (ML-Agents)

Author: Joe Booth (@Sohojoe)
Github repo: https://github.com/Sohojoe/MarathonEnvsBaselines

1.30.7 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

1.31 Plotting Results

stable_baselines.results_plotter.**main**()
Example usage in jupyter-notebook

```
from stable_baselines import log_viewer
%matplotlib inline
log_viewer.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) 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

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

V

W