SafeRLBench Documentation

Release 0.1.0

Nicolas Ochsner

Apr 18, 2018

Contents

SafeRLBench provides an interface for algorithms, environments and policies, to support a reusable benchmark environment.

1 Overview

PyPI: pypi.python.org/pypi/SafeRLBench
Repository: github.com/befelix/Safe-RL-Benchmark
Documentation: saferlbench.readthedocs.io

2 Structure

The main module contains base classes that define an interface and benchmark facilities that are used to run and compare algorithms. Further the library contains three submodules that contain content using respective base classes.

Algorithm module `algo` Contains algorithm implementations like `PolicyGradient` or `SafeOpt`. Classes in this module are subclasses of the `AlgorithmBase` class.

Environment module `envs` Contains environment implementations like `LinearCar` or `Quadrocopter` environments. These are subclasses of the `EnvironmentBase` class.

Policy module `policy` Contains policies. Although some policies are specific for the use with certain algorithms, they are still separated in an individual module, providing a interface as defined through the `Policy` base class, since in there are cases in which they can be optimized through different algorithms.

3 Installation

3.1 Dependencies

SafeRLBench requires:

- NumPy >= 1.7
- SciPy >= 0.19.0
- six >= 1.10
- futures >= 3.0.5 for python 2.7

3.2 Pip

The package is available on PyPi, which means it can easily be installed using pip.

```
pip install SafeRLBench
```

3.3 Clone

The best way to install and use this library is to clone or fork it from the repository.

```
git clone https://github.com/befelix/Safe-RL-Benchmark.git
```

To use the content that has already been implemented as is, navigate into the root directory and execute:

```
python setup.py install
```

In many cases it makes sense to extend or adapt the content. Then the develop setup is your friend. Again, navigate to the root directory of the repository and execute:

```
python setup.py develop
```

4 Getting Started

The following instructions can be executed in many ways. You may use your favorite interactive interpreter, include it in scripts or use some form of notebook to get started.

In the examples directory one may find a notebook containing the examples described below.

4.1 Optimizing a Policy

To get started we will try and optimize a policy on a very simple environment. To accomplish this we need to make a few decisions. First we need some task to solve. This is implemented in the form of environments in the `envs` module.

```
>>> # import the linear car class
>>> from SafeRLBench.envs import LinearCar
>>> # get an instance with the default arguments
>>> linear_car = LinearCar()
```

Ok, so far so good. Next we need a policy. Again, before anything gets too complicated, let us take linear mapping. Fortunately there is a linear mapping implemented in the `policy` module.

```
>>> # import the linear policy class
>>> from SafeRLBench.policy import LinearPolicy
>>> # instantiate it with d_state=2 and d_action=1
>>> policy = LinearPolicy(2, 1)
>>> # setup some initial parameters
>>> policy.parameters = [1, 1, 1]
```

Notice that we did not use the default parameters this time. The LinearPolicy is a linear mapping from an element of a d_state-dimensional space to a d_action-dimensional space. Our `linear_car` instance with the default arguments is just a car with a (position, velocity)-state on a line, thus our state space is two dimensional and we can accelerate along the line, so our action space is one dimensional.

Now we need our third and last ingredient, which is the algorithm that optimizes the policy on the environment. On this environment `PolicyGradient` with central differences gradient estimator proved to be a very stable algorithm.

```
>>> # import the policy gradient class
>>> from SafeRLBench.algo import PolicyGradient
>>> # instantiate it with the environment and algorithm
>>> optimizer = PolicyGradient(linear_car, policy, estimator='central_fd')
```

Earlier we set some initial parameters. The *PolicyGradient* optimizer will check if there are initial parameters and use those if present. If there are no parameters set he will randomly initialize them, until he finds a nonzero gradient.

```
>>> # optimize the policy when everything is set up.
>>> optimizer.optimize()
```

Now the algorithm might run for a while depending on how much effort the optimization takes. Unfortunately no information on the progress shows up, yet. We will deal with that in the next part.

Lets take a look at what actually happened during the run. For this we can access the *monitor* and generate some plots. For example, we could plot the reward evolution during optimization.

```
>>> # use matplotlib for plotting
>>> import matplotlib.pyplot as plt
>>> # retrieve the rewards
>>> y = optimizer.monitor.rewards
>>> plt.plot(range(len(y)), y)
>>> plt.show()
```

4.2 Configuration

Especially when you try to set up a new environment it is often very useful to get some logging information. In *SafeRLBench* there is an easy way to setup some global configurations. Let us access the global *config* variable:

```
>>> # import the config variable
>>> from SafeRLBench import config
```

Well, thats it. The *config* variable is an instance of the class *SRBConfig*, which contains methods to manipulate the overall behavior. For example we can easily make the logger print to stdout:

```
>>> # output to stdout
>>> config.logger_add_stream_handler()
```

Or we might want to change the level of the logger:

```
>>> # print debug information
>>> config.logger_set_level(config.DEBUG)
```

There are some more tricks and tweaks to it, which can be found directly in the class documentation. For example we can directly assign a handler or we can add an additional file handler that writes our output to a file, etc. For more information on that refer to the documentation.

In general the class methods and attributes will follow the a naming convention, that is, the first part of the name will regard the part we want to configure and the second part will describe what we want to change.

Apart from the logger, let's say we want to change the amount of jobs that are used by the benchmarking facility. (We will see it in the next section.) Simply configure it with:

```
>>> # set number of jobs to 4
>>> config.jobs_set(4)
```

Or set the verbosity level of the monitor:

```
>>> # increase verbosity to 2
>>> config.monitor_set_verbosity(2)
```

4.3 Benchmarking

We can optimize policies on environments now, the next thing we want to do is benchmarking. For this we can use the benchmark facilities that the library provides. In order to run a benchmark, we need to produce an instance `BenchConfig`.

When we take a look at the documentation of this class, it takes two arguments. The first one is `algs` the second one `envs`. And now it gets a litte bit weird, both of them are a list of a list of tuples where the second element is a list of dictionaries. Confused? Yes, but here is a simple example:

```
>>> # define environment configuration.
>>> envs = [[(LinearCar, {'horizon': 100})]]
>>> # define algorithms configuration.
>>> algs = [[
...   (PolicyGradient, [{
...     'policy': LinearPolicy(2, 1, par=[-1, -1, 1]),
...     'estimator': 'central_fd',
...     'var': var
...   } for var in [1, 1.5, 2, 2.5]])
... ]]
```

So what happens? The outer most lists of envs and algs will get zipped, such that we can support pair wise configurations. Further, the tuple contains a class in the first element and a list of configurations dictionaries in the second element. This essentially allows quick generation of many configurations for a single algorithm or environment. Finally the cartesian product of **all** configurations in the inner lists will be executed by the `Bench`.

So in the example above, we only have a single environment configuration, but the corresponding list in `algs` contains four configurations for the `PolicyGradient`. Overall this will result in four test runs.

In case we had

```
>>> envs_two = [(LinearCar, {'horizon': 100}), (LinearCar, {'horizon': 200})]
```

`BenchConfig` would supply eight configurations to the `Bench`. By the way, if the outer list is not needed, it can safely be omitted.

```
>>> # import BenchConfig
>>> from SafeRLBench import BenchConfig
>>> # instantiate BenchConfig
>>> config = BenchConfig(algs, envs)
```

Next we can evaluate the configuration achieving the best performance. The library contains a tool for this, the measures.

```
>>> # import the best performance measure
>>> from SafeRLBench.measure import BestPerformance
>>> # import the Bench
>>> from SafeRLBench import Bench
>>> # instantiate the bench
>>> bench = Bench(config, BestPerformance())
```

It is also possible to avoid the config step and do it automatically with a bench factory.

```
>>> # create bench instance with constructor
>>> bench = Bench.make_bench(algs, envs, BestPerformance())
```

Either way, now the bench is ready to run. Calling the instance will first run and then evaluate the results.

```
>>> # run the benchmark
>>> bench()
```

The result of the evaluation is stored in the measure, which is stored in the measures field. `measures` is a list of all measure instances we passed and their result can be accessed through the `result` property.

```
>>> bench.measures[0]
<SafeRLBench.measure.BestPerformance at 0x1211307b8>
>>> best_run = bench.measures[0].result[0][0]
>>> monitor = best_run.get_alg_monitor()
>>> # extract the best trace
>>> best_trace = monitor.traces[monitor.rewards.index(max(monitor.rewards))]
>>> # plot the position of the best trace
>>> y = [t[1][0] for t in best_trace]
>>> x = range(len(y))
>>> plt.plot(x, y)
>>> plt.show()
```

4.4 Using SafeOpt

The last section of **Getting Started** involves optimization using *SafeOpt*. There is a notebook SafeOpt.ipynb in the examples directory containing the following and further examples.

To use *SafeOpt* additional requirements are needed: safeopt, GPy

In the following we want to use *SafeOpt* to safely optimize a controller for the quadrocopter environment. As always, we start by importing all the necessary tools:

```
>>> # GPy is needed to supply `safeopt` with a kernel
>>> import GPy
>>> # Algorithm, Environment and Controller
>>> from SafeRLBench.algo import SafeOptSwarm
>>> from SafeRLBench.envs import Quadrocopter
>>> from SafeRLBench.policy import NonLinearQuadrocopterController
>>> # Bench and Measures
>>> from SafeRLBench import Bench
>>> from SafeRLBench.measure import SafetyMeasure, BestPerformance
```

Unfortunately we can not use multiple jobs when optimizing with *SafeOpt*, because *GPy* does contain lambda expressions, which are not pickable. Let us make sure everything is configured properly.

```
>>> from SafeRLBench import config
>>> config.jobs_set(1)
>>> config.logger_add_stream_handler()
>>> config.logger_set_level(config.INFO)
>>> config.monitor_set_verbosity(2)
```

Now, with everything imported we are ready to define our test runs. For the environment, let us just take the default configuration of the quadrocopter:

```
>>> envs = [(Quadrocopter, {})]
```

And for the algorithm, let us try different values for the variance.

```
>>> noise_var = 0.05**2
>>> # the safety constraint on the performance, we do not want to drop below fmin.
>>> fmin = -2300
>>> # bounds for the possible controller parameters
>>> bounds = [(0., 1.), (0., 1.), (0., 1.), (0., 1.), (0., 1.)]
>>> algos = [
...   (SafeOptSwarm, [{
...     'policy': NonLinearQuadrocopterController(),
...     'kernel': GPy.kern.RBF(input_dim=len(bounds), variance=std**2, lengthscale=0.
↪2, ARD=True),
...     'likelihood': GPy.likelihoods.gaussian.Gaussian(variance=noise_var),
...     'max_it': 20,
...     'avg_reward': -1500,
...     'window': 3,
...     'fmin': fmin,
...     'bounds': bounds,
...     'swarm_size': 1000,
...     'info': std,
...   } for std in [1000, 1250, 1500, 1750, 2000]])]
```

Ok there are a lot of arguments here. The documentation contains descriptions for each of them. Here we will just observe what happens.

```
>>> # produce the bench, initialize the safety measure with fmin
>>> bench = Bench.make_bench(algos, envs, measures=[SafetyMeasure(fmin),␣
↪BestPerformance()])
>>> # start the run and evaluation
>>> bench()
```

After the run is finished we can observe what happened by analyzing the measures. This is a bit cumbersome at the moment, but will potentially be improved in the future with some additional convenience methods. Anyways, the evaluation of the *SafetyMeasure* could be accessed as follows.

```
>>> # (std, number of violations, amount of violations)
>>> [(t[0].alg_conf['info'], t[1], t[2]) for t in bench.measures[0].result]
[(1000, 0, 0), (1250, 0, 0), (1500, 0, 0), (1750, 0, 0), (2000, 0, 0)]
```

And the performance:

```
>>> # (std, max reward)
>>> print([(t[0].alg_conf['info'], int(t[1])) for t in bench.measures[1].result])
[(1000, -1781), (1250, -1853), (2000, -1901), (1500, -1906), (1750, -1958)]
```

Note that the numbers where produced in an example run. Since the optimization process uses a random number generator, the results will be different for every run. If we needed a statistical estimate for the results, we could run the algorithm multiple times with the same parameters and use comprehension to estimate expectation and standard deviation.

5 Content

5.1 Algorithms

Description

The `algo` module contains algorithm implementations based on the `AlgorithmBase` class. The objects should only be accessed through the interface functions defined in the base class.

Overview

Algorithm	Policy
A3C	NeuralNetwork
PolicyGradient	Any
Q-Learning	None
SafeOpt	Any

Implementing an Algorithm

When implementing an algorithm a couple of things have to be considered. `AlgorithmBase` is an abstrace base class. It will require any subclass to implement the private methods listed below. These will be invoked by the public interface methods.

Any algorithm must be structured using four methods. First the `optimize`, which will control the optimization run, it is responsible for using the other methods. The three tools `optimize` should use are the methods `initialize`, `step` and `is_finished`.

`initialize` should be used to initialize the run and all the attributes and parameters that need to be set up. `optimize` should compute one step of the optimization run. `is_finished` is supposed to return `True` when the optimization run is finished.

Requirements

Must implement	
_initialize	Initialize any attributes, objects needed.
_step	Execute one iteration of the algorithm.
_is_finished	Return `True` when done.

May implement	
_optimize(policy)	Optimize the policy. Possibly no policy as in Q-learning.

5.2 Environments

Description

The `envs` module contains environment implementations based on the `EnvironmentBase` class. The objects should only be accessed through the interface functions defined in the base class.

Overview

Environment	State Space	Action Space
GeneralMountainCar	$[-1, 1] \times [-0.07, 0.07]$	$[-1, 1]$
GymWrap		
LinearCar	$\mathbb{R}^{2d}$	$[-1, 1]^d$
MDP		
Quadrocopter		

Implementing an Environment

When implementing an environment a couple of things have to be considered. *EnvironmentBase* is an abstract base class. It will require any subclass to implement certain private methods which will be invoked by the public interface. Further certain attributes should be initialized, also as specified below, to support monitoring the execution.

Requirements

Environments have to inherit from *SafeRLBench.EnvironmentBase*.

Initialize Attributes		
state_space	Space object	
action_space	Space object	
horizon	Integer	Used in default _rollout implementation.

Must implement		
_update	action	Returns (action, state, reward)
_reset		

May implement		
_rollout	policy	Returns list of (action, state, reward)

5.3 API

Algorithm Module

This module contains implementations of different algorithms. Please refer to the class documentation for detailed instructions on how to use them.

Contents

AlgorithmBase

A3C

Policy Gradient

Q-Learning

SafeOpt

SafeOptSwarm

Environment Module

Contents

EnvironmentBase

GeneralMountainCar

GymWrap

LinearCar

MDP

Quadrocopter

Policy Module

Bases

Deterministic Policy Base

Probabilistic Policy Base

Linear Policies

LinearPolicy

DiscreteLinearPolicy

NoisyLinearPolicy

NonLinearQuadrocopterController

NeuralNetwork

Spaces Module

Contents

Space

BoundedSpace

DiscreteSpace

RdSpace

Measure Module

Contents

Measure

BestPerformance

SafetyMeasure

Benchmark

Contents

Bench

BenchConfig

BenchRun

Miscellaneous

Contents

Configuration