AdaptiveStressTestingToolbox

Release 2020.09.01.0

Oct 25, 2020

Contents

CHAPTER 1

Adaptive Stress Testing Toolbox

v2020.09.01.0.

CHAPTER 2

Overview

A toolbox for worst-case validation of autonomous policies.

Adaptive Stress Testing is a worst-case validation method for autonomous policies. This toolbox is being actively developed by the Stanford Intelligent Systems Lab.

See https://ast-toolbox.readthedocs.io/en/latest/ for documentation.

Maintained by the Stanford Intelligent Systems Lab (SISL)

- Free software: MIT license

2.1 Installation

2.1.1 Pip Installation Method

You can install the latest stable release from pypi:

```
pip install ast-toolbox
```

You can also install the latest version with:

```
pip install git+ssh://git@https://github.com/sisl/AdaptiveStressTestingToolbox.
↪git@master
```

Using the Go-Explore work requires having a Berkely DB installation findable on your system. If you are on Linux:

```
sudo apt-get update
sudo apt install libdb-dev python3-bsddb3
```

If you are on OSX:

```
brew install berkeley-db
export BERKELEYDB_DIR=$(brew --cellar)/berkeley-db/5.3
export YES_I_HAVE_THE_RIGHT_TO_USE_THIS_BERKELEY_DB_VERSION=1
```

Once you have the Berkeley DB system dependency met, you can install the toolbox:

```
pip install ast-toolbox[ge]
```

2.1.2 Git Installation Method

If you are interested in development, you should clone the repo. You can use https:

```
git clone https://github.com/sisl/AdaptiveStressTestingToolbox.git
```

You can also use ssh:

```
git clone git@github.com:sisl/AdaptiveStressTestingToolbox.git
```

If you are on Linux, use the following commands to setup the Toolbox:

```
cd AdaptiveStressTestingToolbox
git submodule update --init --recursive
sudo chmod a+x scripts/install_all.sh
sudo scripts/install_all.sh
source scripts/setup.sh
```

2.2 Documentation

You can find our documentation on readthedocs.

2.3 Development

Please see our Contributions Guide.

2.4 Acknowledgements

Built using the cookiecutter-pylibrary by Ionel Cristian Mărieș

CHAPTER 3

Tutorial

This tutorial is up-to-date for version '2020.09.01.0'

3.1 1 Introduction

This tutorial is intended for readers to learn how to use this package with their own simulator. Familiarity with the underlying theory is recommended, but is not strictly necessary for use. Please install the package before proceeding.

3.1.1 1.1 About AST

Adaptive Stress Testing is a way of finding flaws in an autonomous agent. For any non-trivial problem, searching the space of a stochastic simulation is intractable, and grid searches do not perform well. By modeling the search as a Markov decision process (MDP), we can use reinforcement learning to find the most probable failure. AST treats the simulator as a black box, and only needs access in a few specific ways. To interface a simulator to the AST packages, a few things will be needed:

- A **Simulator** wrapper that exposes the simulation software to this package. See *2.1 Simulation Options* for details on closed-loop vs. open-loop Simulators
- A **Reward** function dictates the optimization goals of the algorithm.
- The **Spaces** objects give information on the size and limits of a space. This will be used to define the **Observation Space** and the **Action Space**
- A **Runner** collects all of the run options and starts the experiment.

3.1.2 1.2 About this tutorial

In this tutorial, we will test a basic autonomous vehicle's ability to safely navigate a crosswalk. We will find the most-likely pedestrian trajectory that leads to a collision. The remainder of the tutorial is organized as follows:

- In Section 2, we will interface with a simulator (*2 Creating a Simulator*).

- In Section 3, we will create a reward function (*3 Creating a Reward Function*).
- In Section 4, we will define the action and state spaces (*4 Creating the Spaces*).
- In Section 5, we will create a runner file (*5 Creating a Runner*).
- In Section 6, we run the experiment (*6 Running the Example*).

3.2 2 Creating a Simulator

This sections explains how to create a wrapper that exposes your simulator to the AST package. The wrapper allows the AST solver to specify actions to control the stochasticity in the simulation. Examples of stochastic simulation elements could include an actor, like a pedestrian or a car, or noise elements, like on the beams of a LIDAR sensor. The simulator must be able to reset on command and detect if a goal state had been reached. The simulator state can be used, but is not necessary. Before we begin, let's define 3 different settings that tell the ASTEnv what sort of simulator it is interacting with.

We will be wrapping an example autonomous vehicle simulator that runs a toy problem of an autonomous vehicle approaching a crosswalk with pedestrians crossing. The simulator code can be found at ast_toolbox.simulators.example_av_simulator.toy_av_simulator.py.

3.2.1 2.1 Simulation Options

Three options must be specified to inform ASTEnv what type of simulator it is interacting with. They are listed as follows, with the default in bold, and the actual variable name in parentheses:

- **Open-loop** vs. Closed-loop control (open_loop): A *closed-loop* simulation is one in which control can be injected at each step during the actual simulation run, vs an *open-loop* simulation where all actions must be specified ahead of time. Essentially, in a closed-loop system we are "closing the loop" by including the toolbox in the calculation of each timestep. For example, if a simulation is run by creating a specification file, and no other control is possible, that simulation would be open-loop. There is no inherent advantage to either mode, and open-loop will be far more common. Closed-loop mode will generally only be used by white-box systems, where closed-loop control is required.
- **Black box simulation state** vs. White box simulation state (blackbox_sim_state): When running in *black box* simulation mode, the solver does not have access to the true state of the simulator, instead choosing actions based on the initial condition and the history of actions taken so far. If your simulator can provide access to the simulation state, it can be faster and more efficient to run in *white box* simulation mode, in which the simulation state is used as the input to the reinforcement learning algorithm at each time step. White box simulation mode requires closed-loop control.
- **Fixed initial state** vs. Generalized initial state (fixed_init_state): A simulation with a *fixed initial state* starts every rollout from the exact same simulation state, while a simulation with a *Generalized initial state* samples from a space of initial conditions. For example, if you had a 1-D state space, starting at x=0 would be a fixed initial state, while sampling x from [-2,2] at the start of each simulation would be a generalized initial state. For more information on the specifics see **'Efficient Autonomy Validation in Simulation with Adaptive Stress Testing https://arxiv.org/abs/1907.06795>'_**.

3.2.2 2.2 Inheriting the Base Simulator

Start by creating a file named `example_av_simulator.py` in the `simulators` folder. Create a class titled `ExampleAVSimulator`, which inherits from `Simulator`.

```
import numpy as np  # Used for math

from ast_toolbox.simulators import ASTSimulator  # import parent Simulator class
from ast_toolbox.simulators.example_av_simulator import ToyAVSimulator  # import the␣
↪simulator to wrap


class ExampleAVSimulator(ASTSimulator):  # Define the class
```

The base generator accepts four values, three of which are boolean values for the settings defined in *2.1 Simulation Options*:

- **max_path_length**: The horizon of the simulation, in number of timesteps
- **open_loop**: True for open-loop simulation, False for closed-loop simulation
- **blackbox_sim_state**: True for black box simulation state, False for white box simulation state
- **fixed_init_state**: True for fixed initial simulation state, False for generalized initial simulation state

A child of the `ASTSimulator` class is required to define the following three functions:

- `simulate`.
- `get_reward_info`.
- `is_goal`.

The following functions may be optionally overridden as well:

- `closed_loop_step`.
- `reset`.
- `clone_state`.
- `restore_state`.
- `render`.

Finally, it is not recommended that you touch these functions:

- `step`.
- `observation_return`.
- `is_terminal`.

For use with the Go-Explore algorithm, the `clone_state` and `restore_state` functions must be defined.

3.2.3 2.3 Initializing the Example Simulator

Our example simulator takes 3 values: * **num_peds**: The number of pedestrians in the scenario. * **simulator_args**: A dict of named arguments to be passed to the toy simulator. * **kwargs**: Any keyword arguement not listed here. In particular, the base class arguments covered in *2.2 Inheriting the Base Simulator* should be passed to the base Simulator as one of the **kwargs.

The toy simulator will control a modified version of the Intelligent Driver Model (IDM) as our system under test (SUT), while adding sensor noise and filtering it out with an alpha-beta tracker. Initial simulation conditions are needed here as well. Because of all this, the Simulator accepts a number of inputs:

- **num_peds**: The number of pedestrians in the scenario
- **dt**: The length of the time step, in seconds

- **alpha**: A hyperparameter controlling the alpha-beta tracker that filters noise from the sensors
- **beta**: A hyperparameter controlling the alpha-beta tracker that filters noise from the sensors
- **v_des**: The desired speed of the SUT
- **t_headway**: An IDM hyperparameter that controls the target seperation between the SUT and the agent it is following, measured in seconds
- **a_max**: An IDM hyperparameter that controls the maximum acceleration of the SUT
- **s_min**: An IDM hyperparameter that controls the minimum distance between the SUT and the agent it is following
- **d_cmf**: An IDM hyperparameter that controls the maximum comfortable decceleration of the SUT (a soft maximum that is only violated to avoid crashes)
- **d_max**: An IDM hyperparameter that controls the maximum decceleration of the SUT
- **min_dist_x**: Defines the length of the hitbox in the x direction
- **min_dist_y**: Defines the length of the hitbox in the y direction
- **car_init_x**: Specifies the initial x-position of the SUT
- **car_init_y**: Specifies the initial y-position of the SUT

In addition, there are a number of member variables that need to be initialized. The code is below:

```
def __init__(self,
             num_peds=1,
             simulator_args=None,
             **kwargs):

    # Constant hyper-params -- set by user
    self.c_num_peds = num_peds
    if simulator_args is None:
        simulator_args = {}

    self._action = np.array([0] * (6 * self.c_num_peds))
    self.simulator = ToyAVSimulator(num_peds=num_peds, **simulator_args)

    # initialize the parent ASTSimulator
    super().__init__(**kwargs)
```

3.2.4 2.4 The `simulate` function:

The simulate function runs a simulation using previously generated actions from the policy to control the stochasticity. The simulate function accepts a list of actions and an initial state. It should run the simulation, then return the timestep in which the goal state was achieved, or a -1 if the horizon was reached first. In addition, this function should return any simulation info needed for post-analysis.

For the example, out toy simulator conveniently has a single function to call that already follows the same conventions. Note that in most cases, the simulate function may require significantly more API calls to the simulator, as well as changing the inputs and outputs to forms the simulator will accept and back again. Now we implement the `simulate` function, checking to be sure that the horizon wasn't reached:

```
def simulate(self, actions, s_0):

    return self.simulator.run_simulation(actions=actions, s_0=s_0, simulation_
↪horizon=self.c_max_path_length)
```

3.2.5 2.5 The `closed_loop_step` function (Optional):

If a simulation is closed-loop, the `closed_loop_step` function should step the simulation forward at each timestep. The functions takes as input the current action. We return the output of `observation_return` function defined by the `ASTSimulator`, which ensures we return the correct values depending on the simulator settings. It is highly recommended to use this function. If the simulation is open-loop, other per-step actions can still be put here if it is desirable - this function is called at each timestep either way. Since we are running the simulator open-loop in this tutorial, we could just have this function return None. However, we have implemented the function as an example of how the simulator could be run closed-loop.

Again, our toy simulator already has a closed-loop mode that follows the same convention so we can just call the `step_simulation` function.

```
def closed_loop_step(self, action):

    # grab simulation state, if interactive
    self.observation = np.ndarray.flatten(self.simulator.step_simulation(action))

    return self.observation_return()
```

3.2.6 2.6 The `reset` function (Optional):

The reset function should return the simulation to a state where it can accept the next sequence of actions. In some cases this may mean explicitly resetting the simulation parameters, like SUT location or simulation time. It could also mean opening and initializing a new instance of the simulator (in which case the `simulate` function should close the current instance). Your implementation of the `reset` function may be something else entirely, it is highly dependent on how your simulator functions. The method takes the initial state as an input, and returns the state of the simulator after the reset actions are taken. If reset is defined, `observation_return` should again be used to return the correct observation type. In addition, the super class's reset must still be called.

Our toy simulator already has a reset function, so we just call the super class's reset, call the toy simulator's reset, and then return `observation_return`.

```
def reset(self, s_0):

    # Call ASTSimulator's reset function (required!)
    super(ExampleAVSimulator, self).reset(s_0=s_0)
    # Reset the simulation
    self.observation = np.ndarray.flatten(self.simulator.reset(s_0))
```

3.2.7 2.7 The `get_reward_info` function:

It is likely that your reward function (see *3 Creating a Reward Function*) will need some information from the simulator. The reward function will be passed whatever information is returned from this function.

For the example, the example reward function uses a heuristic reward to help guide the policy toward failures – when a trajectory ends without a crash, an extra penalty is applied that scales with the distance between the SUT and the nearest pedestrian in the last timestep. To do this, both the car and pedestrian locations are returned. In addition, boolean values indicating whether a crash has been found or if the horizon has been reached are returned. To access these values, we grab the ground truth state from the toy simulator.

```
# Get the ground truth state from the toy simulator
sim_state = self.simulator.get_ground_truth()
```

(continues on next page)

(continued from previous page)

```
return {"peds": sim_state['peds'],
        "car": sim_state['car'],
        "is_goal": self.is_goal(),
        "is_terminal": self.is_terminal()}
```

3.2.8 2.8 The `is_goal` function:

This function returns a boolean value indicating if the current state is in the goal set.

In the example, this is True if the pedestrian is hit by the car. The toy simulator has a `collision_detected` function that we can call to check for a collision.

```
def is_goal(self):

    # Ask the toy simulator if a collision was detected
    return self.simulator.collision_detected()
```

3.2.9 2.9 The `log` function (Optional):

The log function is a way to store variables from the simulator for later access.

In the example, some simulation state information is appended to a list at every timestep after getting the ground truth from the toy simulator.

```
# Get the ground truth state from the toy simulator
sim_state = self.simulator.get_ground_truth()

# Create a cache of step specific variables for post-simulation analysis
cache = np.hstack([0.0,  # Dummy, will be filled in with trial # during post␣
↪processing in save_trials.py
                   sim_state['step'],
                   np.ndarray.flatten(sim_state['car']),
                   np.ndarray.flatten(sim_state['peds']),
                   np.ndarray.flatten(sim_state['action']),
                   np.ndarray.flatten(sim_state['car_obs']),
                   0.0])

self._info.append(cache)
```

3.2.10 2.10 The `clone_state` and `restore_state` functions (Optional):

Some parts of the Toolbox (for example, Go-Explore and the Backward Algorithm) rely on deterministic resets of the simulator to find failures efficiently. The `clone_state` and `restore_state` functions provide this functionality.

The `clone_state` function should return a 1-D numpy array with enough information to deterministically reset the simulation to an exact state.

In our example, the toy simulator's `get_ground_truth` returns a dictionary of state variables, so we arrange them into a numpy array:

```
def clone_state(self):
```

(continues on next page)

(continued from previous page)

```
    # Get the ground truth state from the toy simulator
    simulator_state = self.simulator.get_ground_truth()

    return np.concatenate((np.array([simulator_state['step']]),
                           np.array([simulator_state['path_length']]),
                           np.array([int(simulator_state['is_terminal'])]),
                           simulator_state['car'],
                           simulator_state['car_accel'],
                           simulator_state['peds'].flatten(),
                           simulator_state['car_obs'].flatten(),
                           simulator_state['action'].flatten(),
                           simulator_state['initial_conditions']), axis=0)
```

The `restore_state` function should accept a 1-D array and use it to deterministically reset it to a specific state. How you do the reset is up to you, whether it is through a reset style scenario instantiation, through running the simulator from the start back to the exact same point, or another method altogether.

The toy simulator has a `set_ground_truth` function that sets it to a specific state, so we will use that. We take the 1-D array and translate it back into a dictionary of state variables that the toy simulator wants. We also set the state variables of the `ExampleAVSimulator`:

```
def restore_state(self, in_simulator_state):

    # Put the simulators state variables in dict form
    simulator_state = {}

    simulator_state['step'] = in_simulator_state[0]
    simulator_state['path_length'] = in_simulator_state[1]
    simulator_state['is_terminal'] = bool(in_simulator_state[2])
    simulator_state['car'] = in_simulator_state[3:7]
    simulator_state['car_accel'] = in_simulator_state[7:9]
    peds_end_index = 9 + self.c_num_peds * 4
    simulator_state['peds'] = in_simulator_state[9:peds_end_index].reshape((self.c_
↪num_peds, 4))
    car_obs_end_index = peds_end_index + self.c_num_peds * 4
    simulator_state['car_obs'] = in_simulator_state[peds_end_index:car_obs_end_index].
↪reshape((self.c_num_peds, 4))
    simulator_state['action'] = in_simulator_state[car_obs_end_index:car_obs_end_
↪index + self._action.shape[0]]
    simulator_state['initial_conditions'] = in_simulator_state[car_obs_end_index +␣
↪self._action.shape[0]:]

    # Set ground truth of actual simulator
    self.simulator.set_ground_truth(simulator_state)

    # Set wrapper state variables
    self._info = []
    self.initial_conditions = np.array(simulator_state['initial_conditions'])
    self._is_terminal = simulator_state['is_terminal']
    self._path_length = simulator_state['path_length']
```

3.3 3 Creating a Reward Function

This section explains how to create a function that dictates the reward at each timestep of a simulation. AST formulates the problem of searching the space of possible rollouts of a stochastic simulation as an MDP so that modern-day

reinforcement learning (RL) techniques can be used. When optimizing a policy using RL, the reward function is of the utmost importance, as it determines what behavior the agent will learn. Changing the reward function to achieve the desired policy is known as reward shaping.

3.3.1 3.1 Reward Shaping

SPOILER ALERT: This section uses a famous summer-camp game as an example. If you are planning on attending a children's summer-camp in the near future I highly recommend you skip this section, lest you ruin the counselors' attempts at having fun at your expense. You have been warned.

As an example of reinforcement learning, and the importance of the reward function, consider the famous children's game "The Hat Game." Common at summer-camps, the game usually starts with a counselor holding a hat in his hands, telling the kids he is about to teach them a new game. He will say "Ok, ready everyone....? I can play the hat game," proceed to do a bunch of random things with the hat, such as flipping it over or tossing it in the air, and then say "how about you?" He will then pass the hat to a camper, who repeats almost exactly everything the counselor does, but is told "no, you didn't play the hat game." Another counselor will take the hat, say the words, do something completely different with it, and the game is on. The trick is actually the word "OK" - so long as you say that magic word, you have played the hat game, even if you have no hat.

How does this relate to reward shaping? In this case, the children are the policy. They are taking stochastic actions, trying to learn how to play the hat game. The key to the game being fun is that the children are predisposed to pay attention to the hat motions, but not the words beforehand. However, after enough trials (and it can take a long time), most of them will pick up the pattern and attention will shift to "OK." In the vanilla game, there are two rewards. "Yes, you played the hat game" can be considered positive, and "No, you didn't play the hat game" can be considered negative, or just zero. By changing this reward, we could make the game difficulty radically different. Imagine if 10 kids tried the game, and all they got was a binary response on if at least one of them played the game. This would be much harder to pick up on! This is an example of a sparse reward function, or one that only rarely gives rewards, such as at the end of a trajectory. On the other hand, what if the children received feedback after every single word or motion on if they had played the hat game during that trial yet. The game would be much easier! These are examples of how different reward functions can make achieving the same policy easier or harder.

How does this relate yo our tutorial? Similar to the kids, our policy will be trying to learn the correct behavior from rewards. While some policies may be better at this task than others, all of them will struggle if the reward function is too sparse. We can make the task much easier, and therefore get better and faster results, if we can introduce heuristic rewards that guide our policy to failures. .. _tutorial-inheriting-the-base-reward-function:

3.3.2 3.2 Inheriting the Base Reward Function

Start by creating a file named `example_av_reward.py` in the `rewards` folder. Create a class title `ExampleAVReward` which inherits from `ASTReward`:

```
import numpy as np # useful packages for math

from ast_toolbox.rewards import ASTReward # import base class


# Define the class, inherit from the base
class ExampleAVReward(ASTReward):
```

The base class does not take any inputs, and there is only one required function - `give_reward`.

3.3.3 3.3 Initializing the Example Reward Function

The reward function will be calculating some rewards based on the probability of certain actions. We have assumed the means action is the 0 vector, but we still need to take the following inputs:

- **num_peds**: The number of pedestrians in the scenario
- **cov_x**: The covariance of the gaussian distribution used to model the x-acceleration of a pedestrian
- **cov_y**: The covariance of the gaussian distribution used to model the y-acceleration of a pedestrian
- **cov_sensor_noise**: The covariance of the gaussian distribution used to model the noise on a sensor measurement in both the x and y directions (assumed equal)
- **use_heuristic**: Whether our reward function should use the heuristic reward we provide. As mentioned above, using this reward, when possible, will improve results and decrease training time.

The code is below:

```
def __init__(self,
             num_peds=1,
             cov_x=0.1,
             cov_y=0.01,
             cov_sensor_noise=0.1,
             use_heuristic=True):

    self.c_num_peds = num_peds
    self.c_cov_x = cov_x
    self.c_cov_y = cov_y
    self.c_cov_sensor_noise = cov_sensor_noise
    self.use_heuristic = use_heuristic
    super().__init__()
```

3.3.4 3.4 The `give_reward` function

Our example reward function is broken down into three cases, as specified in the paper. The three cases are as follows:

1. There is a crash at the current timestep
2. The horizon of the simulation is reached, with no crash
3. The current step did not find a crash or reach the horizon

The respective reward for each case is as follows:

1. R = 0
2. R = -1E5 - 1E4 * {The distance between the car and the closest pedestrian}
3. R = -log(1 + {likelihood of the actions take})

For case 2, we use the distance between the car and the closest pedestrian as a heuristic to increase convergence speed. In the early trials, this teaches pedestrians to end closer to the car, which makes it easier to find crash trajectories (see *3.1 Reward Shaping*). For case 3, using the negative log-likelihood allows us to sum the rewards to find a value that is proportional to the probability of the trajectory. As a stand in for the probability of an action, we use the Mahalanobis distance, a multi-dimensional generalization of distance from the mean. Add the following helper function to your file:

```
def mahalanobis_d(self, action):
    # Mean action is 0
    mean = np.zeros((6 * self.c_num_peds, 1))
    # Assemble the diagonal covariance matrix
    cov = np.zeros((self.c_num_peds, 6))
    cov[:, 0:6] = np.array([self.c_cov_x, self.c_cov_y,
                            self.c_cov_sensor_noise, self.c_cov_sensor_noise,
                            self.c_cov_sensor_noise, self.c_cov_sensor_noise])
    big_cov = np.diagflat(cov)

    # subtract the mean from our actions
    dif = np.copy(action)
    dif[::2] -= mean[0, 0]
    dif[1::2] -= mean[1, 0]

    # calculate the Mahalanobis distance
    dist = np.dot(np.dot(dif.T, np.linalg.inv(big_cov)), dif)

    return np.sqrt(dist)
```

Now we are ready to calculate the reward. The `give_reward` function takes in an action, as well as the info bundle that was returned from the `get_reward_info` function in the `ExampleAVSimulator` (see *2.7 The get_reward_info function:*). The code is as follows:

```
def give_reward(self, action, **kwargs):
    # get the info from the simulator
    info = kwargs['info']
    peds = info["peds"]
    car = info["car"]
    is_goal = info["is_goal"]
    is_terminal = info["is_terminal"]
    dist = peds[:, 2:4] - car[2:4]

    # update reward and done bool

    if (is_goal):  # We found a crash
        reward = 0
    elif (is_terminal):
        # reward = 0
        # Heuristic reward based on distance between car and ped at end
        if self.use_heuristic:
            heuristic_reward = np.min(np.linalg.norm(dist, axis=1))
        else:
            # No Herusitic
            heuristic_reward = 0
        reward = -100000 - 10000 * heuristic_reward  # We reached
        # the horizon with no crash
    else:
        reward = -self.mahalanobis_d(action)  # No crash or horizon yet

    return reward
```

3.4 4 Creating the Spaces

This section shows how to create the action space and observation space for `garage` to use. The spaces define the limits of what is possible for inputs to and outputs from the policy. The observation space can be used as input if the simulation state is accessible, and can be used to generate initial conditions if they are being sampled from a range. The action space defines the output space of the policy, and controls the size of the output array from the policy.

3.4.1 4.1 Inheriting the Base Spaces

Create a file named `example_av_spaces.py` in the `spaces` folder. Create a class titled `ExampleAVSpaces` which inherits from `ASTSpaces`:

```
import numpy as np
from gym.spaces.box import Box

from ast_toolbox.spaces import ASTSpaces


class ExampleAVSpaces(ASTSpaces):
```

The base spaces don't take any input, but there are two functions to define: `action_space` and `observation_space`. Both of these functions should return an object that inherits from the ''Space" class, imported from `gym.spaces`. There are a few options, and you can implement your own, but the `Box` class is used here. A `Box` is defined by two arrays, `low` and `high`, of equal length, which specify the minimum and maximum value of each position in the array. The space then allows any continuous number between the low and high values.

3.4.2 4.2 Initializing the Spaces

In order to define our spaces, there are a number of inputs:

- **num_peds**: The number of pedestrians in the scenario
- **max_path_length**: The horizon of the trajectory rollout, in number of timesteps
- **v_des**: The desired velocity of the SUT
- **x_accel_low**: The minimum acceleration in the x-direction of the pedestrian
- **y_accel_low**: The minimum acceleration in the y-direction of the pedestrian
- **x_accel_high**: The maximum acceleration in the x-direction of the pedestrian
- **y_accel_high**: The maximum acceleration in the y-direction of the pedestrian
- **x_boundary_low**: The minimum x-position of the pedestrian
- **y_boundary_low**: The minimum y-position of the pedestrian
- **x_boundary_high**: The maximum x-position of the pedestrian
- **y_boundary_high**: The maximum y-position of the pedestrian
- **x_v_low**:: The minimum initial x-velocity of the pedestrian
- **y_v_low**:: The minimum initial y-velocity of the pedestrian
- **x_v_high**:: The maximum initial x-velocity of the pedestrian
- **y_v_high**:: The maximum initial y-velocity of the pedestrian

- **car_init_x**: The initial x-position of the SUT
- **car_init_y**: The initial y-position of the SUT
- **open_loop**: Whether or not the simulation is being run in open-loop mode (See *2.1 Simulation Options*)

The initialization code is below:

```
def __init__(self,
             num_peds=1,
             max_path_length=50,
             v_des=11.17,
             x_accel_low=-1.0,
             y_accel_low=-1.0,
             x_accel_high=1.0,
             y_accel_high=1.0,
             x_boundary_low=-10.0,
             y_boundary_low=-10.0,
             x_boundary_high=10.0,
             y_boundary_high=10.0,
             x_v_low=-10.0,
             y_v_low=-10.0,
             x_v_high=10.0,
             y_v_high=10.0,
             car_init_x=-35.0,
             car_init_y=0.0,
             open_loop=True,
             ):

    # Constant hyper-params -- set by user
    self.c_num_peds = num_peds
    self.c_max_path_length = max_path_length
    self.c_v_des = v_des
    self.c_x_accel_low = x_accel_low
    self.c_y_accel_low = y_accel_low
    self.c_x_accel_high = x_accel_high
    self.c_y_accel_high = y_accel_high
    self.c_x_boundary_low = x_boundary_low
    self.c_y_boundary_low = y_boundary_low
    self.c_x_boundary_high = x_boundary_high
    self.c_y_boundary_high = y_boundary_high
    self.c_x_v_low = x_v_low
    self.c_y_v_low = y_v_low
    self.c_x_v_high = x_v_high
    self.c_y_v_high = y_v_high
    self.c_car_init_x = car_init_x
    self.c_car_init_y = car_init_y
    self.open_loop = open_loop
    self.low_start_bounds = [-1.0, -6.0, -1.0, 5.0, 0.0, -6.0, 0.0, 5.0]
    self.high_start_bounds = [1.0, -1.0, 0.0, 9.0, 1.0, -2.0, 1.0, 9.0]
    self.v_start = [1.0, -1.0, 1.0, -1.0]
    super().__init__()
```

3.4.3 4.3 The Action Space

The `action_space` function takes no inputs and returns a child of the `Space` class. The length of the action space array determines the output dimension of the policy. Note the `@Property` decorator in the code below:

```
@property
def action_space(self):
    """
    Returns a Space object
    """
    low = np.array([self.c_x_accel_low, self.c_y_accel_low, -3.0, -3.0, -3.0, -3.0])
    high = np.array([self.c_x_accel_high, self.c_y_accel_high, 3.0, 3.0, 3.0, 3.0])

    for i in range(1, self.c_num_peds):
        low = np.hstack((low, np.array([self.c_x_accel_low, self.c_y_accel_low, 0.0,␣
↪0.0, 0.0, 0.0])))
        high = np.hstack((high, np.array([self.c_x_accel_high, self.c_y_accel_high, 1.
↪0, 1.0, 1.0, 1.0])))

    return Box(low=low, high=high, dtype=np.float32)
```

3.4.4 4.4 The Observation Space

The `observation_space` function takes no inputs and returns a child of the `Space` class. If the simulation state is accessible, the ranges of possible values should be defined using this function, which determines the expected input shape to the policy. If initial conditions are sampled, they will be sampled from the observation space. Therefore, the observation space should define the maximum and minimum value of every simulation state that will be passed as input to the policy, as well as a value for every initial condition needed to specify a scenario variation. Note the `@Property` decorator in the code below:

```
@property
def observation_space(self):
    """
    Returns a Space object
    """

    low = np.array([self.c_x_v_low, self.c_y_v_low, self.c_x_boundary_low, self.c_y_
↪boundary_low])
    high = np.array([self.c_x_v_high, self.c_y_v_high, self.c_x_boundary_high, self.c_
↪y_boundary_high])

    for i in range(1, self.c_num_peds):
        low = np.hstack(
            (low, np.array([self.c_x_v_low, self.c_y_v_low, self.c_x_boundary_low,␣
↪self.c_y_boundary_low])))
        high = np.hstack(
            (high, np.array([self.c_x_v_high, self.c_y_v_high, self.c_x_boundary_high,
↪ self.c_y_boundary_high])))

    if self.open_loop:
        low = self.low_start_bounds[:self.c_num_peds * 2]
        low = low + np.ndarray.tolist(0.0 * np.array(self.v_start))[:self.c_num_peds]
        low = low + [0.75 * self.c_v_des]

        high = self.high_start_bounds[:self.c_num_peds * 2]
        high = high + np.ndarray.tolist(2.0 * np.array(self.v_start))[:self.c_num_
↪peds]
        high = high + [1.25 * self.c_v_des]

        if self.c_car_init_x > 0:
```

(continues on next page)

(continued from previous page)

```
            low = low + [0.75 * self.c_car_init_x]
            high = high + [1.25 * self.c_car_init_x]
        else:
            low = low + [1.25 * self.c_car_init_x]
            high = high + [0.75 * self.c_car_init_x]

    return Box(low=np.array(low), high=np.array(high), dtype=np.float32)
```

3.5 5 Creating a Runner

This section explains how to create a file to run the experiment we have been creating. This will use all of the example files we have created, and interface them with the a package for handling RL. The backend framework handling the policy definition and optimization is a package called RLLAB. The project is open-source, so if you would like to understand more about what RLLAB is doing please see the documentation here.

3.5.1 5.1 Setting Up the Runners

Create a file called `example_runner.py` in your working directory. Add the following code to handle all of the necessary imports:

```
# Import the example classes
import os

import fire
# Useful imports
import tensorflow as tf
from garage.envs.normalized_env import normalize
from garage.experiment import run_experiment
from garage.np.baselines.linear_feature_baseline import LinearFeatureBaseline
# Import the necessary garage classes
from garage.tf.algos.ppo import PPO
from garage.tf.envs.base import TfEnv
from garage.tf.experiment import LocalTFRunner
from garage.tf.optimizers.conjugate_gradient_optimizer import␣
↪ConjugateGradientOptimizer
from garage.tf.optimizers.conjugate_gradient_optimizer import FiniteDifferenceHvp
# from garage.tf.policies.gaussian_lstm_policy import GaussianLSTMPolicy
from garage.tf.policies import GaussianLSTMPolicy

# Import the AST classes
from ast_toolbox.envs import ASTEnv
from ast_toolbox.rewards import ExampleAVReward
from ast_toolbox.samplers import ASTVectorizedSampler
from ast_toolbox.simulators import ExampleAVSimulator
from ast_toolbox.spaces import ExampleAVSpaces
from ast_toolbox.utils.go_explore_utils import load_convert_and_save_expert_trajectory
```

3.5.2 5.2 Specifying the Experiment

All of the classes imported earlier will now be used to specify the experiment. We will create a `runner` function that takes in dictionaries of keyword arguments for the different objects. The function will define a `run_task` function

that executes an experiment, and then will pass this function's handle to the `run_experiment` function. See the `garage` docs for more info.

```
def runner(
    env_args=None,
    run_experiment_args=None,
    sim_args=None,
    reward_args=None,
    spaces_args=None,
    policy_args=None,
    baseline_args=None,
    algo_args=None,
    runner_args=None,
    sampler_args=None,
    save_expert_trajectory=False,
):

    if env_args is None:
        env_args = {}

    if run_experiment_args is None:
        run_experiment_args = {}

    if sim_args is None:
        sim_args = {}

    if reward_args is None:
        reward_args = {}

    if spaces_args is None:
        spaces_args = {}

    if policy_args is None:
        policy_args = {}

    if baseline_args is None:
        baseline_args = {}

    if algo_args is None:
        algo_args = {}

    if runner_args is None:
        runner_args = {'n_epochs': 1}

    if sampler_args is None:
        sampler_args = {}

    if 'n_parallel' in run_experiment_args:
        n_parallel = run_experiment_args['n_parallel']
    else:
        n_parallel = 1
        run_experiment_args['n_parallel'] = n_parallel

    if 'max_path_length' in sim_args:
        max_path_length = sim_args['max_path_length']
    else:
        max_path_length = 50
        sim_args['max_path_length'] = max_path_length
```

(continues on next page)

(continued from previous page)

```
    if 'batch_size' in runner_args:
        batch_size = runner_args['batch_size']
    else:
        batch_size = max_path_length * n_parallel
        runner_args['batch_size'] = batch_size

    def run_task(snapshot_config, *_):

        config = tf.ConfigProto()
        config.gpu_options.allow_growth = True
        with tf.Session(config=config) as sess:
            with tf.variable_scope('AST', reuse=tf.AUTO_REUSE):

                with LocalTFRunner(
                        snapshot_config=snapshot_config, max_cpus=4, sess=sess) as␣
↪local_runner:
                    # Instantiate the example classes
                    sim = ExampleAVSimulator(**sim_args)
                    reward_function = ExampleAVReward(**reward_args)
                    spaces = ExampleAVSpaces(**spaces_args)

                    # Create the environment
                    if 'id' in env_args:
                        env_args.pop('id')
                    env = TfEnv(normalize(ASTEnv(simulator=sim,
                                                 reward_function=reward_function,
                                                 spaces=spaces,
                                                 **env_args
                                                 )))

                    # Instantiate the garage objects
                    policy = GaussianLSTMPolicy(env_spec=env.spec, **policy_args)

                    baseline = LinearFeatureBaseline(env_spec=env.spec, **baseline_
↪args)

                    optimizer = ConjugateGradientOptimizer
                    optimizer_args = {'hvp_approach': FiniteDifferenceHvp(base_eps=1e-
↪5)}

                    algo = PPO(env_spec=env.spec,
                               policy=policy,
                               baseline=baseline,
                               optimizer=optimizer,
                               optimizer_args=optimizer_args,
                               **algo_args)

                    sampler_cls = ASTVectorizedSampler
                    sampler_args['sim'] = sim
                    sampler_args['reward_function'] = reward_function

                    local_runner.setup(
                        algo=algo,
                        env=env,
                        sampler_cls=sampler_cls,
                        sampler_args=sampler_args)
```

(continues on next page)

(continued from previous page)

```
                    # Run the experiment
                    local_runner.train(**runner_args)
                    print('done!')

    run_experiment(
        run_task,
        **run_experiment_args,
    )
```

3.5.3 5.3 Running the Experiment

Now create a file named `example_batch_runner.py`. While `example_runner.py` gave us a runner template, the batch runner will be where we specify the actual arguments that define our experiment set-up. By dividing the files in this way, it makes it much easier to set-up and run many different experiment specifications at once.

```
import pickle

from examples.AV.example_runner_drl_av import runner as drl_runner

if __name__ == '__main__':
    # Overall settings
    max_path_length = 50
    s_0 = [0.0, -4.0, 1.0, 11.17, -35.0]
    base_log_dir = './data'
    # experiment settings
    run_experiment_args = {'snapshot_mode': 'last',
                           'snapshot_gap': 1,
                           'log_dir': None,
                           'exp_name': None,
                           'seed': 0,
                           'n_parallel': 8,
                           'tabular_log_file': 'progress.csv'
                           }

    # runner settings
    runner_args = {'n_epochs': 101,
                   'batch_size': 5000,
                   'plot': False
                   }

    # env settings
    env_args = {'id': 'ast_toolbox:GoExploreAST-v1',
                'blackbox_sim_state': True,
                'open_loop': False,
                'fixed_init_state': True,
                's_0': s_0,
                }

    # simulation settings
    sim_args = {'blackbox_sim_state': True,
                'open_loop': False,
                'fixed_initial_state': True,
                'max_path_length': max_path_length
                }
```

(continues on next page)

(continued from previous page)

```
# reward settings
reward_args = {'use_heuristic': True}

# spaces settings
spaces_args = {}

# DRL Settings

drl_policy_args = {'name': 'lstm_policy',
                   'hidden_dim': 64,
                   }

drl_baseline_args = {}

drl_algo_args = {'max_path_length': max_path_length,
                 'discount': 0.99,
                 'lr_clip_range': 1.0,
                 'max_kl_step': 1.0,
                 # 'log_dir':None,
                 }


# DRL settings
exp_log_dir = base_log_dir
run_experiment_args['log_dir'] = exp_log_dir + '/drl'
run_experiment_args['exp_name'] = 'drl'

drl_runner(
    env_args=env_args,
    run_experiment_args=run_experiment_args,
    sim_args=sim_args,
    reward_args=reward_args,
    spaces_args=spaces_args,
    policy_args=drl_policy_args,
    baseline_args=drl_baseline_args,
    algo_args=drl_algo_args,
    runner_args=runner_args,
)
```

3.6 6 Running the Example

This section explains how to run the program, and what the results should look like. Double check that all of the files created earlier in the tutorial are correct (a correct version of each is already included in the repository). Also check that the conda environment is activated, and that garage has been added to your `PYTHONPATH`, as explained in the installation guide.

3.6.1 6.1 Running from the Command Line

Since everything has been configured already in the runner file, running the example is easy. Use the code below in the command line to execute the example program from the top-level directory:

```
mkdir data
python example_batch_runner.py
```

Here we are creating a new directory for the output, and then running the batch runner we created above (see *5.3 Running the Experiment*). The program should run for 101 iterations, unless you have changed it. This may take some time!

3.6.2 6.2 Example Output

As you run the program, rllab will output optimization updates to the terminal. When the method runs iteration 100, you should see something that looks like this:

```
| ----------------------  ----------------
| PolicyExecTime               0.138965
| EnvExecTime                  0.471907
| ProcessExecTime              0.0285957
| Iteration                  100
| AverageDiscountedReturn   -897.273
| AverageReturn            -1437.22
| ExplainedVariance            0.136119
| NumTrajs                   80
| Entropy                      8.22841
| Perplexity                3745.86
| StdReturn                 4448.98
| MaxReturn                 -102.079
| MinReturn                -24631
| LossBefore                  -5.66416e-05
| LossAfter                   -0.0234421
| MeanKLBefore                 0.0725254
| MeanKL                       0.0915881
| dLoss                        0.0233855
| Time                       857.771
| ItrTime                      8.16877
| ----------------------  ----------------
```

If everything works right, the max return in the last several iterations should be around -100. If you got particularly lucky, the average return may be close to that as well. For your own projects, these numbers may be very different, depending on your reward function.

CHAPTER 4

Installation

At the command line:

```
pip install ast-toolbox
```

You can also install the in-development version with:

```
pip install git+ssh://git@https://github.com/sisl/AdaptiveStressTestingToolbox.
↪git@master
```

Using the Go-Explore work requires having a Berkely DB installation findable on your system. If you are on Linux:

```
sudo apt-get update
sudo apt install libdb-dev python3-bsddb3
```

If you are on OSX:

```
brew install berkeley-db
export BERKELEYDB_DIR=$(brew --cellar)/berkeley-db/5.3
export YES_I_HAVE_THE_RIGHT_TO_USE_THIS_BERKELEY_DB_VERSION=1
```

Once you have the Berkeley DB system dependency met, you can install the toolbox:

```
pip install ast-toolbox[ge]
```

CHAPTER 5

Usage

The Adaptive Stress Testing (AST) Toolbox is designed to allow users to use AST to validate their own autonomous policies within their own simulators. AST formulates the problem of finding the most-likely failure in a system as a Markov decision process (MDP), which can then be solved with reinforcement learning (RL) techniques. The AST methodology is shown below:

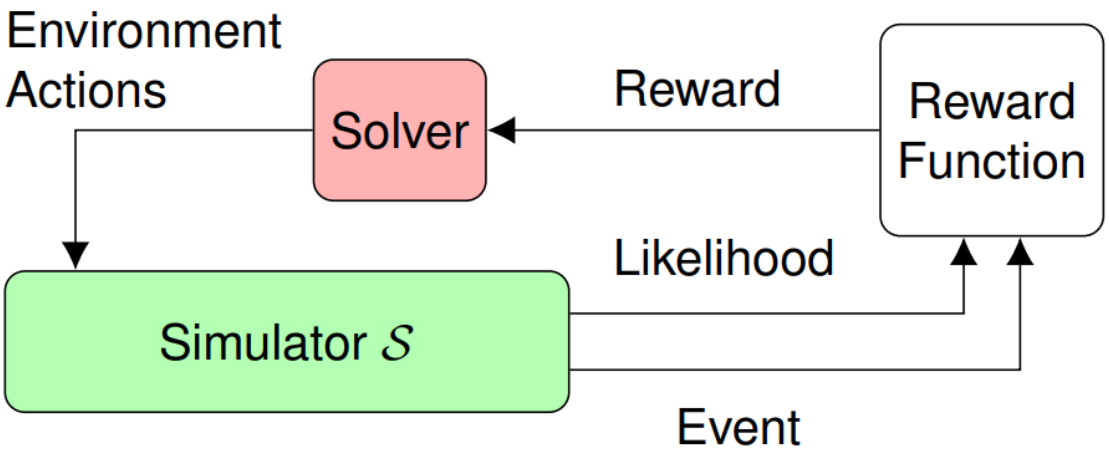

Fig. 1: The Adaptive Stress Testing methodology.

In AST the simulator, which contains the system under test (SUT) is treated as a black-box. The solver tried to force failures in the SUT by controlling the simulation through the environment actions. After a simulation rollout, the solver receives a reward, calculated by the reward function, that is dependent on if a failure occurred and how likely the trajectory was. The solver uses the reward during optimization, allowing it to learn to cause likelier failures. This methodology leads to the following Toolbox framework:

The ASTEnv is the core of the toolbox. Using the provided wrappers, the ASTEnv turns a user's simulator into a gym environment, which can then be solved using existing reinforcement learning software. In order for this to work, a user

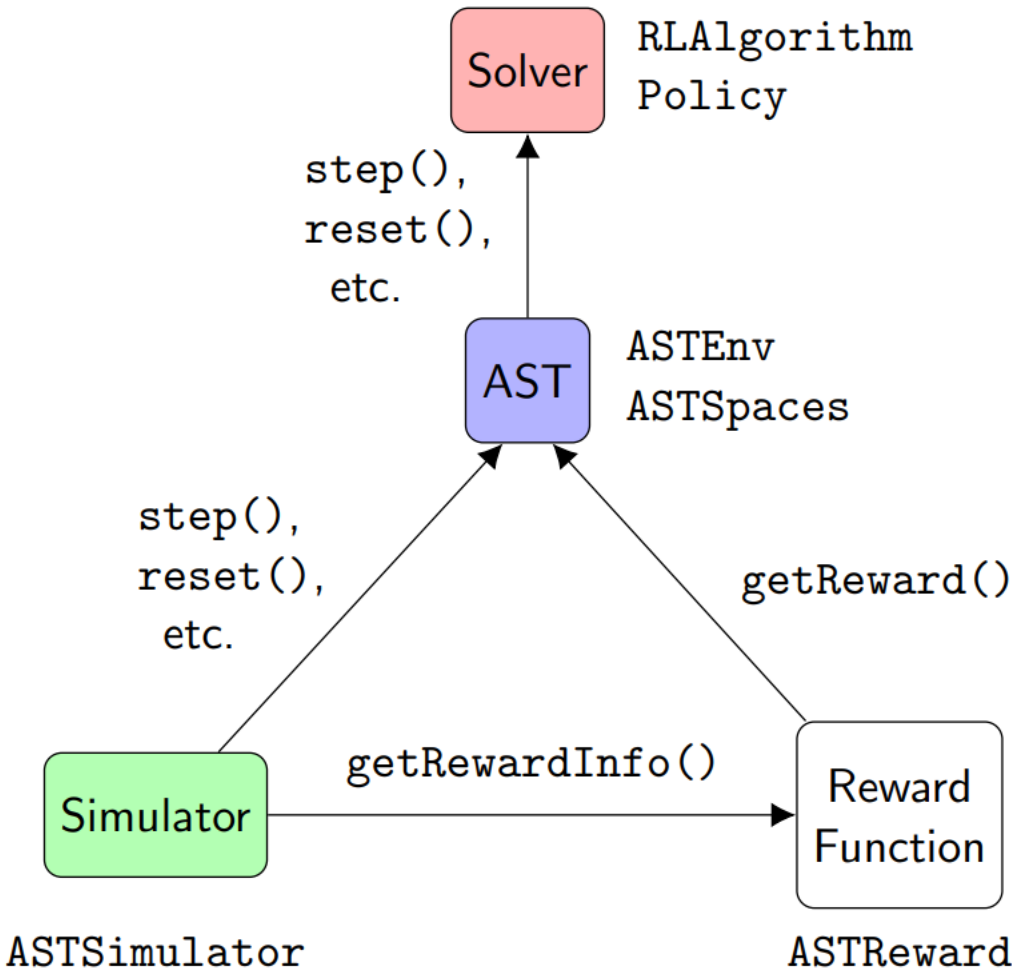

Fig. 2: The AST Toolbox framework architecture.

must define their state and action spaces as an ASTSpaces class, define their reward function as an ASTReward class, and provide the ASTEnv control of the simulator through the ASTSimulator class. Solvers are built on the Garage framework.

We have created a *tutorial* to show users how to use the Toolbox to validate an autonomous policy.

CHAPTER 6

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

6.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.2 Documentation improvements

AdaptiveStressTestingToolbox could always use more documentation, whether as part of the official AdaptiveStressTestingToolbox docs, in docstrings, or even on the web in blog posts, articles, and such.

6.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/sisl/AdaptiveStressTestingToolbox/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

6.4 Development

To set up *AdaptiveStressTestingToolbox* for local development:

1. Fork AdaptiveStressTestingToolbox (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@https://github.com:sisl/AdaptiveStressTestingToolbox.git
```

3. Follow the Git Installation
3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes run all the checks and docs builder with tox. See the Testing and Documenting sections for more details.
5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

6.4.1 Testing

We use Travis+Tox to test the Toolbox, and your PR will not be approved if the tests fail, or if the code coverage would drop too low. To avoid this, use tox to test on your local machine.

First, make sure you have all the testig dependencies:

```
pip install -r ci/requirements.txt
```

From the main folder, you can run all tests with verbose output with the following command:

```
tox -v
```

If you only want to check one of the Tox test environments, you can specify which one to run:

```
tox -v -e [environment_name]
```

There are 5 tox environments that are run during the full test:

1. **clean** - Cleans unneeded files from previous tests/development to prepare for testing.
2. **check** - Enforces code formatting. Checks are run using the check-manifest, flake8, and isort packages. You can run the check-autofix tox environment beforehand to fix most issues.
3. **docs** - Builds and checks the documentation.
4. **py36-cover** - Runs the code tests using pytest and codecov.
5. **report** - Reports the code coverage of the previous tests.

6.4.2 Documentation

The primary form of documentation for the Toolbox is numpy-style docstrings within the code. We use these to automatically generate online documentation. If you are changing or adding files, make sure the docstrings are up-to-date.

First, make sure you have the documentation dependencies:

```
pip install -r docs/requirements.txt
```

Some docstring guidelines:

- Make the descriptions as explanatory as possible.
- If the parameter has a default value, indicate this by adding "optional" to the type
- If the type of a parameter is a non-python class (for example, a class from Garage or from elsewhere in the Toolbox), make the type link to that class's documentation. You can do this using intersphinx.

For example, to link to a garage class, we first added:

```
'garage': ('https://garage.readthedocs.io/en/v2019.10.1/', None)
```

to the *intersphinx_mapping* settings in *docs/source/conf.py*. We can then link to a class with the following syntax:

```
:domain:'[text to show] <intersphinx_mapping:location>'
```

For example, for the garage.experiment.LocalRunner class, we would link using:

```
:py:class:`garage.experiment.LocalRunner <garage:garage.experiment.LocalRunner>`
```

Note that some links will use a different domains . The correct domains and locations can be a bit tricky to find. I recommend using the sphobjinv package . For example, we could have run the following command from the terminal to find the correct link syntax:

```
sphobjinv suggest -siu https://garage.readthedocs.io/en/v2019.10.1/objects.inv␣
↪LocalRunner
```

Once you have updated all of the docstrings, run the following commands from the *docs* folder to update the documentation source and generate a local HTML version for inspection:

```
sphinx-apidoc -o ./source/_apidoc ../src/ast_toolbox -eMf
make clean
make html
```

6.4.3 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

CHAPTER 7

Authors

- Stanford Intelligent Systems Laboratory - http://sisl.stanford.edu/

CHAPTER 8

Changelog

8.1 2020.06.01.dev1 (2020-05-17)

- First release on PyPI.

8.2 2020.09.01.dev1 (2020-09-01)

- Update documentation.
- Added docstrings and full apidocs.
- Fix for Backward Algorithm.
- Separate the toy AV simulator from the AST wrapper.
- Change AST environments to save the cloned sim state from pre-action, not post-action.
- Add travis deployment to PyPI.
- Removal of unsupported files.
- Expanded codecov to 90+%.

CHAPTER 9

ast_toolbox

9.1 ast_toolbox package

AST-Toolbox Base

ast_toolbox.**register**(*id*, *entry_point*, *force=True*, ***kwargs*)

9.1.1 Subpackages

ast_toolbox.algos package

Algorithms for solving AST formulated RL problems.

class ast_toolbox.algos.**GA**(*top_paths=None*, *n_itr=2*, *batch_size=500*, *step_size=0.01*, *step_size_anneal=1.0*, *pop_size=5*, *truncation_size=2*, *keep_best=1*, *f_F='mean'*, *log_interval=4000*, *init_step=1.0*, ***kwargs*)

Bases: garage.tf.algos.batch_polopt.BatchPolopt

Deep Genetic Algorithm from Such et al. [1]_.

Parameters

- **top_paths** (*ast_toolbox.mcts.BoundedPriorityQueues*, optional) – The bounded priority queue to store top-rewarded trajectories.
- **step_size** (*float, optional*) – Standard deviation for each mutation.
- **step_size_anneal** (*float, optional*) – The linear annealing rate of step_size after each iteration.
- **pop_size** (*int, optional*) – The population size
- **truncation_size** (*int, optional*) – The number of top-performed individuals that are chosen as parents.

- **keep_best** (*int, optional*) – The number of top-performed individuals that remain unchanged for next generation.
- **f_F** (*string, optional*) – The method used to calculate fitness: 'mean' for the average return, 'max' for the max return.
- **log_interval** (*int, optional*) – The log interval in terms of environment calls.
- **kwargs** – Keyword arguments passed to garage.tf.algos.BatchPolopt.

References

alternative for training deep neural networks for reinforcement learning." arXiv:1712.06567 (2017).

extra_recording(*itr*)
Record extra training statistics per-iteration.

Parameters **itr** (*int*) – The iteration number.

get_fitness(*itr*, *all_paths*)
Calculate the fitness of the collexted paths.

Parameters

- **itr** (*int*) – The iteration number.
- **all_paths** (*list[dict]*) – The collected paths from the sampler.

Returns **fitness** (*list[float]*) – The list of fitness of each individual.

get_itr_snapshot(*itr*, *samples_data*)
Get the snapshot of the current population.

Parameters

- **itr** (*int*) – The iteration number.
- **samples_data** (*dict*) – The processed data samples.

Returns **snaposhot** (*dict*) – The training snapshot.

init_opt()
Initiate trainer internal tensorflow operations.

initial()
Initiate trainer internal parameters.

mutation(*itr*, *new_seeds*, *new_magnitudes*, *all_paths*)
Generate new random seeds and magnitudes for the next generation.

The first self.keep_best seeds are set to no-mutation value (0).

Parameters

- **itr** (*int*) – The iteration number.
- **new_seeds** (`numpy.ndarry`) – The original seeds.
- **new_magnitudes** (`numpy.ndarry`) – The original magnitudes.
- **all_paths** (*list[dict]*) – The collected paths from the sampler.

Returns

- **new_seeds** (`numpy.ndarry`) – The new seeds.

- **new_magnitudes** (`numpy.ndarry`) – The new magnitudes.

obtain_samples (*itr*, *runner*)
Collect rollout samples using the current policy paramter.

Parameters

- **itr** (*int*) – The iteration number.
- **runner** (`garage.experiment.LocalRunner`) – `LocalRunner` is passed to give algorithm the access to `runner.obtain_samples()`, which collects rollout paths from the sampler.

Returns **paths** (*list[dict]*) – The collected paths from the sampler.

optimize_policy (*itr*, *all_paths*)
Update the population represented by self.seeds and self.parents.

Parameters

- **itr** (*int*) – The iteration number.
- **all_paths** (*list[dict]*) – The collected paths from the sampler.

process_samples (*itr*, *paths*)
Return processed sample data based on the collected paths.

Parameters

- **itr** (*int*) – The iteration number.
- **paths** (*list[dict]*) – The collected paths from the sampler.

Returns **samples_data** (*dict*) – Processed sample data with same trajectory length (padded with 0)

record_tabular (*itr*)
Record training performace per-iteration.

Parameters **itr** (*int*) – The iteration number.

select_parents (*fitness*)
Select the individuals to be the parents of the next generation.

Parameters **fitness** (*list[float]*) – The list of fitness of each individual.

set_params (*itr*, *p*)
Set the current policy paramter to the specified iteration and individual.

Parameters

- **itr** (*int*) – The iteration number.
- **p** (*int*) – The individual index.

train (*runner*)
Start training.

Parameters **runner** (`garage.experiment.LocalRunner`) – `LocalRunner` is passed to give algorithm the access to `runner.step_epochs()`, which provides services such as snapshotting and sampler control.

class ast_toolbox.algos.**GASM**(*step_size=0.01*, ***kwargs*)
Bases: *ast_toolbox.algos.ga.GA*

Deep Genetic Algorithm [1]_ with Safe Mutation [2]_.

Parameters

- **step_size** (*float, optional*) – The constraint on the KL divergence of each mutation.
- **kwargs** – Keyword arguments passed to *ast_toolbox.algos.ga.GA*.

References

training deep neural networks for reinforcement learning." arXiv preprint arXiv:1712.06567 (2017).

data2inputs(*samples_data*)
Transfer the processed data samples to training inputs

Parameters **samples_data** (*dict*) – The processed data samples

Returns **all_input_values** (*tuple*) – The input used in training

extra_recording(*itr*)
Record extra training statistics per-iteration.

Parameters **itr** (*int*) – The iteration number.

init_opt()
Initiate trainer internal tensorflow operations.

mutation(*itr*, *new_seeds*, *new_magnitudes*, *all_paths*)
Generate new random seeds and magnitudes for the next generation.

The first self.keep_best seeds are set to no-mutation value (0).

Parameters

- **itr** (*int*) – The iteration number.
- **new_seeds** (`numpy.ndarry`) – The original seeds.
- **new_magnitudes** (`numpy.ndarry`) – The original magnitudes.
- **all_paths** (*list[dict]*) – The collected paths from the sampler.

Returns

- **new_seeds** (`numpy.ndarry`) – The new seeds.
- **new_magnitudes** (`numpy.ndarry`) – The new magnitudes.

class ast_toolbox.algos.**MCTS**(*env*, *max_path_length*, *ec*, *n_itr*, *k*, *alpha*, *clear_nodes*, *log_interval*, *top_paths*, *log_dir*, *gamma=1.0*, *stress_test_mode=2*, *log_tabular=True*, *plot_tree=False*, *plot_path=None*, *plot_format='png'*)

Bases: `object`

Monte Carlo Tress Search (MCTS) with double progressive widening (DPW) [1]_ using the env's action space as its action space.

Parameters

- **env** (*ast_toolbox.envs.go_explore_ast_env.GoExploreASTEnv*.) – The environment.
- **max_path_length** (*int*) – The maximum search depth.
- **ec** (*float*) – The exploration constant used in UCT equation.

- **n_itr** (*int*) – The iteration number, the total numeber of environment call is approximately n_itr*max_path_length*max_path_length.
- **k** (*float*) – The constraint parameter used in DPW: |N(s,a)|<=kN(s)^alpha.
- **alpha** (*float*) – The constraint parameter used in DPW: |N(s,a)|<=kN(s)^alpha.
- **clear_nodes** (*bool*) – Whether to clear redundant nodes in tree. Set it to True for saving memoray. Set it to False to better tree plotting.
- **log_interval** (*int*) – The log interval in terms of environment calls.
- **top_paths** (*ast_toolbox.mcts.BoundedPriorityQueues*, optional) – The bounded priority queue to store top-rewarded trajectories.
- **gamma** (*float, optional*) – The discount factor.
- **stress_test_mode** (*int, optional*) – The mode of the tree search. 1 for single tree. 2 for multiple trees.
- **log_tabular** (*bool, optional*) – Whether to log the training statistics into a tabular file.
- **plot_tree** (*bool, optional*) – Whether to plot the resulting searching tree.
- **plot_path** (*str, optional*) – The storing path for the tree plot.
- **plot_format** (*str, optional*) – The storing format for the tree plot

References

init()
Initiate AST internal parameters

train(*runner*)
Start training.

Parameters **runner** (*garage.experiment.LocalRunner*) – LocalRunner is passed to give algorithm the access to runner.step_epochs(), which provides services such as snapshotting and sampler control.

class ast_toolbox.algos.**MCTSBV**(*M=10*, ***kwargs*)
Bases: *ast_toolbox.algos.mcts.MCTS*

Monte Carlo Tress Search (MCTS) with double progressive widening (DPW) [1]_ using Blind Value search from Couetoux et al. [2]_.

Parameters

- **M** (*int, optional*) – The number of randon decisions generated for the action pool.
- **kwargs** – Keyword arguments passed to *ast_toolbox.algos.mcts.MCTS*.

References

init()
Initiate AST internal parameters

class ast_toolbox.algos.**MCTSRS**(*seed=0*, *rsg_length=1*, ***kwargs*)
Bases: *ast_toolbox.algos.mcts.MCTS*

Monte Carlo Tress Search (MCTS) with double progressive widening (DPW) [1]_ using the random seeds as its action space.

Parameters

- **seed** (*int, optional*) – The seed used to generate the initial random seed generator.
- **rsg_length** (*int, optional*) – The length of the state of the random seed generator. Set it to higher values for extreme large problems.

References

init()
Initiate AST internal parameters

class ast_toolbox.algos.**GoExplore**(*db_filename*, *max_db_size*, *env*, *env_spec*, *policy*, *baseline*, *save_paths_gap=0*, *save_paths_path=None*, *overwrite_db=True*, *use_score_weight=True*, ***kwargs*)
Bases: garage.tf.algos.batch_polopt.BatchPolopt

Implementation of the Go-Explore[1]_ algorithm that is compatible with AST[2]_. :Parameters: * **db_filename** (*str*) – The base path and name for the database files. The CellPool saves a *[filename]_pool.dat* and a *[filename]_meta.dat*.

- **max_db_size** (*int*) – Maximum allowable size (in GB) of the CellPool database. Algorithm will immediately stop and exit if this size is exceeded.
- **env** (*ast_toolbox.envs.go_explore_ast_env.GoExploreASTEnv*) – The environment.
- **env_spec** (garage.envs.EnvSpec) – Environment specification.
- **policy** (garage.tf.policies.Policy) – The policy.
- **baseline** (garage.np.baselines.Baseline) – The baseline.
- **save_paths_gap** (*int, optional*) – How many epochs to skip between saving out full paths. Set to *1* to save every epoch. Set to *0* to disable saving.
- **save_paths_path** (*str, optional*) – Path to the directory where paths should be saved. Set to *None* to disable saving.
- **overwrite_db** (*bool, optional*) – Indicates if an existing database should be overwritten if found.
- **use_score_weight** (*bool*) – Whether or not to scale the cell's fitness by a function of the cell's score
- **kwargs** – Keyword arguments passed to garage.tf.algos.BatchPolopt

References

downsample(*obs*, *step=None*)
Create a downsampled approximation of the observed simulation state.

Parameters

- **obs** (*array_like*) – The observed simulation state.
- **step** (*int, optional*) – The current iteration number

Returns *array_like* – The downsampled approximation of the observed simulation state.

get_itr_snapshot(*itr*)
Returns all the data that should be saved in the snapshot for this iteration.

Parameters **itr** (*int*) – The current epoch number.

Returns *dict* – A dict containing the current iteration number, the current policy, and the current baseline.

init_opt()
Initialize the optimization procedure. If using tensorflow, this may include declaring all the variables and compiling functions

optimize_policy(*itr*, *samples_data*)
Optimize the policy using the samples.

Parameters

- **itr** (*int*) – The current epoch number.
- **samples_data** (*dict*) – The data from the sampled rollouts.

train(*runner*)
Obtain samplers and start actual training for each epoch.

Parameters **runner** (garage.experiment.LocalRunner) – LocalRunner is passed to give algorithm the access to runner.step_epochs(), which provides services such as snapshotting and sampler control.

Returns **last_return** (*ast_toolbox.algos.go_explore.Cell*) – The highest scoring cell found so far

train_once(*itr*, *paths*)
Perform one step of policy optimization given one batch of samples.

Parameters

- **itr** (*int*) – Iteration number.
- **paths** (*list[dict]*) – A list of collected paths.

Returns **best_cell** (*ast_toolbox.algos.go_explore.Cell*) – The highest scoring cell found so far

class ast_toolbox.algos.**BackwardAlgorithm**(*env*, *policy*, *expert_trajectory*, *epochs_per_step=10*, *max_epochs=None*, *skip_until_step=0*, *max_path_length=500*, ***kwargs*)

Bases: garage.tf.algos.ppo.PPO

Backward Algorithm from Salimans and Chen [1]_.

Parameters

- **env** (*ast_toolbox.envs.go_explore_ast_env.GoExploreASTEnv*) – The environment.
- **policy** (garage.tf.policies.Policy) – The policy.
- **expert_trajectory** (*array_like[dict]*) – The expert trajectory, an array_like where each member represents a timestep in a trajectory. The array_like should be 1-D and in chronological order. Each member of the array_like is a dictionary with the following keys:
  - state: The simulator state at that timestep (pre-action).
  - reward: The reward at that timestep (post-action).
  - observation: The simulation observation at that timestep (post-action).
  - action: The action taken at that timestep.

- **epochs_per_step** (*int, optional*) – Maximum number of epochs to run per step of the trajectory.
- **max_epochs** (*int, optional*) – Maximum number of total epochs to run. If not set, defaults to `epochs_per_step` times the number of steps in the `expert_trajectory`.
- **skip_until_step** (*int, optional*) – Skip training for a certain number of steps at the start, counted backwards from the end of the trajectory. For example, if this is set to 3 for an `expert_trajectory` of length 10, training will start from step 7.
- **max_path_length** (*int, optional*) – Maximum length of a single rollout.
- **kwargs** – Keyword arguments passed to garage.tf.algos.PPO

References

get_next_epoch(*runner*)
Wrapper of garage's `runner.step_epochs()` generator to handle initialization to correct trajectory state

Parameters **runner** (`garage.experiment.LocalRunner`) – `LocalRunner` is passed to give algorithm the access to `runner.step_epochs()`, which provides services such as snapshotting and sampler control.

Yields

- **runner.step_itr** (*int*) – The current epoch number.
- **runner.obtain_samples(runner.step_itr)** (*list[dict]*) – A list of sampled rollouts for the current epoch

set_env_to_expert_trajectory_step()
Updates the algorithm to use the data from `expert_trajectory` up to the current step.

train(*runner*)
Obtain samplers and start actual training for each epoch.

Parameters **runner** (`garage.experiment.LocalRunner`) – `LocalRunner` is passed to give algorithm the access to `runner.step_epochs()`, which provides services such as snapshotting and sampler control.

Returns **full_paths** (*array_like*) – A list of the path data from each epoch.

train_once(*itr*, *paths*)
Perform one step of policy optimization given one batch of samples.

Parameters

- **itr** (*int*) – Iteration number.
- **paths** (*list[dict]*) – A list of collected paths.

Returns **paths** (*list[dict]*) – A list of processed paths

Submodules

ast_toolbox.algos.backward_algorithm module

Backward Algorithm from Salimans and Chen.

class ast_toolbox.algos.backward_algorithm.**BackwardAlgorithm**(*env*, *policy*, *expert_trajectory*, *epochs_per_step=10*, *max_epochs=None*, *skip_until_step=0*, *max_path_length=500*, ***kwargs*)

Bases: garage.tf.algos.ppo.PPO

Backward Algorithm from Salimans and Chen[1].

Parameters

- **env** (*ast_toolbox.envs.go_explore_ast_env.GoExploreASTEnv*) – The environment.
- **policy** (garage.tf.policies.Policy) – The policy.
- **expert_trajectory** (*array_like[dict]*) – The expert trajectory, an array_like where each member represents a timestep in a trajectory. The array_like should be 1-D and in chronological order. Each member of the array_like is a dictionary with the following keys:
  - state: The simulator state at that timestep (pre-action).
  - reward: The reward at that timestep (post-action).
  - observation: The simulation observation at that timestep (post-action).
  - action: The action taken at that timestep.
- **epochs_per_step** (*int, optional*) – Maximum number of epochs to run per step of the trajectory.
- **max_epochs** (*int, optional*) – Maximum number of total epochs to run. If not set, defaults to `epochs_per_step` times the number of steps in the `expert_trajectory`.
- **skip_until_step** (*int, optional*) – Skip training for a certain number of steps at the start, counted backwards from the end of the trajectory. For example, if this is set to 3 for an `expert_trajectory` of length 10, training will start from step 7.
- **max_path_length** (*int, optional*) – Maximum length of a single rollout.
- **kwargs** – Keyword arguments passed to garage.tf.algos.PPO

References

get_next_epoch(*runner*)

Wrapper of garage's `runner.step_epochs()` generator to handle initialization to correct trajectory state

Parameters **runner** (garage.experiment.LocalRunner) – `LocalRunner` is passed to give algorithm the access to `runner.step_epochs()`, which provides services such as snapshotting and sampler control.

Yields

- **runner.step_itr** (*int*) – The current epoch number.
- **runner.obtain_samples(runner.step_itr)** (*list[dict]*) – A list of sampled rollouts for the current epoch

[1] Salimans, Tim, and Richard Chen. "Learning Montezuma's Revenge from a Single Demonstration." arXiv preprint arXiv:1812.03381 (2018). https://arxiv.org/abs/1812.03381

set_env_to_expert_trajectory_step()
Updates the algorithm to use the data from `expert_trajectory` up to the current step.

train(*runner*)
Obtain samplers and start actual training for each epoch.

Parameters **runner** (`garage.experiment.LocalRunner`) – `LocalRunner` is passed to give algorithm the access to `runner.step_epochs()`, which provides services such as snapshotting and sampler control.

Returns **full_paths** (*array_like*) – A list of the path data from each epoch.

train_once(*itr*, *paths*)
Perform one step of policy optimization given one batch of samples.

Parameters

- **itr** (*int*) – Iteration number.
- **paths** (*list[dict]*) – A list of collected paths.

Returns **paths** (*list[dict]*) – A list of processed paths

ast_toolbox.algos.ga module

class ast_toolbox.algos.ga.**GA**(*top_paths=None*, *n_itr=2*, *batch_size=500*, *step_size=0.01*, *step_size_anneal=1.0*, *pop_size=5*, *truncation_size=2*, *keep_best=1*, *f_F='mean'*, *log_interval=4000*, *init_step=1.0*, ***kwargs*)
Bases: `garage.tf.algos.batch_polopt.BatchPolopt`

Deep Genetic Algorithm from Such et al.[1].

Parameters

- **top_paths** (*ast_toolbox.mcts.BoundedPriorityQueues*, optional) – The bounded priority queue to store top-rewarded trajectories.
- **step_size** (*float, optional*) – Standard deviation for each mutation.
- **step_size_anneal** (*float, optional*) – The linear annealing rate of step_size after each iteration.
- **pop_size** (*int, optional*) – The population size
- **truncation_size** (*int, optional*) – The number of top-performed individuals that are chosen as parents.
- **keep_best** (*int, optional*) – The number of top-performed individuals that remain unchanged for next generation.
- **f_F** (*string, optional*) – The method used to calculate fitness: 'mean' for the average return, 'max' for the max return.
- **log_interval** (*int, optional*) – The log interval in terms of environment calls.
- **kwargs** – Keyword arguments passed to garage.tf.algos.BatchPolopt.

[1] Such, Felipe Petroski, et al. "Deep neuroevolution: Genetic algorithms are a competitive

References

alternative for training deep neural networks for reinforcement learning." arXiv:1712.06567 (2017).

extra_recording(*itr*)
Record extra training statistics per-iteration.

Parameters **itr** (*int*) – The iteration number.

get_fitness(*itr*, *all_paths*)
Calculate the fitness of the collexted paths.

Parameters

- **itr** (*int*) – The iteration number.
- **all_paths** (*list[dict]*) – The collected paths from the sampler.

Returns **fitness** (*list[float]*) – The list of fitness of each individual.

get_itr_snapshot(*itr*, *samples_data*)
Get the snapshot of the current population.

Parameters

- **itr** (*int*) – The iteration number.
- **samples_data** (*dict*) – The processed data samples.

Returns **snaposhot** (*dict*) – The training snapshot.

init_opt()
Initiate trainer internal tensorflow operations.

initial()
Initiate trainer internal parameters.

mutation(*itr*, *new_seeds*, *new_magnitudes*, *all_paths*)
Generate new random seeds and magnitudes for the next generation.

The first self.keep_best seeds are set to no-mutation value (0).

Parameters

- **itr** (*int*) – The iteration number.
- **new_seeds** (`numpy.ndarry`) – The original seeds.
- **new_magnitudes** (`numpy.ndarry`) – The original magnitudes.
- **all_paths** (*list[dict]*) – The collected paths from the sampler.

Returns

- **new_seeds** (`numpy.ndarry`) – The new seeds.
- **new_magnitudes** (`numpy.ndarry`) – The new magnitudes.

obtain_samples(*itr*, *runner*)
Collect rollout samples using the current policy paramter.

Parameters

- **itr** (*int*) – The iteration number.

- **runner** (garage.experiment.LocalRunner) – LocalRunner is passed to give algorithm the access to runner.obtain_samples(), which collects rollout paths from the sampler.

Returns **paths** (*list[dict]*) – The collected paths from the sampler.

optimize_policy(*itr*, *all_paths*)
Update the population represented by self.seeds and self.parents.

Parameters

- **itr** (*int*) – The iteration number.
- **all_paths** (*list[dict]*) – The collected paths from the sampler.

process_samples(*itr*, *paths*)
Return processed sample data based on the collected paths.

Parameters

- **itr** (*int*) – The iteration number.
- **paths** (*list[dict]*) – The collected paths from the sampler.

Returns **samples_data** (*dict*) – Processed sample data with same trajectory length (padded with 0)

record_tabular(*itr*)
Record training performace per-iteration.

Parameters **itr** (*int*) – The iteration number.

select_parents(*fitness*)
Select the individuals to be the parents of the next generation.

Parameters **fitness** (*list[float]*) – The list of fitness of each individual.

set_params(*itr*, *p*)
Set the current policy paramter to the specified iteration and individual.

Parameters

- **itr** (*int*) – The iteration number.
- **p** (*int*) – The individual index.

train(*runner*)
Start training.

Parameters **runner** (garage.experiment.LocalRunner) – LocalRunner is passed to give algorithm the access to runner.step_epochs(), which provides services such as snapshotting and sampler control.

ast_toolbox.algos.gasm module

class ast_toolbox.algos.gasm.**GASM**(*step_size=0.01*, ***kwargs*)
Bases: *ast_toolbox.algos.ga.GA*

Deep Genetic Algorithm[1] with Safe Mutation[2].

Parameters

[1] Such, Felipe Petroski, et al. "Deep neuroevolution: Genetic algorithms are a competitive alternative for

[2] Lehman, Joel, et al. "Safe mutations for deep and recurrent neural networks through output gradients." Proceedings of the Genetic and Evolutionary Computation Conference. 2018.

- **step_size** (*float, optional*) – The constraint on the KL divergence of each mutation.
- **kwargs** – Keyword arguments passed to *ast_toolbox.algos.ga.GA*.

References

training deep neural networks for reinforcement learning." arXiv preprint arXiv:1712.06567 (2017).

data2inputs(*samples_data*)
Transfer the processed data samples to training inputs

Parameters **samples_data** (*dict*) – The processed data samples

Returns **all_input_values** (*tuple*) – The input used in training

extra_recording(*itr*)
Record extra training statistics per-iteration.

Parameters **itr** (*int*) – The iteration number.

init_opt()
Initiate trainer internal tensorflow operations.

mutation(*itr*, *new_seeds*, *new_magnitudes*, *all_paths*)
Generate new random seeds and magnitudes for the next generation.

The first self.keep_best seeds are set to no-mutation value (0).

Parameters

- **itr** (*int*) – The iteration number.
- **new_seeds** (`numpy.ndarry`) – The original seeds.
- **new_magnitudes** (`numpy.ndarry`) – The original magnitudes.
- **all_paths** (*list[dict]*) – The collected paths from the sampler.

Returns

- **new_seeds** (`numpy.ndarry`) – The new seeds.
- **new_magnitudes** (`numpy.ndarry`) – The new magnitudes.

ast_toolbox.algos.go_explore module

Implementation of the Go-Explore algorithm.

class ast_toolbox.algos.go_explore.**Cell**(*use_score_weight=True*)
Bases: `object`

A representation of a state visited during exploration.

Parameters **use_score_weight** (*bool*) – Whether or not to scale the cell's fitness by a function of the cell's score

reset_cached_property(*cached_property*)
Removes cached properties so they will be recalculated on next access.

Parameters **cached_property** (*str*) – The cached_property key to remove from the class dict.

count_subscores
A function of *times_chosen_subscore*, *times_chosen_since_improved_subscore*, and *times_visited_subscore* that is used in calculating the cell's *fitness* score.

Returns *float* – The count subscore of the cell.

fitness
The *fitness* score of the cell. Cells are sampled with probability proportional to their *fitness* score.

Returns *float* – The fitness score of the cell.

is_goal
Whether or not the current cell is a goal state.

Returns *bool* – Is the current cell a goal.

is_root
Checks if the cell is the root of the tree (trajectory length is 0).

Returns *bool* – Whether the cell is root or not

is_terminal
Whether or not the current cell is a terminal state.

Returns *bool* – Is the current cell terminal.

reward
The reward obtained in the current cell.

Returns *float* – The reward.

score
The *score* obtained in the current cell.

Returns *float* – The score.

score_weight
A heuristic function basedon the cell's score, and other values, to bias the rollouts towards high-scoring areas.

Returns *float* – The cell's *score_weight*

step
How many steps led to the current cell.

Returns *int* – Length of the trajectory.

times_chosen
How many times the current cell has been chosen to start a rollout.

Returns *int* – Number of times chosen.

times_chosen_since_improved
How many times the current cell has been chosen to start a rollout since the last time the cell was updated with an improved score or trajectory.

Returns *int* – Number of times chosen since last improved.

times_chosen_since_improved_subscore
A function of *times_chosen_since_improved* that is used in calculating the cell's *times_chosen_since_improved_subscore* score.

Returns *float* – The *times_chosen_since_improved_subscore*

times_chosen_subscore
A function of *times_chosen* that is used in calculating the cell's *times_chosen_subscore* score.

Returns *float* – The *times_chosen_subscore*

times_visited
How many times the current cell has been visited during all rollouts.

Returns *int* – Number of times visited.

times_visited_subscore
A function of *_times_visited* that is used in calculating the cell's *times_visited_subscore* score.

Returns *float* – The *times_visited_subscore*

value_approx
The approximate value of the current cell, based on backpropigation of previous rollouts.

Returns *float* – The value approximation.

class ast_toolbox.algos.go_explore.**CellPool**(*filename='database'*, *discount=0.99*, *use_score_weight=True*)

Bases: object

A hashtree data structure containing and updating all of the cells seen during rollouts.

Parameters

- **filename** (*str, optional*) – The base name for the database files. The CellPool saves a *[filename]_pool.dat* and a *[filename]_meta.dat*.
- **discount** (*float, optional*) – Discount factor used in calculating a cell's value approximation.
- **use_score_weight** (*bool*) – Whether or not to scale a cell's fitness by a function of the cell's score

close_pool(*cell_pool_shelf*)
Close the database that the CellPool uses to store cells.

Parameters **cell_pool_shelf** (shelve.Shelf) – A *shelve.Shelf* wrapping a bsddb3 database.

d_update(*cell_pool_shelf*, *observation*, *action*, *trajectory*, *score*, *state*, *parent=None*, *is_terminal=False*, *is_goal=False*, *reward=-inf*, *chosen=0*)
Runs the update algorithm for the CellPool. The process is: 1. Create a cell from the given data. 2. Check if the cell already exists in the CellPool. 3. If the cell already exists and our version is better (higher fitness or shorter trajectory), update the existing cell. 4. If the cell already exists and our version is not better, end. 5. If the cell does not already exists, add the new cell to the CellPool

Parameters

- **cell_pool_shelf** (shelve.Shelf) – A *shelve.Shelf* wrapping a bsddb3 database.
- **observation** (*array_like*) – The observation seen in the current cell.
- **action** (*array_like*) – The action taken in the current cell.
- **trajectory** (*array_like*) – The trajectory leading to the current cell.
- **score** (*float*) – The score at the current cell.
- **state** (*array_like*) – The cloned simulation state at the current cell, used for resetting if chosen to start a rollout.
- **parent** (*int, optional*) – The hash key of the cell immediately preceding the current cell in the trajectory.
- **is_terminal** (*bool, optional*) – Whether the current cell is a terminal state.
- **is_goal** (*bool, optional*) – Whether the current cell is a goal state.

- **reward** (*float, optional*) – The reward obtained at the current cell.
- **chosen** (*int, optional*) – Whether the current cell was chosen to start the rollout.

Returns *bool* – True if a new cell was added to the CellPool, False otherwise

delete_pool()
Remove the CellPool files saved on disk.

load(*cell_pool_shelf*)
Load a CellPool from disk.

Parameters **cell_pool_shelf** (shelve.Shelf) – A *shelve.Shelf* wrapping a bsddb3 database.

open_pool(*dbname=None*, *dbtype=<sphinx.ext.autodoc.importer._MockObject object>*, *flags=<sphinx.ext.autodoc.importer._MockObject object>*, *protocol=4*, *overwrite=False*)
Open the database that the CellPool uses to store cells.

Parameters

- **dbname** (*string*)
- **dbtype** (*int, optional*) – Specifies the type of database to open. Use enumerations provided by bsddb3.
- **flags** (*int, optional*) – Specifies the configuration of the database to open. Use enumerations provided by bsddb3.
- **protocol** (*int, optional*) – Specifies the data stream format used by pickle.
- **overwrite** (*bool, optional*) – Indicates if an existing database should be overwritten if found.

Returns

cell_pool_shelf (shelve.Shelf) – A *shelve.Shelf* wrapping a bsddb3 database.

save()
Save the CellPool to disk.

sync_and_close_pool(*cell_pool_shelf*)
Sync and then close the database that the CellPool uses to store cells.

Parameters **cell_pool_shelf** (shelve.Shelf) – A *shelve.Shelf* wrapping a bsddb3 database

sync_pool(*cell_pool_shelf*)
Syncs the pool, ensuring that the database on disk is up-to-date.

Parameters **cell_pool_shelf** (shelve.Shelf) – A *shelve.Shelf* wrapping a bsddb3 database.

value_approx_update(*value*, *obs_hash*, *cell_pool_shelf*)
Recursively calculate a value approximation through back-propagation.

Parameters

- **value** (*Value approximation of the previous cell.*)
- **obs_hash** (*Hash key of the current cell.*)
- **cell_pool_shelf** (shelve.Shelf) – A *shelve.Shelf* wrapping a bsddb3 database.

meta_filename
The CellPool metadata filename.

Returns *str* – The CellPool metadata filename.

pool_filename
The CellPool database filename.

Returns *str* – The CellPool database filename.

class ast_toolbox.algos.go_explore.**GoExplore**(*db_filename*, *max_db_size*, *env*, *env_spec*, *policy*, *baseline*, *save_paths_gap=0*, *save_paths_path=None*, *overwrite_db=True*, *use_score_weight=True*, ***kwargs*)

Bases: garage.tf.algos.batch_polopt.BatchPolopt

Implementation of the Go-Explore[1]_ algorithm that is compatible with AST[2]_. :Parameters: * **db_filename** (*str*) – The base path and name for the database files. The CellPool saves a *[filename]_pool.dat* and a *[filename]_meta.dat*.

- **max_db_size** (*int*) – Maximum allowable size (in GB) of the CellPool database. Algorithm will immediately stop and exit if this size is exceeded.
- **env** (*ast_toolbox.envs.go_explore_ast_env.GoExploreASTEnv*) – The environment.
- **env_spec** (garage.envs.EnvSpec) – Environment specification.
- **policy** (garage.tf.policies.Policy) – The policy.
- **baseline** (garage.np.baselines.Baseline) – The baseline.
- **save_paths_gap** (*int, optional*) – How many epochs to skip between saving out full paths. Set to *1* to save every epoch. Set to *0* to disable saving.
- **save_paths_path** (*str, optional*) – Path to the directory where paths should be saved. Set to *None* to disable saving.
- **overwrite_db** (*bool, optional*) – Indicates if an existing database should be overwritten if found.
- **use_score_weight** (*bool*) – Whether or not to scale the cell's fitness by a function of the cell's score
- **kwargs** – Keyword arguments passed to garage.tf.algos.BatchPolopt

References

downsample(*obs*, *step=None*)

Create a downsampled approximation of the observed simulation state.

Parameters

- **obs** (*array_like*) – The observed simulation state.
- **step** (*int, optional*) – The current iteration number

Returns *array_like* – The downsampled approximation of the observed simulation state.

get_itr_snapshot(*itr*)

Returns all the data that should be saved in the snapshot for this iteration.

Parameters **itr** (*int*) – The current epoch number.

Returns *dict* – A dict containing the current iteration number, the current policy, and the current baseline.

init_opt()

Initialize the optimization procedure. If using tensorflow, this may include declaring all the variables and compiling functions

optimize_policy(*itr*, *samples_data*)

Optimize the policy using the samples.

Parameters

- **itr** (*int*) – The current epoch number.
- **samples_data** (*dict*) – The data from the sampled rollouts.

train(*runner*)
Obtain samplers and start actual training for each epoch.

Parameters **runner** (`garage.experiment.LocalRunner`) – `LocalRunner` is passed to give algorithm the access to `runner.step_epochs()`, which provides services such as snapshotting and sampler control.

Returns **last_return** (*ast_toolbox.algos.go_explore.Cell*) – The highest scoring cell found so far

train_once(*itr*, *paths*)
Perform one step of policy optimization given one batch of samples.

Parameters

- **itr** (*int*) – Iteration number.
- **paths** (*list[dict]*) – A list of collected paths.

Returns **best_cell** (*ast_toolbox.algos.go_explore.Cell*) – The highest scoring cell found so far

ast_toolbox.algos.mcts module

class ast_toolbox.algos.mcts.**MCTS**(*env*, *max_path_length*, *ec*, *n_itr*, *k*, *alpha*, *clear_nodes*, *log_interval*, *top_paths*, *log_dir*, *gamma=1.0*, *stress_test_mode=2*, *log_tabular=True*, *plot_tree=False*, *plot_path=None*, *plot_format='png'*)
Bases: `object`

Monte Carlo Tress Search (MCTS) with double progressive widening (DPW)[1] using the env's action space as its action space.

Parameters

- **env** (*ast_toolbox.envs.go_explore_ast_env.GoExploreASTEnv*.) – The environment.
- **max_path_length** (*int*) – The maximum search depth.
- **ec** (*float*) – The exploration constant used in UCT equation.
- **n_itr** (*int*) – The iteration number, the total numeber of environment call is approximately n_itr*max_path_length*max_path_length.
- **k** (*float*) – The constraint parameter used in DPW: |N(s,a)|<=kN(s)^alpha.
- **alpha** (*float*) – The constraint parameter used in DPW: |N(s,a)|<=kN(s)^alpha.
- **clear_nodes** (*bool*) – Whether to clear redundant nodes in tree. Set it to True for saving memoray. Set it to False to better tree plotting.
- **log_interval** (*int*) – The log interval in terms of environment calls.

[1] Lee, Ritchie, et al. "Adaptive stress testing of airborne collision avoidance systems." 2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC). IEEE, 2015.

- **top_paths** (*ast_toolbox.mcts.BoundedPriorityQueues*, optional) – The bounded priority queue to store top-rewarded trajectories.
- **gamma** (*float, optional*) – The discount factor.
- **stress_test_mode** (*int, optional*) – The mode of the tree search. 1 for single tree. 2 for multiple trees.
- **log_tabular** (*bool, optional*) – Whether to log the training statistics into a tabular file.
- **plot_tree** (*bool, optional*) – Whether to plot the resulting searching tree.
- **plot_path** (*str, optional*) – The storing path for the tree plot.
- **plot_format** (*str, optional*) – The storing format for the tree plot

References

init()
Initiate AST internal parameters

train(*runner*)
Start training.

Parameters **runner** (garage.experiment.LocalRunner) – LocalRunner is passed to give algorithm the access to runner.step_epochs(), which provides services such as snapshotting and sampler control.

ast_toolbox.algos.mctsbv module

class ast_toolbox.algos.mctsbv.**MCTSBV**(*M=10*, ***kwargs*)
Bases: *ast_toolbox.algos.mcts.MCTS*

Monte Carlo Tress Search (MCTS) with double progressive widening (DPW)[1] using Blind Value search from Couetoux et al.[2].

Parameters

- **M** (*int, optional*) – The number of randon decisions generated for the action pool.
- **kwargs** – Keyword arguments passed to *ast_toolbox.algos.mcts.MCTS*.

References

init()
Initiate AST internal parameters

ast_toolbox.algos.mctsrs module

class ast_toolbox.algos.mctsrs.**MCTSRS**(*seed=0*, *rsg_length=1*, ***kwargs*)
Bases: *ast_toolbox.algos.mcts.MCTS*

[1] Lee, Ritchie, et al. "Adaptive stress testing of airborne collision avoidance systems." 2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC). IEEE, 2015.

[2] Couetoux, Adrien, Hassen Doghmen, and Olivier Teytaud. "Improving the exploration in upper confidence trees." International Conference on Learning and Intelligent Optimization. Springer, Berlin, Heidelberg, 2012.

Monte Carlo Tress Search (MCTS) with double progressive widening (DPW)[1] using the random seeds as its action space.

Parameters

- **seed** (*int, optional*) – The seed used to generate the initial random seed generator.
- **rsg_length** (*int, optional*) – The length of the state of the random seed generator. Set it to higher values for extreme large problems.

References

init()
Initiate AST internal parameters

ast_toolbox.envs package

Environments for formulating validation as an AST reinforcement learning problem.

class ast_toolbox.envs.**GoExploreASTEnv**(*open_loop=True*, *blackbox_sim_state=True*, *fixed_init_state=False*, *s_0=None*, *simulator=None*, *reward_function=None*, *spaces=None*)
Bases: gym.core.Env, *ast_toolbox.envs.go_explore_ast_env.Parameterized*

Gym environment to turn general AST tasks into garage compatible problems with Go-Explore style resets.

Certain algorithms, such as Go-Explore and the Backwards Algorithm, require deterministic resets of the simulator. *GoExploreASTEnv* handles this by cloning simulator states and saving them in a cell structure. The cells are then stored in a hashed database.

Parameters

- **open_loop** (*bool*) – True if the simulation is open-loop, meaning that AST must generate all actions ahead of time, instead of being able to output an action in sync with the simulator, getting an observation back before the next action is generated. False to get interactive control, which requires that *blackbox_sim_state* is also False.
- **blackbox_sim_state** (*bool*) – True if the true simulation state can not be observed, in which case actions and the initial conditions are used as the observation. False if the simulation state can be observed, in which case it will be used
- **fixed_init_state** (*bool*) – True if the initial state is fixed, False to sample the initial state for each rollout from the observaation space.
- **s_0** (*array_like*) – The initial state for the simulation (ignored if *fixed_init_state* is False)
- **simulator** (*ast_toolbox.simulators.ASTSimulator*) – The simulator wrapper, inheriting from *ast_toolbox.simulators.ASTSimulator*.
- **reward_function** (*ast_toolbox.rewards.ASTReward*) – The reward function, inheriting from *ast_toolbox.rewards.ASTReward*.
- **spaces** (*ast_toolbox.spaces.ASTSpaces*) – The observation and action space definitions, inheriting from *ast_toolbox.spaces.ASTSpaces*.

close()
Calls the simulator's *close* function, if it exists.

[1] Lee, Ritchie, et al. "Adaptive stress testing of airborne collision avoidance systems." 2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC). IEEE, 2015.

Returns *None or object* – Returns the output of the simulator's *close* function, or None if the simulator has no *close* function.

downsample(*obs*)
Create a downsampled approximation of the observed simulation state.

Parameters **obs** (*array_like*) – The observed simulation state.

Returns *array_like* – The downsampled approximation of the observed simulation state.

env_reset()
Resets the state of the environment, returning an initial observation.

Returns **observation** (*array_like*) – The initial observation of the space. (Initial reward is assumed to be 0.)

get_cache_list()
Returns the environment info cache.

Returns *dict* – A dictionary containing diagnostic and logging information for the environment.

get_first_cell()
Returns a the observation and state of the initial state, to be used for a root cell.

Returns

- **obs** (*array_like*) – Agent's observation of the current environment.
- **state** (*array_like*) – The cloned simulation state at the current cell, used for resetting if chosen to start a rollout.

get_param_values(**tags*)
Return the values of internal parameters.

Parameters **tags** (*dict[bool]*) – For each tag, a parameter is returned if the parameter name matches the tag's key

Returns *list* – A list of parameter values.

get_params_internal(**tags*)
Returns the parameters associated with the given tags.

Parameters **tags** (*dict[bool]*) – For each tag, a parameter is returned if the parameter name matches the tag's key

Returns *list* – List of parameters

log()
Calls the simulator's *log* function.

render(**kwargs*)
Calls the simulator's *render* function, if it exists.

Returns *None or object* – Returns the output of the simulator's *render* function, or None if the simulator has no *render* function.

reset(**kwargs*)
Resets the state of the environment, returning an initial observation.

The reset has 2 modes.

In the "robustify" mode (self.p_robustify_state.value is not None), the simulator resets the environment to *p_robustify_state.value*. It then returns the initial condition.

In the "Go-Explore" mode, the environment attempts to sample a cell from the cell pool. If successful, the simulator is reset to the cell's state. On an error, the environment is reset to the intial state.

Returns **observation** (*array_like*) – The initial observation of the space. (Initial reward is assumed to be 0.)

sample (*population*)

Sample a cell from the cell pool with likelihood proportional to cell fitness.

The sampling is done using Stochastic Acceptance[1], with inspiration from John B Nelson's blog [2]_.

The sampler rejects cells until the acceptance criterea is met. If the maximum number of rejections is exceeded, the sampler then will sample uniformly sample a cell until it finds a cell with fitness > 0. If the second sampling phase also exceeds the rejection limit, then the function raises an exception.

Parameters **population** (*list*) – A list containing the population of cells to sample from.

Returns *object* – The sampled cell.

Raises `ValueError` – If the maximum number of rejections is exceeded in both the proportional and the uniform sampling phases.

References

Physica A: Statistical Mechanics and its Applications 391.6 (2012): 2193-2196. https://arxiv.org/pdf/1109.3627.pdf .. [2] https://jbn.github.io/fast_proportional_selection/

set_param_values (*param_values*, ***tags*)

Set the values of parameters

Parameters

- **param_values** (*object*) – Value to set the parameter to.
- **tags** (*dict[bool]*) – For each tag, a parameter is returned if the parameter name matches the tag's key

simulate (*actions*)

Run a full simulation rollout.

Parameters **actions** (*list[array_like]*) – A list of array_likes, where each member is the action taken at that step.

Returns

- *int* – The step of the trajectory where a collision was found, or -1 if a collision was not found.
- *dict* – A dictionary of simulation information for logging and diagnostics.

step (*action*)

Run one timestep of the environment's dynamics. When end of episode is reached, reset() should be called to reset the environment's internal state.

Parameters **action** (*array_like*) – An action provided by the environment.

Returns

`garage.envs.base.Step()` – A step in the rollout. Contains the following information:

- observation (array_like): Agent's observation of the current environment.
- reward (float): Amount of reward due to the previous action.
- done (bool): Is the current step a terminal or goal state, ending the rollout.

[1] Lipowski, Adam, and Dorota Lipowska. "Roulette-wheel selection via stochastic acceptance."

- cache (dict): A dictionary containing other diagnostic information from the current step.
- actions (array_like): The action taken at the current.
- state (array_like): The cloned simulation state at the current cell, used for resetting if chosen to start a rollout.
- is_terminal (bool): Whether or not the current cell is a terminal state.
- is_goal (bool): Whether or not the current cell is a goal state.

action_space
Convenient access to the environment's action space.

Returns gym.spaces.Space – The action space of the reinforcement learning problem.

observation_space
Convenient access to the environment's observation space.

Returns

gym.spaces.Space – The observation space of the reinforcement learning problem.

spec
Returns a garage environment specification.

Returns `garage.envs.env_spec.EnvSpec` – A garage environment specification.

class ast_toolbox.envs.**Custom_GoExploreASTEnv**(*open_loop=True*, *blackbox_sim_state=True*, *fixed_init_state=False*, *s_0=None*, *simulator=None*, *reward_function=None*, *spaces=None*)

Bases: *ast_toolbox.envs.go_explore_ast_env.GoExploreASTEnv*

Custom class to change how downsampling works.

Example class of how to overload downsample to make the environment work for different environments.

downsample(*obs*, *step=None*)
Create a downsampled approximation of the observed simulation state.

Parameters

- **obs** (*array_like*) – The observed simulation state.
- **step** (*int, optional*) – The current iteration number

Returns *array_like* – The downsampled approximation of the observed simulation state.

class ast_toolbox.envs.**ASTEnv**(*open_loop=True*, *blackbox_sim_state=True*, *fixed_init_state=False*, *s_0=None*, *simulator=None*, *reward_function=None*, *spaces=None*)

Bases: `gym.core.Env`

Gym environment to turn general AST tasks into garage compatible problems.

Parameters

- **open_loop** (*bool*) – True if the simulation is open-loop, meaning that AST must generate all actions ahead of time, instead of being able to output an action in sync with the simulator, getting an observation back before the next action is generated. False to get interactive control, which requires that *blackbox_sim_state* is also False.

- **blackbox_sim_state** (*bool*) – True if the true simulation state can not be observed, in which case actions and the initial conditions are used as the observation. False if the simulation state can be observed, in which case it will be used.
- **fixed_init_state** (*bool*) – True if the initial state is fixed, False to sample the initial state for each rollout from the observaation space.
- **s_0** (*array_like*) – The initial state for the simulation (ignored if *fixed_init_state* is False)
- **simulator** (*ast_toolbox.simulators.ASTSimulator*) – The simulator wrapper, inheriting from *ast_toolbox.simulators.ASTSimulator*.
- **reward_function** (*ast_toolbox.rewards.ASTReward*) – The reward function, inheriting from *ast_toolbox.rewards.ASTReward*.
- **spaces** (*ast_toolbox.spaces.ASTSpaces*) – The observation and action space definitions, inheriting from *ast_toolbox.spaces.ASTSpaces*.

close()
Calls the simulator's *close* function, if it exists.

Returns *None or object* – Returns the output of the simulator's *close* function, or None if the simulator has no *close* function.

log()
Calls the simulator's *log* function.

render(***kwargs*)
Calls the simulator's *render* function, if it exists.

Parameters **kwargs** – Keyword arguments used in the simulators *render* function.

Returns *None or object* – Returns the output of the simulator's *render* function, or None if the simulator has no *render* function.

reset()
Resets the state of the environment, returning an initial observation.

Returns **observation** (*array_like*) – The initial observation of the space. (Initial reward is assumed to be 0.)

simulate(*actions*)
Run a full simulation rollout.

Parameters **actions** (*list[array_like]*) – A list of array_likes, where each member is the action taken at that step.

Returns

- *int* – The step of the trajectory where a collision was found, or -1 if a collision was not found.
- *dict* – A dictionary of simulation information for logging and diagnostics.

step(*action*)
Run one timestep of the environment's dynamics. When end of episode is reached, reset() should be called to reset the environment's internal state.

Parameters **action** (*array_like*) – An action provided by the environment.

Returns

garage.envs.base.Step() – A step in the rollout. Contains the following information:

- observation (array_like): Agent's observation of the current environment.
- reward (float): Amount of reward due to the previous action.
- done (bool): Is the current step a terminal or goal state, ending the rollout.
- actions (array_like): The action taken at the current.
- state (array_like): The cloned simulation state at the current cell, used for resetting if chosen to start a rollout.
- is_terminal (bool): Whether or not the current cell is a terminal state.
- is_goal (bool): Whether or not the current cell is a goal state.

action_space
Convenient access to the environment's action space.

Returns

gym.spaces.Space – The action space of the reinforcement learning problem.

observation_space
Convenient access to the environment's observation space.

Returns

gym.spaces.Space – The observation space of the reinforcement learning problem.

spec
Returns a garage environment specification.

Returns `garage.envs.env_spec.EnvSpec` – A garage environment specification.

Submodules

ast_toolbox.envs.ast_env module

Gym environment to turn general AST tasks into garage compatible problems.

class ast_toolbox.envs.ast_env.**ASTEnv**(*open_loop=True*, *blackbox_sim_state=True*, *fixed_init_state=False*, *s_0=None*, *simulator=None*, *reward_function=None*, *spaces=None*)

Bases: `gym.core.Env`

Gym environment to turn general AST tasks into garage compatible problems.

Parameters

- **open_loop** (*bool*) – True if the simulation is open-loop, meaning that AST must generate all actions ahead of time, instead of being able to output an action in sync with the simulator, getting an observation back before the next action is generated. False to get interactive control, which requires that *blackbox_sim_state* is also False.
- **blackbox_sim_state** (*bool*) – True if the true simulation state can not be observed, in which case actions and the initial conditions are used as the observation. False if the simulation state can be observed, in which case it will be used.
- **fixed_init_state** (*bool*) – True if the initial state is fixed, False to sample the initial state for each rollout from the observaation space.
- **s_0** (*array_like*) – The initial state for the simulation (ignored if *fixed_init_state* is False)

- **simulator** (*ast_toolbox.simulators.ASTSimulator*) – The simulator wrapper, inheriting from *ast_toolbox.simulators.ASTSimulator*.
- **reward_function** (*ast_toolbox.rewards.ASTReward*) – The reward function, inheriting from *ast_toolbox.rewards.ASTReward*.
- **spaces** (*ast_toolbox.spaces.ASTSpaces*) – The observation and action space definitions, inheriting from *ast_toolbox.spaces.ASTSpaces*.

close()
Calls the simulator's *close* function, if it exists.

Returns *None or object* – Returns the output of the simulator's *close* function, or None if the simulator has no *close* function.

log()
Calls the simulator's *log* function.

render(***kwargs*)
Calls the simulator's *render* function, if it exists.

Parameters **kwargs** – Keyword arguments used in the simulators *render* function.

Returns *None or object* – Returns the output of the simulator's *render* function, or None if the simulator has no *render* function.

reset()
Resets the state of the environment, returning an initial observation.

Returns **observation** (*array_like*) – The initial observation of the space. (Initial reward is assumed to be 0.)

simulate(*actions*)
Run a full simulation rollout.

Parameters **actions** (*list[array_like]*) – A list of array_likes, where each member is the action taken at that step.

Returns

- *int* – The step of the trajectory where a collision was found, or -1 if a collision was not found.
- *dict* – A dictionary of simulation information for logging and diagnostics.

step(*action*)
Run one timestep of the environment's dynamics. When end of episode is reached, reset() should be called to reset the environment's internal state.

Parameters **action** (*array_like*) – An action provided by the environment.

Returns

garage.envs.base.Step() – A step in the rollout. Contains the following information:

- observation (array_like): Agent's observation of the current environment.
- reward (float): Amount of reward due to the previous action.
- done (bool): Is the current step a terminal or goal state, ending the rollout.
- actions (array_like): The action taken at the current.
- state (array_like): The cloned simulation state at the current cell, used for resetting if chosen to start a rollout.

- is_terminal (bool): Whether or not the current cell is a terminal state.
- is_goal (bool): Whether or not the current cell is a goal state.

action_space
Convenient access to the environment's action space.

Returns gym.spaces.Space – The action space of the reinforcement learning problem.

observation_space
Convenient access to the environment's observation space.

Returns

gym.spaces.Space – The observation space of the reinforcement learning problem.

spec
Returns a garage environment specification.

Returns `garage.envs.env_spec.EnvSpec` – A garage environment specification.

ast_toolbox.envs.go_explore_ast_env module

Gym environment to turn general AST tasks into garage compatible problems with Go-Explore style resets.

class ast_toolbox.envs.go_explore_ast_env.**Custom_GoExploreASTEnv**(*open_loop=True*, *blackbox_sim_state=True*, *fixed_init_state=False*, *s_0=None*, *simulator=None*, *reward_function=None*, *spaces=None*)

Bases: *ast_toolbox.envs.go_explore_ast_env.GoExploreASTEnv*

Custom class to change how downsampling works.

Example class of how to overload downsample to make the environment work for different environments.

downsample(*obs*, *step=None*)
Create a downsampled approximation of the observed simulation state.

Parameters

- **obs** (*array_like*) – The observed simulation state.
- **step** (*int, optional*) – The current iteration number

Returns *array_like* – The downsampled approximation of the observed simulation state.

class ast_toolbox.envs.go_explore_ast_env.**GoExploreASTEnv**(*open_loop=True*, *blackbox_sim_state=True*, *fixed_init_state=False*, *s_0=None*, *simulator=None*, *reward_function=None*, *spaces=None*)

Bases: gym.core.Env, *ast_toolbox.envs.go_explore_ast_env.Parameterized*

Gym environment to turn general AST tasks into garage compatible problems with Go-Explore style resets.

Certain algorithms, such as Go-Explore and the Backwards Algorithm, require deterministic resets of the simulator. *GoExploreASTEnv* handles this by cloning simulator states and saving them in a cell structure. The cells are then stored in a hashed database.

Parameters

- **open_loop** (*bool*) – True if the simulation is open-loop, meaning that AST must generate all actions ahead of time, instead of being able to output an action in sync with the simulator, getting an observation back before the next action is generated. False to get interactive control, which requires that *blackbox_sim_state* is also False.
- **blackbox_sim_state** (*bool*) – True if the true simulation state can not be observed, in which case actions and the initial conditions are used as the observation. False if the simulation state can be observed, in which case it will be used
- **fixed_init_state** (*bool*) – True if the initial state is fixed, False to sample the initial state for each rollout from the observaation space.
- **s_0** (*array_like*) – The initial state for the simulation (ignored if *fixed_init_state* is False)
- **simulator** (*ast_toolbox.simulators.ASTSimulator*) – The simulator wrapper, inheriting from *ast_toolbox.simulators.ASTSimulator*.
- **reward_function** (*ast_toolbox.rewards.ASTReward*) – The reward function, inheriting from *ast_toolbox.rewards.ASTReward*.
- **spaces** (*ast_toolbox.spaces.ASTSpaces*) – The observation and action space definitions, inheriting from *ast_toolbox.spaces.ASTSpaces*.

close()
Calls the simulator's *close* function, if it exists.

Returns *None or object* – Returns the output of the simulator's *close* function, or None if the simulator has no *close* function.

downsample(*obs*)
Create a downsampled approximation of the observed simulation state.

Parameters **obs** (*array_like*) – The observed simulation state.

Returns *array_like* – The downsampled approximation of the observed simulation state.

env_reset()
Resets the state of the environment, returning an initial observation.

Returns **observation** (*array_like*) – The initial observation of the space. (Initial reward is assumed to be 0.)

get_cache_list()
Returns the environment info cache.

Returns *dict* – A dictionary containing diagnostic and logging information for the environment.

get_first_cell()
Returns a the observation and state of the initial state, to be used for a root cell.

Returns

- **obs** (*array_like*) – Agent's observation of the current environment.
- **state** (*array_like*) – The cloned simulation state at the current cell, used for resetting if chosen to start a rollout.

get_param_values(***tags*)
Return the values of internal parameters.

Parameters **tags** (*dict[bool]*) – For each tag, a parameter is returned if the parameter name matches the tag's key

Returns *list* – A list of parameter values.

get_params_internal(***tags*)
Returns the parameters associated with the given tags.

Parameters **tags** (*dict[bool]*) – For each tag, a parameter is returned if the parameter name matches the tag's key

Returns *list* – List of parameters

log()
Calls the simulator's *log* function.

render(***kwargs*)
Calls the simulator's *render* function, if it exists.

Returns *None or object* – Returns the output of the simulator's *render* function, or None if the simulator has no *render* function.

reset(***kwargs*)
Resets the state of the environment, returning an initial observation.

The reset has 2 modes.

In the "robustify" mode (self.p_robustify_state.value is not None), the simulator resets the environment to *p_robustify_state.value*. It then returns the initial condition.

In the "Go-Explore" mode, the environment attempts to sample a cell from the cell pool. If successful, the simulator is reset to the cell's state. On an error, the environment is reset to the intial state.

Returns **observation** (*array_like*) – The initial observation of the space. (Initial reward is assumed to be 0.)

sample(*population*)
Sample a cell from the cell pool with likelihood proportional to cell fitness.

The sampling is done using Stochastic Acceptance[1], with inspiration from John B Nelson's blog [2]_.

The sampler rejects cells until the acceptance criterea is met. If the maximum number of rejections is exceeded, the sampler then will sample uniformly sample a cell until it finds a cell with fitness > 0. If the second sampling phase also exceeds the rejection limit, then the function raises an exception.

Parameters **population** (*list*) – A list containing the population of cells to sample from.

Returns *object* – The sampled cell.

Raises `ValueError` – If the maximum number of rejections is exceeded in both the proportional and the uniform sampling phases.

References

Physica A: Statistical Mechanics and its Applications 391.6 (2012): 2193-2196. https://arxiv.org/pdf/1109.3627.pdf .. [2] https://jbn.github.io/fast_proportional_selection/

set_param_values(*param_values*, ***tags*)
Set the values of parameters

Parameters

[1] Lipowski, Adam, and Dorota Lipowska. "Roulette-wheel selection via stochastic acceptance."

- **param_values** (*object*) – Value to set the parameter to.
- **tags** (*dict[bool]*) – For each tag, a parameter is returned if the parameter name matches the tag's key

simulate(*actions*)
Run a full simulation rollout.

Parameters **actions** (*list[array_like]*) – A list of array_likes, where each member is the action taken at that step.

Returns

- *int* – The step of the trajectory where a collision was found, or -1 if a collision was not found.
- *dict* – A dictionary of simulation information for logging and diagnostics.

step(*action*)
Run one timestep of the environment's dynamics. When end of episode is reached, reset() should be called to reset the environment's internal state.

Parameters **action** (*array_like*) – An action provided by the environment.

Returns

`garage.envs.base.Step()` – A step in the rollout. Contains the following information:

- observation (array_like): Agent's observation of the current environment.
- reward (float): Amount of reward due to the previous action.
- done (bool): Is the current step a terminal or goal state, ending the rollout.
- cache (dict): A dictionary containing other diagnostic information from the current step.
- actions (array_like): The action taken at the current.
- state (array_like): The cloned simulation state at the current cell, used for resetting if chosen to start a rollout.
- is_terminal (bool): Whether or not the current cell is a terminal state.
- is_goal (bool): Whether or not the current cell is a goal state.

action_space
Convenient access to the environment's action space.

Returns gym.spaces.Space – The action space of the reinforcement learning problem.

observation_space
Convenient access to the environment's observation space.

Returns

gym.spaces.Space – The observation space of the reinforcement learning problem.

spec
Returns a garage environment specification.

Returns `garage.envs.env_spec.EnvSpec` – A garage environment specification.

class ast_toolbox.envs.go_explore_ast_env.**GoExploreParameter**(*name*, *value*)
Bases: `object`

A wrapper for variables that will be set as parameters in the *GoExploreASTEnv* :Parameters: * **name** (*str*) – Name of the parameter.

- **value** (*value*) – Value of the parameter.

get_value(***kwargs*)
Return the value of the parameter.

Parameters **kwargs** – Extra keyword arguments (Not currently used).

Returns *object* – The value of the parameter.

set_value(*value*)
Set the value of the parameter

Parameters **value** (*object*) – What to set the parameters *value* to.

class ast_toolbox.envs.go_explore_ast_env.**Parameterized**
Bases: object

A slimmed down version of the (deprecated) Parameterized class from garage for passing parameters to environments.

Garage uses pickle to handle parallelization, which limits the types of objects that can be used as class attributes withing the environment. This class is a workaround, so that the parallel environments can have access to things like a database.

get_params(***tags*)
Get the list of parameters, filtered by the provided tags. Some common tags include 'regularizable' and 'trainable'

Parameters **tags** (*str*) – Names of the paramters to return.

get_params_internal(***tags*)
Internal method to be implemented which does not perform caching

Parameters **tags** (*str*) – Names of the paramters to return.

ast_toolbox.mcts package

MCTS Helper files

Submodules

ast_toolbox.mcts.ASTSim module

class ast_toolbox.mcts.ASTSim.**AcionSequence**(*sequence*, *index=0*)
Bases: object

Sturcture storing the actions sequences.

Parameters

- **sequence** (*list*) – The list of actions.
- **index** (*int, optional*) – The initial action index in the sequence.

ast_toolbox.mcts.ASTSim.**action_seq_policy**(*action_seq*, *s*)
The policy wrapper for the action sequence.

Parameters

- **action_seq** (*ast_toolbox.mcts.ASTSim.AcionSequence*) – The action sequence.
- **s** (*ast_toolbox.mcts.AdaptiveStressTesting.ASTState*) – The AST state.

Returns **action** (*ast_toolbox.mcts.AdaptiveStressTesting.ASTAction*) – The AST action.

ast_toolbox.mcts.ASTSim.**play_sequence**(*ast*, *actions*, *verbose=False*, *sleeptime=0.0*)
Rollout the action sequence.

Parameters

- **ast** (*ast_toolbox.mcts.AdaptiveStressTesting.AdaptiveStressTest*) – The AST object.
- **actions** (*list*) – The action sequence.
- **verbose** (*bool, optional*) – Whether to log the rollout information.
- **sleeptime** (*float, optional*) – The pause time between each step.

Returns

- **rewards** (*list[float]*) – The rewards.
- **actions2** (*list*) – The action sequence of the path. Should be the same as the input actions.

ast_toolbox.mcts.AST_MCTS module

ast_toolbox.mcts.AST_MCTS.**explore_getAction**(*ast*)
Get the exploration function from ast.

Parameters **ast** (ast_toolbox.mcts.AdaptiveStressTest.AdaptiveStressTesting) – The AST object.

ast_toolbox.mcts.AST_MCTS.**rollout_getAction**(*ast*)
Get the rollout function from ast.

Parameters **ast** (ast_toolbox.mcts.AdaptiveStressTest.AdaptiveStressTesting) – The AST object.

ast_toolbox.mcts.AST_MCTS.**stress_test**(*ast*, *mcts_params*, *top_paths*, *verbose=True*, *return_tree=False*)
Run stress test with mode 1 (search with single tree).

Parameters

- **ast** (ast_toolbox.mcts.AdaptiveStressTest.AdaptiveStressTesting) – The AST object.
- **mcts_params** (*ast_toolbox.mcts.MCTSdpw.DPWParams*) – The mcts parameters.
- **top_paths** (*ast_toolbox.mcts.BoundedPriorityQueues*) – The bounded priority queue to store top-rewarded trajectories.
- **verbose** (*bool, optional*) – Whether to logging test information
- **return_tree** (*bool, optional*) – Whether to return the search tree

Returns

- **results** (ast_toolbox.mcts.AdaptiveStressTest.AdaptiveStressTesting) – The bounded priority queue storing top-rewarded trajectories.

- **tree** (*dict*) – The resulting searching tree.

ast_toolbox.mcts.AST_MCTS.**stress_test2**(*ast*, *mcts_params*, *top_paths*, *verbose=True*, *return_tree=False*)

Run stress test with mode 2 (search with multiple trees).

Parameters

- **ast** (ast_toolbox.mcts.AdaptiveStressTest.AdaptiveStressTesting) – The AST object.
- **mcts_params** (*ast_toolbox.mcts.MCTSdpw.DPWParams*) – The mcts parameters.
- **top_paths** (*ast_toolbox.mcts.BoundedPriorityQueues*) – The bounded priority queue to store top-rewarded trajectories.
- **verbose** (*bool, optional*) – Whether to logging test information
- **return_tree** (*bool, optional*) – Whether to return the search tree

Returns

- **results** (ast_toolbox.mcts.AdaptiveStressTest.AdaptiveStressTesting) – The bounded priority queue storing top-rewarded trajectories.
- **tree** (*dict*) – The resulting searching tree.

ast_toolbox.mcts.AdaptiveStressTesting module

class ast_toolbox.mcts.AdaptiveStressTesting.**ASTAction**(*action*)

Bases: object

get()

Get the true action.

Returns *action* – The true actions used in the env.

class ast_toolbox.mcts.AdaptiveStressTesting.**ASTParams**(*max_steps*, *log_interval*, *log_tabular*, *log_dir=None*, *n_itr=100*)

Bases: object

Structure that stores internal parameters for AST.

Parameters **max_steps** (*int, optional*) – The maximum search depth.

class ast_toolbox.mcts.AdaptiveStressTesting.**ASTState**(*t_index*, *parent*, *action*)

Bases: object

The AST state.

Parameters

- **t_index** (*int*) – The index of the timestep.
- **parent** (*ast_toolbox.mcts.AdaptiveStressTesting.ASTState*) – The parent state.
- **action** (*ast_toolbox.mcts.AdaptiveStressTesting.ASTAction*) – The action leading to this state.

class ast_toolbox.mcts.AdaptiveStressTesting.**AdaptiveStressTest**(*p*, *env*, *top_paths*)

Bases: object

The AST wrapper for MCTS using the actions in env.action_space.

Parameters

- **p** (*ast_toolbox.mcts.AdaptiveStressTesting.ASTParams*) – The AST parameters
- **env** (*ast_toolbox.envs.go_explore_ast_env.GoExploreASTEnv.*) – The environment.
- **top_paths** (*ast_toolbox.mcts.BoundedPriorityQueues*, optional) – The bounded priority queue to store top-rewarded trajectories.

explore_action(*s*, *tree*)

Randomly sample an action for the exploration.

Parameters

- **s** (*ast_toolbox.mcts.AdaptiveStressTesting.ASTState*) – The current state.
- **tree** (*dict*) – The searching tree.

Returns **action** (*ast_toolbox.mcts.AdaptiveStressTesting.ASTAction*) – The sampled action.

get_reward()

Get the current AST reward.

Returns **reward** (*bool*) – The AST reward.

initialize()

Initialize training variables.

Returns *env_reset* – The reset result from the env.

isterminal()

Check whether the current path is finished.

Returns **isterinal** (*bool*) – Whether the current path is finished.

logging()

Logging the training information.

random_action()

Randomly sample an action for the rollout.

Returns **action** (*ast_toolbox.mcts.AdaptiveStressTesting.ASTAction*) – The sampled action.

reset_step_count()

Reset the env step count.

transition_model()

Generate the transition model used in MCTS.

Returns **transition_model** (*ast_toolbox.mcts.MDP.TransitionModel*) – The transition model.

update(*action*)

Update the environment as well as the assosiated parameters.

Parameters **action** (*ast_toolbox.mcts.AdaptiveStressTesting.ASTAction*) – The AST action.

Returns

- **obs** (numpy.ndarry) – The observation from the env step.
- **reward** (*float*) – The reward from the env step.
- **done** (*bool*) – The terminal indicator from the env step.
- **info** (*dict*) – The env info from the env step.

ast_toolbox.mcts.AdaptiveStressTesting.**get_action_sequence**(*s*)

Get the action sequence that leads to the state.

Parameters **s** (*ast_toolbox.mcts.AdaptiveStressTesting.ASTState*) – The target state.

Returns **actions** (list[*ast_toolbox.mcts.AdaptiveStressTesting.ASTAction*]) – The action sequences leading to the target state.

ast_toolbox.mcts.AdaptiveStressTestingBlindValue module

class ast_toolbox.mcts.AdaptiveStressTestingBlindValue.**AdaptiveStressTestBV**(***kwargs*)

Bases: *ast_toolbox.mcts.AdaptiveStressTesting.AdaptiveStressTest*

The AST wrapper for MCTS using the Blind Value exploration[1].

Parameters **kwargs** – Keyword arguments passed to *ast_toolbox.mcts.AdaptiveStressTesting.AdaptiveStressTest*

References

explore_action(*s*, *tree*)

Sample an action for the exploration using Blind Value.

Parameters

- **s** (*ast_toolbox.mcts.AdaptiveStressTesting.ASTState*) – The current state.
- **tree** (*dict*) – The searching tree.

Returns **action** (*ast_toolbox.mcts.AdaptiveStressTesting.ASTAction*) – The sampled action.

getBV(*y*, *rho*, *A*, *UCB*)

Calculate the Blind Value for the candidate action y

Parameters

- **y** (numpy.ndarry) – The candidate action.
- **rho** (*float*) – The standard deviation ratio.
- **A** (list[*ast_toolbox.mcts.AdaptiveStressTesting.ASTAction*]) – The list of the explored AST actions
- **UCB** (*dict*) – The dictionary containing the upper confidence bound for each explored action in the state node.

[1] Couetoux, Adrien, Hassen Doghmen, and Olivier Teytaud. "Improving the exploration in upper confidence trees." International Conference on Learning and Intelligent Optimization. Springer, Berlin, Heidelberg, 2012.

Returns **BV** (*float*) – The blind value.

getDistance(*a*, *b*)
Get the (L2) distance between two actions.

Parameters

- **a** (numpy.ndarry) – The first action.
- **b** (numpy.ndarry) – The second action.

Returns **distance** (*float*) – The L2 distance between a and b.

getUCB(*s*)
Get the upper confidnece bound for the expected return for evary actions that has been explored at the state.

Parameters **s** (ast_toolbox.MCTSdpw.StateNode) – The state node in the searching tree

Returns **UCB** (*dict*) – The dictionary containing the upper confidence bound for each explored action in the state node.

ast_toolbox.mcts.AdaptiveStressTestingRandomSeed module

class ast_toolbox.mcts.AdaptiveStressTestingRandomSeed.**ASTRSAction**(*action*, *env*)
Bases: object

The AST action containing the random seed.

Parameters **action** – The random seed. env : *ast_toolbox.envs.go_explore_ast_env.GoExploreASTEnv*

The environment.

get()
Get the true action.

Returns *action* – The true actions used in the env.

class ast_toolbox.mcts.AdaptiveStressTestingRandomSeed.**AdaptiveStressTestRS**(***kwargs*)
Bases: *ast_toolbox.mcts.AdaptiveStressTesting.AdaptiveStressTest*

The AST wrapper for MCTS using random seeds as actions.

Parameters **kwargs** – Keyword arguments passed to *ast_toolbox.mcts.AdaptiveStressTesting.AdaptiveStressTest*

explore_action(*s*, *tree*)
Randomly sample an action for the exploration.

Returns **action** (*ast_toolbox.mcts.AdaptiveStressTestingRandomSeed.ASTRSAction*) – The sampled action.

random_action()
Randomly sample an action for the rollout.

Returns **action** (*ast_toolbox.mcts.AdaptiveStressTestingRandomSeed.ASTRSAction*) – The sampled action.

reset_rsg()
Reset the random seed generator.

ast_toolbox.mcts.BoundedPriorityQueues module

class ast_toolbox.mcts.BoundedPriorityQueues.**BoundedPriorityQueue**(*N*)
Bases: object

The bounded priority Queue.

Parameters **N** (*int*) – Size of the queue.

empty()
Clear the queue.

enqueue(*k*, *v*, *make_copy=False*)
Storing k into the queue based on the priority value v.

Parameters

- **k** – The object to be stored.
- **v** (*float*) – The priority value.
- **make_copy** (*bool, optional*) – Whether to make a copy of the k.

haskey(*k*)
Check whether k in in the queue.

Returns **has_key** (*bool*) – Whether k in in the queue.

isempty()
Check whether the queue is empty.

Returns **is_empty** (*bool*) – Whether the queue is empty.

length()
Return the current size of the queue.

Returns **length** (*int*) – The current size of the queue.

ast_toolbox.mcts.MCTSdpw module

class ast_toolbox.mcts.MCTSdpw.**DPWModel**(*model*, *getAction*, *getNextAction*)
Bases: object

The model used in the tree search.

Parameters

- **model** (*ast_toolbox.mcts.MDP.TransitionModel*) – The transition model.
- **getAction** (*function*) – getAction(s, tree) returns the action used in rollout.
- **getNextAction** (*function*) – getNextAction(s, tree) returns the action used in exploration.

class ast_toolbox.mcts.MCTSdpw.**DPWParams**(*d*, *gamma*, *ec*, *n*, *k*, *alpha*, *clear_nodes*)
Bases: object

Structure that stores the parameters for the MCTS with DPW.

Parameters

- **d** (*int*) – The maximum searching depth.
- **gamma** (*float*) – The discount factor.
- **ec** (*float*) – The weight for the exploration bonus.

- **n** (*int*) – The mximum number of iterations.
- **k** (*float*) – The constraint parameter used in DPW: |N(s,a)|<=kN(s)^alpha.
- **alpha** (*float*) – The constraint parameter used in DPW: |N(s,a)|<=kN(s)^alpha.
- **clear_nodes** (*bool*) – Whether to clear redundant nodes in tree. Set it to True for saving memoray. Set it to False to better tree plotting.

class ast_toolbox.mcts.MCTSdpw.**DPWTree**(*p*, *f*)
Bases: object

The structure storing the seaching tree.

class ast_toolbox.mcts.MCTSdpw.**StateActionNode**
Bases: object

The structure representing the state-action node.

class ast_toolbox.mcts.MCTSdpw.**StateActionStateNode**
Bases: object

The structure storing the transition state-action-state.

class ast_toolbox.mcts.MCTSdpw.**StateNode**
Bases: object

The structure representing the state node.

ast_toolbox.mcts.MCTSdpw.**rollout**(*tree*, *s*, *depth*)
Rollout from the current state s.

Parameters

- **tree** (*ast_toolbox.mcts.MCTSdpw.DPWTree*) – The seach tree.
- **s** (*ast_toolbox.mcts.AdaptiveStressTesting.ASTState*) – The current state.
- **depth** (*int*) – The maximum search depth

Returns **q** (*float*) – The estimated return.

ast_toolbox.mcts.MCTSdpw.**saveBackwardState**(*old_s_tree*, *new_s_tree*, *s_current*)
Saving the s_current as well as all its predecessors in the old_s_tree into the new_s_tree.

Parameters

- **old_s_tree** (*dict*) – The old tree.
- **new_s_tree** (*dict*) – The new tree.
- **s_current** (*ast_toolbox.mcts.AdaptiveStressTesting.ASTState*) – The current state.

Returns **new_s_tree** (*dict*) – The new tree.

ast_toolbox.mcts.MCTSdpw.**saveForwardState**(*old_s_tree*, *new_s_tree*, *s*)
Saving the s_current as well as all its successors in the old_s_tree into the new_s_tree.

Parameters

- **old_s_tree** (*dict*) – The old tree.
- **new_s_tree** (*dict*) – The new tree.
- **s_current** (*ast_toolbox.mcts.AdaptiveStressTesting.ASTState*) – The current state.

Returns **new_s_tree** (*dict*) – The new tree.

ast_toolbox.mcts.MCTSdpw.**saveState**(*old_s_tree*, *s*)
Saving the s_current as well as all its predecessors and successors in the old_s_tree into the new_s_tree.

Parameters

- **old_s_tree** (*dict*) – The old tree.
- **s** (*ast_toolbox.mcts.AdaptiveStressTesting.ASTState*) – The current state.

Returns **new_s_tree** (*dict*) – The new tree.

ast_toolbox.mcts.MCTSdpw.**selectAction**(*tree*, *s*, *verbose=False*)
Run MCTS to select one action for the state s

Parameters

- **tree** (*ast_toolbox.mcts.MCTSdpw.DPWTree*) – The seach tree.
- **s** (*ast_toolbox.mcts.AdaptiveStressTesting.ASTState*) – The current state.
- **verbose** (*bool, optional*) – Where to log the seaching information.

Returns **action** (*ast_toolbox.mcts.AdaptiveStressTesting.ASTAction*) – The selected AST action.

ast_toolbox.mcts.MCTSdpw.**simulate**(*tree*, *s*, *depth*, *verbose=False*)
Single run of the forward MCTS search.

Parameters

- **tree** (*ast_toolbox.mcts.MCTSdpw.DPWTree*) – The seach tree.
- **s** (*ast_toolbox.mcts.AdaptiveStressTesting.ASTState*) – The current state.
- **depth** (*int*) – The maximum search depth
- **verbose** (*bool, optional*) – Where to log the seaching information.

Returns **q** (*float*) – The estimated return.

ast_toolbox.mcts.MDP module

class ast_toolbox.mcts.MDP.**TransitionModel**(*getInitialState*, *getNextState*, *isEndState*, *maxSteps*, *goToState*)
Bases: object

The wrapper for the transitin model used in the tree search.

Parameters

- **getInitialState** (*function*) – getInitialState() returns the initial AST state.
- **getNextState** (*function*) – getNextState(s, a) returns the next state and the reward.
- **isEndState** (*function*) – isEndState(s) returns whether s is a terminal state.
- **maxSteps** (*int*) – The maximum path length.
- **goToState** (*function*) – goToState(s) sets the simulator to the target state s.

ast_toolbox.mcts.MDP.**simulate**(*model*, *p*, *policy*, *verbose=False*, *sleeptime=0.0*)
Simulate the environment model using the policy and the parameter p.

Parameters

- **model** (*ast_toolbox.mcts.MDP.TransitionModel*) – The environment model.
- **p** – The extra paramters needed by the policy.
- **policy** (*function*) – policy(p, s) returns the next action.
- **verbose** (*bool, optional*) – Whether to logging simulating information.
- **sleeptime** (*float, optional*) – The pause time between each step.

Returns

- **cum_reward** (*float*) – The cumulative reward.
- **actions** (*list*) – The action sequence of the path.

ast_toolbox.mcts.RNGWrapper module

class ast_toolbox.mcts.RNGWrapper.**RSG**(*state_length=1*, *seed=0*)
Bases: object

The random seed generator for AST using random seeds.

Parameters

- **state_length** (*int, optional*) – The length of the RSG state.
- **seed** (*int, optional*) – The initial seed to generate the initial state.

length()
Return the length of the RSG state.

Returns **length** (*int*) – The length of the RSG state.

next()
Step the RSG state.

set_from_seed(*length*, *seed*)
Set the RSG state using the seed.

Parameters

- **length** (*int*) – The length of the RSG state.
- **seed** (*int*) – The seed to generate the state.

ast_toolbox.mcts.RNGWrapper.**seed_to_state_itr**(*state_length*, *seed*)
Generate the RSG state using the seed.

Parameters

- **state_length** (*int*) – The length of the RSG state.
- **seed** (*int*) – The seed to generate the state.

Returns **state** (numpy.ndarry) – The generated state.

ast_toolbox.mcts.tree_plot module

ast_toolbox.mcts.tree_plot.**add_children**(*s*, *s_node*, *tree*, *graph*, *d*)
Add successors of s into the graph.

Parameters

- **s** (*ast_toolbox.mcts.AdaptiveStressTesting.ASTState*) – The AST state.
- **s_node** (pydot.Node) – The pydot node corresponding to s.
- **tree** (*dict*) – The tree.
- **graph** (pydot.Dot) – The pydot graph.
- **d** (*int*) – The depth.

ast_toolbox.mcts.tree_plot.**get_root**(*tree*)
Get the root node of the tree.

Parameters **tree** (*dict*) – The tree.

Returns **s** (*ast_toolbox.mcts.AdaptiveStressTesting.ASTState*) – The root state.

ast_toolbox.mcts.tree_plot.**plot_tree**(*tree*, *d*, *path*, *format='svg'*)
Plot the tree.

Parameters

- **tree** (*dict*) – The tree.
- **d** (*int*) – The depth.
- **path** (*str*) – The plotting path.
- **format** (*str*) – The plotting format.

ast_toolbox.mcts.tree_plot.**s2node**(*s*, *tree*)
Transfer the AST state to pydot node.

Parameters

- **s** (*ast_toolbox.mcts.AdaptiveStressTesting.ASTState*) – The AST state.
- **tree** (*dict*) – The tree.

Returns **node** (pydot.Node) – The pydot node.

ast_toolbox.optimizers package

Optimizers for RL problems

Submodules

ast_toolbox.optimizers.direction_constraint_optimizer module

class ast_toolbox.optimizers.direction_constraint_optimizer.**DirectionConstraintOptimizer**(*cg*
re
05
su
sa
pl
ba
tr
m
de
bu
ac
ce
hv
nu

Bases: `object`

Performs constrained optimization via line search on the given gradient direction.

Parameters

- **cg_iters** (*int, optional*) – The number of CG iterations used to calculate A^-1 g.
- **reg_coeff** (*float, optional*) – A small value so that A -> A + reg*I.
- **subsample_factor** (*int, optional*) – Subsampling factor to reduce samples when using "conjugate gradient. Since the computation time for the descent direction dominates, this can greatly reduce the overall computation time.
- **debug_nan** (*bool, optional*) – If set to True, NanGuard will be added to the compilation, and ipdb will be invoked when nan is detected.
- **accept_violation** (*bool, optional*) – Whether to accept the descent step if it violates the line search condition after exhausting all backtracking budgets.

constraint_val(*inputs*, *extra_inputs=None*)

Calculate the constraint value.

Parameters

- **inputs** – A list of symbolic variables as inputs, which could be subsampled if needed. It is assumed that the first dimension of these inputs should correspond to the number of data points.
- **extra_inputs** (*optional*) – A list of symbolic variables as extra inputs which should not be subsampled.

Returns **constraint_value** (*float*) – The value of the constrained variable.

get_magnitude(*direction*, *inputs*, *max_constraint_val=None*, *extra_inputs=None*, *subsample_grouped_inputs=None*)

Calculate the update magnitude.

Parameters

- **direction** (*:py:class:'tensorflow.Tensor'*) – The gradient direction.
- **inputs** – A list of symbolic variables as inputs, which could be subsampled if needed. It is assumed that the first dimension of these inputs should correspond to the number of data points.

- **max_constraint_val** (*float, optional*) – The maximum value for the constrained variale.
- **extra_inputs** (*optional*) – A list of symbolic variables as extra inputs which should not be subsampled.
- **subsample_grouped_inputs** (*optional*) – The list of inputs that are needed to be subsampled.

Returns **magnitude** (*float*) – The update magnitude.

update_opt(*target*, *leq_constraint*, *inputs*, *extra_inputs=None*, *constraint_name='constraint'*, **args*, ***kwargs*)

Update the internal tensowflow operations.

Parameters

- **target** – A parameterized object to optimize over. It should implement methods of the `garage.core.paramerized.Parameterized` class.
- **leq_constraint** (*:py:class:'tensorflow.Tensor'*) – The variable to be constrained.
- **inputs** – A list of symbolic variables as inputs, which could be subsampled if needed. It is assumed that the first dimension of these inputs should correspond to the number of data points.
- **extra_inputs** – A list of symbolic variables as extra inputs which should not be subsampled.

ast_toolbox.policies package

Policies for solving AST problems.

class ast_toolbox.policies.**GoExplorePolicy**(*env_spec*, *name='GoExplorePolicy'*)

Bases: `garage.tf.policies.base.StochasticPolicy`

A stochastic policy for Go-Explore that takes actions uniformally at random.

Parameters

- **env_spec** (`garage.envs.EnvSpec`) – Environment specification.
- **name** (*str*) – Name for the tensors.

dist_info(*obs*, *state_infos*)

Distribution info.

Return the distribution information about the actions.

Parameters

- **obs** (*array_like*) – Observation values.
- **state_infos** (*dict*) – A dictionary whose values should contain information about the state of the policy at the time it received the observation.

dist_info_sym(*obs_var*, *state_info_vars*, *name='dist_info_sym'*)

Symbolic graph of the distribution.

Return the symbolic distribution information about the actions.

Parameters

- **obs_var** (tf.Tensor) – Symbolic variable for observations.

- **state_infos** (*dict*) – A dictionary whose values should contain information about the state of the policy at the time it received the observation.
- **name** (*str*) – Name of the symbolic graph.

get_action(*observation*)
Get action sampled from the policy.

Parameters **observation** (*array_like*) – Observation from the environment.

Returns *array_like* – Action sampled from the policy.

get_actions(*observations*)
Get actions sampled from the policy.

Parameters **observations** (*list[array_like]*) – Observations from the environment.

Returns *array_like* – Actions sampled from the policy.

log_diagnostics(*paths*)
Log extra information per iteration based on the collected paths.

reset(*dones=None*)
Reset the policy.

If dones is None, it will be by default np.array([True]) which implies the policy will not be "vectorized", i.e. number of parallel environments for training data sampling = 1.

Parameters **dones** (*array_like*) – Bool that indicates terminal state(s).

terminate()
Clean up operation.

distribution
Distribution.

Returns *Distribution.*

vectorized
Indicates whether the policy is vectorized. If True, it should implement get_actions(), and support resetting with multiple simultaneous states.

Submodules

ast_toolbox.policies.go_explore_policy module

class ast_toolbox.policies.go_explore_policy.**GoExplorePolicy**(*env_spec*, *name='GoExplorePolicy'*)
Bases: garage.tf.policies.base.StochasticPolicy

A stochastic policy for Go-Explore that takes actions uniformally at random.

Parameters

- **env_spec** (garage.envs.EnvSpec) – Environment specification.
- **name** (*str*) – Name for the tensors.

dist_info(*obs*, *state_infos*)
Distribution info.

Return the distribution information about the actions.

Parameters

- **obs** (*array_like*) – Observation values.
- **state_infos** (*dict*) – A dictionary whose values should contain information about the state of the policy at the time it received the observation.

dist_info_sym(*obs_var*, *state_info_vars*, *name='dist_info_sym'*)
Symbolic graph of the distribution.

Return the symbolic distribution information about the actions.

Parameters

- **obs_var** (tf.Tensor) – Symbolic variable for observations.
- **state_infos** (*dict*) – A dictionary whose values should contain information about the state of the policy at the time it received the observation.
- **name** (*str*) – Name of the symbolic graph.

get_action(*observation*)
Get action sampled from the policy.

Parameters **observation** (*array_like*) – Observation from the environment.

Returns *array_like* – Action sampled from the policy.

get_actions(*observations*)
Get actions sampled from the policy.

Parameters **observations** (*list[array_like]*) – Observations from the environment.

Returns *array_like* – Actions sampled from the policy.

log_diagnostics(*paths*)
Log extra information per iteration based on the collected paths.

reset(*dones=None*)
Reset the policy.

If dones is None, it will be by default np.array([True]) which implies the policy will not be "vectorized", i.e. number of parallel environments for training data sampling = 1.

Parameters **dones** (*array_like*) – Bool that indicates terminal state(s).

terminate()
Clean up operation.

distribution
Distribution.

Returns *Distribution.*

vectorized
Indicates whether the policy is vectorized. If True, it should implement get_actions(), and support resetting with multiple simultaneous states.

ast_toolbox.rewards package

Reward functions for AST formulated RL problems.

class ast_toolbox.rewards.**ASTReward**
Bases: object

Function to calculate the rewards for timesteps when optimizing AST solver policies.

give_reward(*action*, ***kwargs*)
Returns the reward for a given time step.

Parameters

- **action** (*array_like*) – Action taken by the AST solver.
- **kwargs** – Accepts relevant info for computing the reward.

Returns **reward** (*float*) – Reward based on the previous action.

class ast_toolbox.rewards.**ExampleAVReward**(*num_peds=1*, *cov_x=0.1*, *cov_y=0.01*, *cov_sensor_noise=0.1*, *use_heuristic=True*)
Bases: *ast_toolbox.rewards.ast_reward.ASTReward*

An example implementation of an ASTReward for an AV validation scenario.

Parameters

- **num_peds** (*int*) – The number of pedestrians in the scenario.
- **cov_x** (*float*) – Covariance of the x-acceleration.
- **cov_y** (*float*) – Covariance of the y-acceleration.
- **cov_sensor_noise** (*float*) – Covariance of the sensor noise.
- **use_heuristic** (*bool*) – Whether to include a heuristic in the reward based on how close the pedestrian is to the vehicle at the end of the trajectory.

give_reward(*action*, ***kwargs*)
Returns the reward for a given time step.

Parameters

- **action** (*array_like*) – Action taken by the AST solver.
- **kwargs** – Accepts relevant info for computing the reward.

Returns **reward** (*float*) – Reward based on the previous action.

mahalanobis_d(*action*)
Calculate the Mahalanobis distance[1] between the action and the mean action.

Parameters **action** (*array_like*) – Action taken by the AST solver.

Returns *float* – The Mahalanobis distance between the action and the mean action.

References

Submodules

ast_toolbox.rewards.ast_reward module

class ast_toolbox.rewards.ast_reward.**ASTReward**
Bases: object

Function to calculate the rewards for timesteps when optimizing AST solver policies.

give_reward(*action*, ***kwargs*)
Returns the reward for a given time step.

[1] Mahalanobis, Prasanta Chandra. "On the generalized distance in statistics." National Institute of Science of India, 1936. http://library.isical.ac.in:8080/jspui/bitstream/123456789/6765/1/Vol02_1936_1_Art05-pcm.pdf

Parameters

- **action** (*array_like*) – Action taken by the AST solver.
- **kwargs** – Accepts relevant info for computing the reward.

Returns **reward** (*float*) – Reward based on the previous action.

ast_toolbox.rewards.example_av_reward module

An example implementation of an ASTReward for an AV validation scenario.

class ast_toolbox.rewards.example_av_reward.**ExampleAVReward**(*num_peds=1*, *cov_x=0.1*, *cov_y=0.01*, *cov_sensor_noise=0.1*, *use_heuristic=True*)

Bases: *ast_toolbox.rewards.ast_reward.ASTReward*

An example implementation of an ASTReward for an AV validation scenario.

Parameters

- **num_peds** (*int*) – The number of pedestrians in the scenario.
- **cov_x** (*float*) – Covariance of the x-acceleration.
- **cov_y** (*float*) – Covariance of the y-acceleration.
- **cov_sensor_noise** (*float*) – Covariance of the sensor noise.
- **use_heuristic** (*bool*) – Whether to include a heuristic in the reward based on how close the pedestrian is to the vehicle at the end of the trajectory.

give_reward(*action*, ***kwargs*)

Returns the reward for a given time step.

Parameters

- **action** (*array_like*) – Action taken by the AST solver.
- **kwargs** – Accepts relevant info for computing the reward.

Returns **reward** (*float*) – Reward based on the previous action.

mahalanobis_d(*action*)

Calculate the Mahalanobis distance[1] between the action and the mean action.

Parameters **action** (*array_like*) – Action taken by the AST solver.

Returns *float* – The Mahalanobis distance between the action and the mean action.

References

ast_toolbox.samplers package

Samplers for solving AST formualted RL problems.

[1] Mahalanobis, Prasanta Chandra. "On the generalized distance in statistics." National Institute of Science of India, 1936. http://library.isical.ac.in:8080/jspui/bitstream/123456789/6765/1/Vol02_1936_1_Art05-pcm.pdf

class ast_toolbox.samplers.**ASTVectorizedSampler**(*algo*, *env*, *n_envs=1*, *open_loop=True*, *sim=<ast_toolbox.simulators.example_av_simulator.example_d object>*, *reward_function=<ast_toolbox.rewards.example_av_reward.Exa object>*)

Bases: garage.sampler.on_policy_vectorized_sampler.OnPolicyVectorizedSampler

A vectorized sampler for AST to handle open-loop simulators.

Garage usually genearates samples in a closed-loop process. This version of the vectorized sampler instead grabs dummy data until the full rollout specification is generated, then goes back and runs the *simulate* function to actually obtain results. Rewards are then calculated and the path data is corrected.

Parameters

- **algo** (garage.np.algos.base.RLAlgorithm) – The algorithm.
- **env** (*ast_toolbox.envs.ASTEnv*) – The environment.
- **n_envs** (*int*) – Number of parallel environments to run.
- **open_loop** (*bool*) – True if the simulation is open-loop, meaning that AST must generate all actions ahead of time, instead of being able to output an action in sync with the simulator, getting an observation back before the next action is generated. False to get interactive control, which requires that *blackbox_sim_state* is also False.
- **sim** (*ast_toolbox.simulators.ASTSimulator*) – The simulator wrapper, inheriting from *ast_toolbox.simulators.ASTSimulator*.
- **reward_function** (*ast_toolbox.rewards.ASTReward*) – The reward function, inheriting from *ast_toolbox.rewards.ASTReward*.

obtain_samples(*itr*, *batch_size=None*, *whole_paths=False*)

Sample the policy for new trajectories.

Parameters

- **itr** (*int*) – Iteration number.
- **batch_size** (*int*) – Number of samples to be collected. If None, it will be default [algo.max_path_length * n_envs].
- **whole_paths** (*bool*) – Whether return all the paths or not. True by default. It's possible for the paths to have total actual sample size larger than batch_size, and will be truncated if this flag is true.

Returns

list[dict] – A list of sampled rollout paths. Each rollout path is a dictionary with the following keys:

- observations (numpy.ndarray)
- actions (numpy.ndarray)
- rewards (numpy.ndarray)
- agent_infos (dict)
- env_infos (dict)

slice_dict(*in_dict*, *slice_idx*)

Helper function to recursively parse through a dictionary of dictionaries and arrays to slice the arrays at a certain index.

Parameters

- **in_dict** (*dict*) – Dictionary where the values are arrays or other dictionaries that follow this stipulation.
- **slice_idx** (*int*) – Index to slice each array at.

Returns *dict* – Dictionary where arrays at every level are sliced.

class ast_toolbox.samplers.**BatchSampler**(*algo*, *env*, *n_envs=1*, *open_loop=True*, *batch_simulate=False*, *sim=<ast_toolbox.simulators.example_av_simulator.example_av_simulator object>*, *reward_function=<ast_toolbox.rewards.example_av_reward.Exam object>*)

Bases: garage.sampler.base.BaseSampler

Collects samples in parallel using a stateful pool of workers.

Parameters

- **algo** (garage.np.algos.base.RLAlgorithm) – The algorithm.
- **env** (*ast_toolbox.envs.ASTEnv*) – The environment.
- **n_envs** (*int*) – Number of parallel environments to run.
- **open_loop** (*bool*) – True if the simulation is open-loop, meaning that AST must generate all actions ahead of time, instead of being able to output an action in sync with the simulator, getting an observation back before the next action is generated. False to get interactive control, which requires that *blackbox_sim_state* is also False.
- **batch_simulate** (*bool*) – When in *obtain_samples* with *open_loop == True*, the sampler will call *self.sim.batch_simulate_paths* if *batch_simulate* is True, and *self.sim.simulate* if False.
- **sim** (*ast_toolbox.simulators.ASTSimulator*) – The simulator wrapper, inheriting from *ast_toolbox.simulators.ASTSimulator*.
- **reward_function** (*ast_toolbox.rewards.ASTReward*) – The reward function, inheriting from *ast_toolbox.rewards.ASTReward*.
- **Args** – algo (garage.np.algos.RLAlgorithm): The algorithm. env (gym.Env): The environment.

obtain_samples(*itr*, *batch_size=None*, *whole_paths=True*)

Collect samples for the given iteration number.

Parameters

- **itr** (*int*) – Iteration number.
- **batch_size** (*int, optional*) – How many simulation steps to run in each epoch.
- **whole_paths** (*bool, optional*) – Whether to return the full rollout paths data.

shutdown_worker()

Terminate workers if necessary.

slice_dict(*in_dict*, *slice_idx*)

Helper function to recursively parse through a dictionary of dictionaries and arrays to slice the arrays at a certain index.

Parameters

- **in_dict** (*dict*) – Dictionary where the values are arrays or other dictionaries that follow this stipulation.
- **slice_idx** (*int*) – Index to slice each array at.

Returns *dict* – Dictionary where arrays at every level are sliced.

start_worker()
Initialize the sampler.

Submodules

ast_toolbox.samplers.ast_vectorized_sampler module

class ast_toolbox.samplers.ast_vectorized_sampler.**ASTVectorizedSampler**(*algo*, *env*, *n_envs=1*, *open_loop=True*, *sim=<ast_toolbox.simulators object>*, *reward_function=<ast_toolbox object>*)

Bases: garage.sampler.on_policy_vectorized_sampler.OnPolicyVectorizedSampler

A vectorized sampler for AST to handle open-loop simulators.

Garage usually genearates samples in a closed-loop process. This version of the vectorized sampler instead grabs dummy data until the full rollout specification is generated, then goes back and runs the *simulate* function to actually obtain results. Rewards are then calculated and the path data is corrected.

Parameters

- **algo** (garage.np.algos.base.RLAlgorithm) – The algorithm.
- **env** (*ast_toolbox.envs.ASTEnv*) – The environment.
- **n_envs** (*int*) – Number of parallel environments to run.
- **open_loop** (*bool*) – True if the simulation is open-loop, meaning that AST must generate all actions ahead of time, instead of being able to output an action in sync with the simulator, getting an observation back before the next action is generated. False to get interactive control, which requires that *blackbox_sim_state* is also False.
- **sim** (*ast_toolbox.simulators.ASTSimulator*) – The simulator wrapper, inheriting from *ast_toolbox.simulators.ASTSimulator*.
- **reward_function** (*ast_toolbox.rewards.ASTReward*) – The reward function, inheriting from *ast_toolbox.rewards.ASTReward*.

obtain_samples(*itr*, *batch_size=None*, *whole_paths=False*)
Sample the policy for new trajectories.

Parameters

- **itr** (*int*) – Iteration number.
- **batch_size** (*int*) – Number of samples to be collected. If None, it will be default [algo.max_path_length * n_envs].
- **whole_paths** (*bool*) – Whether return all the paths or not. True by default. It's possible for the paths to have total actual sample size larger than batch_size, and will be truncated if this flag is true.

Returns

list[dict] – A list of sampled rollout paths. Each rollout path is a dictionary with the following keys:

- observations (numpy.ndarray)
- actions (numpy.ndarray)
- rewards (numpy.ndarray)
- agent_infos (dict)
- env_infos (dict)

slice_dict(*in_dict*, *slice_idx*)

Helper function to recursively parse through a dictionary of dictionaries and arrays to slice the arrays at a certain index.

Parameters

- **in_dict** (*dict*) – Dictionary where the values are arrays or other dictionaries that follow this stipulation.
- **slice_idx** (*int*) – Index to slice each array at.

Returns *dict* – Dictionary where arrays at every level are sliced.

ast_toolbox.samplers.batch_sampler module

Module for parallel sampling a batch of rollouts

class ast_toolbox.samplers.batch_sampler.**BatchSampler**(*algo*, *env*, *n_envs=1*, *open_loop=True*, *batch_simulate=False*, *sim=<ast_toolbox.simulators.example_av_simulator.e object>*, *reward_function=<ast_toolbox.rewards.example_av_rew object>*)

Bases: garage.sampler.base.BaseSampler

Collects samples in parallel using a stateful pool of workers.

Parameters

- **algo** (garage.np.algos.base.RLAlgorithm) – The algorithm.
- **env** (*ast_toolbox.envs.ASTEnv*) – The environment.
- **n_envs** (*int*) – Number of parallel environments to run.
- **open_loop** (*bool*) – True if the simulation is open-loop, meaning that AST must generate all actions ahead of time, instead of being able to output an action in sync with the simulator, getting an observation back before the next action is generated. False to get interactive control, which requires that *blackbox_sim_state* is also False.
- **batch_simulate** (*bool*) – When in *obtain_samples* with *open_loop == True*, the sampler will call *self.sim.batch_simulate_paths* if *batch_simulate* is True, and *self.sim.simulate* if False.
- **sim** (*ast_toolbox.simulators.ASTSimulator*) – The simulator wrapper, inheriting from *ast_toolbox.simulators.ASTSimulator*.

- **reward_function** (*ast_toolbox.rewards.ASTReward*) – The reward function, inheriting from *ast_toolbox.rewards.ASTReward*.
- **Args** – algo (garage.np.algos.RLAlgorithm): The algorithm. env (gym.Env): The environment.

obtain_samples(*itr*, *batch_size=None*, *whole_paths=True*)
Collect samples for the given iteration number.

Parameters

- **itr** (*int*) – Iteration number.
- **batch_size** (*int, optional*) – How many simulation steps to run in each epoch.
- **whole_paths** (*bool, optional*) – Whether to return the full rollout paths data.

shutdown_worker()
Terminate workers if necessary.

slice_dict(*in_dict*, *slice_idx*)
Helper function to recursively parse through a dictionary of dictionaries and arrays to slice the arrays at a certain index.

Parameters

- **in_dict** (*dict*) – Dictionary where the values are arrays or other dictionaries that follow this stipulation.
- **slice_idx** (*int*) – Index to slice each array at.

Returns *dict* – Dictionary where arrays at every level are sliced.

start_worker()
Initialize the sampler.

ast_toolbox.samplers.batch_sampler.**worker_init_tf**(*g*)
Initialize the tf.Session on a worker.

Parameters **g** (garage.sampler.stateful_pool.SharedGlobal) – SharedGlobal class from garage.sampler.stateful_pool.

ast_toolbox.samplers.batch_sampler.**worker_init_tf_vars**(*g*)
Initialize the policy parameters on a worker.

Parameters **g** (garage.sampler.stateful_pool.SharedGlobal) – SharedGlobal class from garage.sampler.stateful_pool.

ast_toolbox.samplers.parallel_sampler module

Original parallel sampler pool backend.

ast_toolbox.samplers.parallel_sampler.**close**()
Close the worker pool.

ast_toolbox.samplers.parallel_sampler.**initialize**(*n_parallel*)
Initialize the worker pool.

SIGINT is blocked for all processes created in parallel_sampler to avoid the creation of sleeping and zombie processes.

If the user interrupts run_experiment, there's a chance some processes won't die due to a dead lock condition where one of the children in the parallel sampler exits without releasing a lock once after it catches SIGINT.

Later the parent tries to acquire the same lock to proceed with his cleanup, but it remains sleeping waiting for the lock to be released. In the meantime, all the process in parallel sampler remain in the zombie state since the parent cannot proceed with their clean up.

Parameters **n_parallel** (*int*) – Number of workers to run in parallel.

ast_toolbox.samplers.parallel_sampler.**populate_task**(*env*, *policy*, *scope=None*)
Set each worker's env and policy.

Parameters

- **env** (*ast_toolbox.envs.ASTEnv*) – The environment.
- **policy** (garage.tf.policies.Policy) – The policy.
- **scope** (*str*) – Scope for identifying the algorithm. Must be specified if running multiple algorithms simultaneously, each using different environments and policies.

ast_toolbox.samplers.parallel_sampler.**sample_paths**(*policy_params*, *max_samples*, *max_path_length=inf*, *env_params=None*, *scope=None*)
Sample paths from each worker.

Parameters

- **policy_params** – parameters for the policy. This will be updated on each worker process
- **max_samples** (*int*) – desired maximum number of samples to be collected. The actual number of collected samples might be greater since all trajectories will be rolled out either until termination or until max_path_length is reached
- **max_path_length** (*int, optional*) – horizon / maximum length of a single trajectory
- **scope** (*str*) – Scope for identifying the algorithm. Must be specified if running multiple algorithms simultaneously, each using different environments and policies.

ast_toolbox.samplers.parallel_sampler.**set_seed**(*seed*)
Set the seed in each worker.

Parameters **seed** (*int*) – The random seed to be used by the worker.

ast_toolbox.samplers.parallel_sampler.**terminate_task**(*scope=None*)
Close each worker's env and terminate each policy.

Parameters **scope** (*str*) – Scope for identifying the algorithm. Must be specified if running multiple algorithms simultaneously, each using different environments and policies.

ast_toolbox.simulators package

Simulator wrappers to formulate validation as an AST RL problem

class ast_toolbox.simulators.**ASTSimulator**(*blackbox_sim_state=True*, *open_loop=True*, *fixed_initial_state=True*, *max_path_length=50*)
Bases: object

Class template to wrap a simulator for interaction with AST.

This class already tracks the simulator options to return the correct observation type. In addition, *max_path_length* and *self._path_length* are handled by this parent class.

Parameters

- **blackbox_sim_state** (*bool, optional*) – True if the true simulation state can not be observed, in which case actions and the initial conditions are used as the observation. False if the simulation state can be observed, in which case it will be used.
- **open_loop** (*bool, optional*) – True if the simulation is open-loop, meaning that AST must generate all actions ahead of time, instead of being able to output an action in sync with the simulator, getting an observation back before the next action is generated. False to get interactive control, which requires that *blackbox_sim_state* is also False.
- **fixed_init_state** (*bool, optional*) – True if the initial state is fixed, False to sample the initial state for each rollout from the observaation space.
- **max_path_length** (*int, optional*) – Maximum length of a single rollout.

clone_state()
Clone the simulator state for later resetting.

This function is used in conjunction with *restore_state* for Go-Explore and Backwards Algorithm to do their deterministic resets.

Returns *array_like* – An array of all the simulation state variables.

closed_loop_step(*action*)
User implemented function to step the simulation forward in time when closed-loop control is active.

This function should step the simulator forward a single timestep based on the given action. It will only be called when *open_loop* is False. This function should always return *self.observation_return()*.

Parameters **action** (*array_like*) – A 1-D array of actions taken by the AST Solver which deterministically control a single step forward in the simulation.

Returns *array_like* – An observation from the timestep, determined by the settings and the *observation_return* helper function.

get_reward_info()
Returns any info needed by the reward function to calculate the current reward.

is_goal()
Returns whether the current state is in the goal set. :returns: *bool* – True if current state is in goal set.

is_terminal()
Returns whether rollout horizon has been reached. :returns: *bool* – True if rollout horizon has been reached.

log()
perform any logging steps

observation_return()
Helper function to return the correct observation based on settings.

Returns *array_like* – An observation from the timestep, which is either from the simulator if *open_loop* is False and *blackbox_sim_state* is True, or else the initial conditions.

render(***kwargs*)
Either renders a simulation scene or returns data used for external rendering.

Parameters **kwargs** – Keyword arguments used in the simulators *render* function.

reset(*s_0*)
Resets the state of the environment, returning an initial observation.

User implementations should always call the super class implementation. This function should always return *self.observation_return()*.

Parameters **s_0** (*array_like*) – The initial conditions to reset the simulator to.

Returns *array_like* – An observation from the timestep, determined by the settings and the *observation_return* helper function.

restore_state(*in_simulator_state*)
Reset the simulation deterministically to a previously cloned state.

This function is used in conjunction with *clone_state* for Go-Explore and Backwards Algorithm to do their deterministic resets.

Parameters **in_simulator_state** (*array_like*) – An array of all the simulation state variables.

simulate(*actions*, *s_0*)
Run a full simulation given the AST solver's actions and initial conditions.

simulate takes in the AST solver's actions and the initial conditions. It should return two values: a terminal index and an array of relevant simulation information.

Parameters

- **actions** (*list[array_like]*) – A sequential list of actions taken by the AST Solver which deterministically control the simulation.
- **s_0** (*array_like*) – An array specifying the initial conditions to set the simulator to.

Returns

- **terminal_index** (*int*) – The index of the action that resulted in a state in the goal set E. If no state is found terminal_index should be returned as -1.
- *array_like* – An array of relevant simulator info, which can then be used for analysis or diagnostics.

step(*action*)
Step the simulation forward in time.

step takes in a the actions that deterministically control a single step forward in the simulation. It checks to see if the rollout horizon has been reached, and then calls *closed_loop_step* if the simulation is set to *open_loop == False*.

Parameters **action** (*array_like*) – A 1-D array of actions taken by the AST Solver which deterministically control a single step forward in the simulation.

Returns *array_like* – An observation from the timestep, which is either from the simulator if *open_loop* is False and *blackbox_sim_state* is True, or else the initial conditions.

class ast_toolbox.simulators.**ExampleAVSimulator**(*num_peds=1*, *simulator_args=None*, ***kwargs*)
Bases: *ast_toolbox.simulators.ast_simulator.ASTSimulator*

Example simulator wrapper for a scenario of an AV approaching a crosswalk where some pedestrians are crossing.

Wraps *ast_toolbox.simulators.example_av_simulator.ToyAVSimulator*

Parameters

- **num_peds** (*int*) – Number of pedestrians crossing the street.
- **simulator_args** (*dict*) – Dictionary of keyword arguments to be passed to the wrapped simulator.
- **kwargs** – Keyword arguments passed to the super class.

clone_state()
Clone the simulator state for later resetting.

This function is used in conjunction with *restore_state* for Go-Explore and Backwards Algorithm to do their deterministic resets.

Returns *array_like* – An array of all the simulation state variables.

closed_loop_step(*action*)
User implemented function to step the simulation forward in time when closed-loop control is active.

This function should step the simulator forward a single timestep based on the given action. It will only be called when *open_loop* is False. This function should always return *self.observation_return()*.

Parameters **action** (*array_like*) – A 1-D array of actions taken by the AST Solver which deterministically control a single step forward in the simulation.

Returns *array_like* – An observation from the timestep, determined by the settings and the *observation_return* helper function.

get_first_action()
An initialization method used in Go-Explore.

Returns *array_like* – A 1-D array of the same dimension as the action space, all zeros.

get_reward_info()
Returns any info needed by the reward function to calculate the current reward.

is_goal()
Returns whether the current state is in the goal set. :returns: *bool* – True if current state is in goal set.

log()
Perform any logging steps.

reset(*s_0*)
Resets the state of the environment, returning an initial observation.

User implementations should always call the super class implementation. This function should always return *self.observation_return()*.

Parameters **s_0** (*array_like*) – The initial conditions to reset the simulator to.

Returns *array_like* – An observation from the timestep, determined by the settings and the *observation_return* helper function.

restore_state(*in_simulator_state*)
Reset the simulation deterministically to a previously cloned state.

This function is used in conjunction with *clone_state* for Go-Explore and Backwards Algorithm to do their deterministic resets.

Parameters **in_simulator_state** (*array_like*) – An array of all the simulation state variables.

simulate(*actions*, *s_0*)
Run a full simulation given the AST solver's actions and initial conditions.

simulate takes in the AST solver's actions and the initial conditions. It should return two values: a terminal index and an array of relevant simulation information.

Parameters

- **actions** (*list[array_like]*) – A sequential list of actions taken by the AST Solver which deterministically control the simulation.
- **s_0** (*array_like*) – An array specifying the initial conditions to set the simulator to.

Returns

- **terminal_index** (*int*) – The index of the action that resulted in a state in the goal set E. If no state is found terminal_index should be returned as -1.
- *array_like* – An array of relevant simulator info, which can then be used for analysis or diagnostics.

Subpackages

ast_toolbox.simulators.example_av_simulator package

Toy AV simulator and an example AST simulator wrapper

class ast_toolbox.simulators.example_av_simulator.**ToyAVSimulator**(*num_peds=1*, *dt=0.1*, *alpha=0.85*, *beta=0.005*, *v_des=11.17*, *delta=4.0*, *t_headway=1.5*, *a_max=3.0*, *s_min=4.0*, *d_cmf=2.0*, *d_max=9.0*, *min_dist_x=2.5*, *min_dist_y=1.4*, *car_init_x=-35.0*, *car_init_y=0.0*)

Bases: `object`

A toy simulator of a scenario of an AV approaching a crosswalk where some pedestrians are crossing.

The vehicle runs a modified version of the Intelligent Driver Model[1]. The vehicle treats the closest pedestrian in the road as a car to follow. If no pedestrians are in the road, it attempts to maintain the desired speed. Noisy observations of the pedestrian are smoothed through an alpha-beta filter[2].

A collision results if any pedestrian's x-distance and y-distance to the ego vehicle are less than the respective *min_dist_x* and *min_dist_y*.

The origin is centered in the middle of the east/west lane and the north/south crosswalk. The positive x proceeds east down the lane, the positive y proceeds north across the crosswalk.

Parameters

- **num_peds** (*int*) – The number of pedestrians crossing the street.
- **dt** (*float*) – The length (in seconds) of each timestep.
- **alpha** (*float*) – The alpha parameter in the tracker's alpha-beta filter[2].
- **beta** (*float*) – The beta parameter in the tracker's alpha-beta filter[2].
- **v_des** (*float*) – The desired velocity, in meters per second, for the ego vehicle to maintain
- **delta** (*float*) – The delta parameter in the IDM algorithm[1].

[1] Treiber, Martin, Ansgar Hennecke, and Dirk Helbing. "Congested traffic states in empirical observations and microscopic simulations." Physical review E 62.2 (2000): 1805. https://journals.aps.org/pre/abstract/10.1103/PhysRevE.62.1805

[2] Rogers, Steven R. "Alpha-beta filter with correlated measurement noise." IEEE Transactions on Aerospace and Electronic Systems 4 (1987): 592-594. https://ieeexplore.ieee.org/abstract/document/4104388

- **t_headway** (*float*) – The headway parameter in the IDM algorithm[1].
- **a_max** (*float*) – The maximum acceleration parameter in the IDM algorithm[1].
- **s_min** (*float*) – The minimum follow distance parameter in the IDM algorithm[1].
- **d_cmf** (*float*) – The maximum comfortable deceleration parameter in the IDM algorithm[1].
- **d_max** (*float*) – The maximum deceleration parameter in the IDM algorithm[1].
- **min_dist_x** (*float*) – The minimum x-distance between the ego vehicle and a pedestrian.
- **min_dist_y** (*float*) – The minimum y-distance between the ego vehicle and a pedestrian.
- **car_init_x** (*float*) – The initial x-position of the ego vehicle.
- **car_init_y** (*float*) – The initial y-position of the ego vehicle.

References

collision_detected()
Returns whether the current state is in the goal set.

Checks to see if any pedestrian's position violates both the *min_dist_x* and *min_dist_y* constraints.

Returns *bool* – True if current state is in goal set.

get_ground_truth()
Clones the ground truth simulator state.

Returns *dict* – A dictionary of simulator state variables.

log()
Perform any logging steps.

move_car(*car*, *accel*)
Update the ego vehicle's state.

Parameters

- **car** (*array_like*) – The ego vehicle's state: [x-velocity, y-velocity, x-position, y-position].
- **accel** (*float*) – The ago vehicle's acceleration.

Returns *array_like* – An updated version of the ego vehicle's state.

observe()
Get the ground truth state of the pedestrian relative to the ego vehicle.

reset(*s_0*)
Resets the state of the environment, returning an initial observation.

Parameters **s_0** (*array_like*) – The initial conditions to reset the simulator to.

Returns *array_like* – An observation from the timestep, determined by the settings and the *observation_return* helper function.

run_simulation(*actions*, *s_0*, *simulation_horizon*)
Run a full simulation given the AST solver's actions and initial conditions.

Parameters

- **actions** (*list[array_like]*) – A sequential list of actions taken by the AST Solver which deterministically control the simulation.
- **s_0** (*array_like*) – An array specifying the initial conditions to set the simulator to.

- **simulation_horizon** (*int*) – The maximum number of steps a simulation rollout is allowed to run.

Returns

- **terminal_index** (*int*) – The index of the action that resulted in a state in the goal set E. If no state is found terminal_index should be returned as -1.
- *array_like* – An array of relevant simulator info, which can then be used for analysis or diagnostics.

sensors (*peds*, *noise*)
Get a noisy observation of the pedestrians' locations and velocities.

Parameters

- **peds** (*array_like*) – Positions and velocities of the pedestrians.
- **noise** (*array_like*) – Noise to add to the positions and velocities of the pedestrians.

Returns *array_like* – Noisy observation of the pedestrians' locations and velocities.

set_ground_truth (*in_simulator_state*)
Sets the simulator state variables.

Parameters **in_simulator_state** (*dict*) – A dictionary of simulator state variables.

step_simulation (*action*)
Handle anything that needs to take place at each step, such as a simulation update or write to file.

Parameters **action** (*array_like*) – A 1-D array of actions taken by the AST Solver which deterministically control a single step forward in the simulation.

Returns *array_like* – An observation from the timestep, determined by the settings and the *observation_return* helper function.

tracker (*estimate_old*, *measurements*)
An alpha-beta filter to smooth noisy observations into an estimate of pedestrian state.

Parameters

- **estimate_old** (*array_like*) – The smoothed state estimate from the previous timestep.
- **measurements** (*array_like*) – The noisy observation of pedestrian state from the current timestep.

Returns *array_like* – The smoothed state estimate of pedestrian state from the current timestep.

update_car (*obs*, *v_car*)
Calculate the ego vehicle's acceleration.

Parameters

- **obs** (*array_like*) – Smoothed estimate of pedestrian state from the *tracker*.
- **v_car** (*float*) – Current velocity of the ego vehicle.

Returns *float* – The acceleration of the ego vehicle.

update_peds ()
Update the pedestrian's state.

class ast_toolbox.simulators.example_av_simulator.**ExampleAVSimulator**(*num_peds=1*, *simulator_args=None*, ***kwargs*)

Bases: *ast_toolbox.simulators.ast_simulator.ASTSimulator*

Example simulator wrapper for a scenario of an AV approaching a crosswalk where some pedestrians are crossing.

Wraps *ast_toolbox.simulators.example_av_simulator.ToyAVSimulator*

Parameters

- **num_peds** (*int*) – Number of pedestrians crossing the street.
- **simulator_args** (*dict*) – Dictionary of keyword arguments to be passed to the wrapped simulator.
- **kwargs** – Keyword arguments passed to the super class.

clone_state()
Clone the simulator state for later resetting.

This function is used in conjunction with *restore_state* for Go-Explore and Backwards Algorithm to do their deterministic resets.

Returns *array_like* – An array of all the simulation state variables.

closed_loop_step(*action*)
User implemented function to step the simulation forward in time when closed-loop control is active.

This function should step the simulator forward a single timestep based on the given action. It will only be called when *open_loop* is False. This function should always return *self.observation_return()*.

Parameters **action** (*array_like*) – A 1-D array of actions taken by the AST Solver which deterministically control a single step forward in the simulation.

Returns *array_like* – An observation from the timestep, determined by the settings and the *observation_return* helper function.

get_first_action()
An initialization method used in Go-Explore.

Returns *array_like* – A 1-D array of the same dimension as the action space, all zeros.

get_reward_info()
Returns any info needed by the reward function to calculate the current reward.

is_goal()
Returns whether the current state is in the goal set. :returns: *bool* – True if current state is in goal set.

log()
Perform any logging steps.

reset(*s_0*)
Resets the state of the environment, returning an initial observation.

User implementations should always call the super class implementation. This function should always return *self.observation_return()*.

Parameters **s_0** (*array_like*) – The initial conditions to reset the simulator to.

Returns *array_like* – An observation from the timestep, determined by the settings and the *observation_return* helper function.

restore_state(*in_simulator_state*)
Reset the simulation deterministically to a previously cloned state.

This function is used in conjunction with *clone_state* for Go-Explore and Backwards Algorithm to do their deterministic resets.

Parameters **in_simulator_state** (*array_like*) – An array of all the simulation state variables.

simulate(*actions*, *s_0*)
Run a full simulation given the AST solver's actions and initial conditions.

simulate takes in the AST solver's actions and the initial conditions. It should return two values: a terminal index and an array of relevant simulation information.

Parameters

- **actions** (*list[array_like]*) – A sequential list of actions taken by the AST Solver which deterministically control the simulation.
- **s_0** (*array_like*) – An array specifying the initial conditions to set the simulator to.

Returns

- **terminal_index** (*int*) – The index of the action that resulted in a state in the goal set E. If no state is found terminal_index should be returned as -1.
- *array_like* – An array of relevant simulator info, which can then be used for analysis or diagnostics.

Submodules

ast_toolbox.simulators.example_av_simulator.example_av_simulator module

Example simulator wrapper for a scenario of an AV approaching a crosswalk where some pedestrians are crossing.

class ast_toolbox.simulators.example_av_simulator.example_av_simulator.**ExampleAVSimulator**(

Bases: *ast_toolbox.simulators.ast_simulator.ASTSimulator*

Example simulator wrapper for a scenario of an AV approaching a crosswalk where some pedestrians are crossing.

Wraps *ast_toolbox.simulators.example_av_simulator.ToyAVSimulator*

Parameters

- **num_peds** (*int*) – Number of pedestrians crossing the street.
- **simulator_args** (*dict*) – Dictionary of keyword arguments to be passed to the wrapped simulator.
- **kwargs** – Keyword arguments passed to the super class.

clone_state()
Clone the simulator state for later resetting.

This function is used in conjunction with *restore_state* for Go-Explore and Backwards Algorithm to do their deterministic resets.

Returns *array_like* – An array of all the simulation state variables.

closed_loop_step(*action*)
User implemented function to step the simulation forward in time when closed-loop control is active.

This function should step the simulator forward a single timestep based on the given action. It will only be called when *open_loop* is False. This function should always return *self.observation_return()*.

Parameters **action** (*array_like*) – A 1-D array of actions taken by the AST Solver which deterministically control a single step forward in the simulation.

Returns *array_like* – An observation from the timestep, determined by the settings and the *observation_return* helper function.

get_first_action()
An initialization method used in Go-Explore.

Returns *array_like* – A 1-D array of the same dimension as the action space, all zeros.

get_reward_info()
Returns any info needed by the reward function to calculate the current reward.

is_goal()
Returns whether the current state is in the goal set. :returns: *bool* – True if current state is in goal set.

log()
Perform any logging steps.

reset(*s_0*)
Resets the state of the environment, returning an initial observation.

User implementations should always call the super class implementation. This function should always return *self.observation_return()*.

Parameters **s_0** (*array_like*) – The initial conditions to reset the simulator to.

Returns *array_like* – An observation from the timestep, determined by the settings and the *observation_return* helper function.

restore_state(*in_simulator_state*)
Reset the simulation deterministically to a previously cloned state.

This function is used in conjunction with *clone_state* for Go-Explore and Backwards Algorithm to do their deterministic resets.

Parameters **in_simulator_state** (*array_like*) – An array of all the simulation state variables.

simulate(*actions*, *s_0*)
Run a full simulation given the AST solver's actions and initial conditions.

simulate takes in the AST solver's actions and the initial conditions. It should return two values: a terminal index and an array of relevant simulation information.

Parameters

- **actions** (*list[array_like]*) – A sequential list of actions taken by the AST Solver which deterministically control the simulation.
- **s_0** (*array_like*) – An array specifying the initial conditions to set the simulator to.

Returns

- **terminal_index** (*int*) – The index of the action that resulted in a state in the goal set E. If no state is found terminal_index should be returned as -1.
- *array_like* – An array of relevant simulator info, which can then be used for analysis or diagnostics.

ast_toolbox.simulators.example_av_simulator.toy_av_simulator module

A toy simulator of a scenario of an AV approaching a crosswalk where some pedestrians are crossing.

class ast_toolbox.simulators.example_av_simulator.toy_av_simulator.**ToyAVSimulator**(*num_peds=1*, *dt=0.1*, *alpha=0.85*, *beta=0.005*, *v_des=11.17*, *delta=4.0*, *t_headway=*, *a_max=3.0*, *s_min=4.0*, *d_cmf=2.0*, *d_max=9.0*, *min_dist_x=*, *min_dist_y=*, *car_init_x=-35.0*, *car_init_y=0*)

Bases: object

A toy simulator of a scenario of an AV approaching a crosswalk where some pedestrians are crossing.

The vehicle runs a modified version of the Intelligent Driver Model[1]. The vehicle treats the closest pedestrian in the road as a car to follow. If no pedestrians are in the road, it attempts to maintain the desired speed. Noisy observations of the pedestrian are smoothed through an alpha-beta filter[2].

A collision results if any pedestrian's x-distance and y-distance to the ego vehicle are less than the respective *min_dist_x* and *min_dist_y*.

The origin is centered in the middle of the east/west lane and the north/south crosswalk. The positive x proceeds east down the lane, the positive y proceeds north across the crosswalk.

Parameters

- **num_peds** (*int*) – The number of pedestrians crossing the street.
- **dt** (*float*) – The length (in seconds) of each timestep.
- **alpha** (*float*) – The alpha parameter in the tracker's alpha-beta filter[2].
- **beta** (*float*) – The beta parameter in the tracker's alpha-beta filter[2].
- **v_des** (*float*) – The desired velocity, in meters per second, for the ego vehicle to maintain
- **delta** (*float*) – The delta parameter in the IDM algorithm[1].
- **t_headway** (*float*) – The headway parameter in the IDM algorithm[1].
- **a_max** (*float*) – The maximum acceleration parameter in the IDM algorithm[1].
- **s_min** (*float*) – The minimum follow distance parameter in the IDM algorithm[1].
- **d_cmf** (*float*) – The maximum comfortable deceleration parameter in the IDM algorithm[1].
- **d_max** (*float*) – The maximum deceleration parameter in the IDM algorithm[1].
- **min_dist_x** (*float*) – The minimum x-distance between the ego vehicle and a pedestrian.
- **min_dist_y** (*float*) – The minimum y-distance between the ego vehicle and a pedestrian.
- **car_init_x** (*float*) – The initial x-position of the ego vehicle.

[1] Treiber, Martin, Ansgar Hennecke, and Dirk Helbing. "Congested traffic states in empirical observations and microscopic simulations." Physical review E 62.2 (2000): 1805. https://journals.aps.org/pre/abstract/10.1103/PhysRevE.62.1805

[2] Rogers, Steven R. "Alpha-beta filter with correlated measurement noise." IEEE Transactions on Aerospace and Electronic Systems 4 (1987): 592-594. https://ieeexplore.ieee.org/abstract/document/4104388

- **car_init_y** (*float*) – The initial y-position of the ego vehicle.

References

collision_detected()
Returns whether the current state is in the goal set.

Checks to see if any pedestrian's position violates both the *min_dist_x* and *min_dist_y* constraints.

Returns *bool* – True if current state is in goal set.

get_ground_truth()
Clones the ground truth simulator state.

Returns *dict* – A dictionary of simulator state variables.

log()
Perform any logging steps.

move_car(*car*, *accel*)
Update the ego vehicle's state.

Parameters

- **car** (*array_like*) – The ego vehicle's state: [x-velocity, y-velocity, x-position, y-position].
- **accel** (*float*) – The ago vehicle's acceleration.

Returns *array_like* – An updated version of the ego vehicle's state.

observe()
Get the ground truth state of the pedestrian relative to the ego vehicle.

reset(*s_0*)
Resets the state of the environment, returning an initial observation.

Parameters **s_0** (*array_like*) – The initial conditions to reset the simulator to.

Returns *array_like* – An observation from the timestep, determined by the settings and the *observation_return* helper function.

run_simulation(*actions*, *s_0*, *simulation_horizon*)
Run a full simulation given the AST solver's actions and initial conditions.

Parameters

- **actions** (*list[array_like]*) – A sequential list of actions taken by the AST Solver which deterministically control the simulation.
- **s_0** (*array_like*) – An array specifying the initial conditions to set the simulator to.
- **simulation_horizon** (*int*) – The maximum number of steps a simulation rollout is allowed to run.

Returns

- **terminal_index** (*int*) – The index of the action that resulted in a state in the goal set E. If no state is found terminal_index should be returned as -1.
- *array_like* – An array of relevant simulator info, which can then be used for analysis or diagnostics.

sensors(*peds*, *noise*)
Get a noisy observation of the pedestrians' locations and velocities.

Parameters

- **peds** (*array_like*) – Positions and velocities of the pedestrians.
- **noise** (*array_like*) – Noise to add to the positions and velocities of the pedestrians.

Returns *array_like* – Noisy observation of the pedestrians' locations and velocities.

set_ground_truth(*in_simulator_state*)
Sets the simulator state variables.

Parameters **in_simulator_state** (*dict*) – A dictionary of simulator state variables.

step_simulation(*action*)
Handle anything that needs to take place at each step, such as a simulation update or write to file.

Parameters **action** (*array_like*) – A 1-D array of actions taken by the AST Solver which deterministically control a single step forward in the simulation.

Returns *array_like* – An observation from the timestep, determined by the settings and the *observation_return* helper function.

tracker(*estimate_old*, *measurements*)
An alpha-beta filter to smooth noisy observations into an estimate of pedestrian state.

Parameters

- **estimate_old** (*array_like*) – The smoothed state estimate from the previous timestep.
- **measurements** (*array_like*) – The noisy observation of pedestrian state from the current timestep.

Returns *array_like* – The smoothed state estimate of pedestrian state from the current timestep.

update_car(*obs*, *v_car*)
Calculate the ego vehicle's acceleration.

Parameters

- **obs** (*array_like*) – Smoothed estimate of pedestrian state from the *tracker*.
- **v_car** (*float*) – Current velocity of the ego vehicle.

Returns *float* – The acceleration of the ego vehicle.

update_peds()
Update the pedestrian's state.

Submodules

ast_toolbox.simulators.ast_simulator module

Class template to wrap a simulator for interaction with AST.

class ast_toolbox.simulators.ast_simulator.**ASTSimulator**(*blackbox_sim_state=True*, *open_loop=True*, *fixed_initial_state=True*, *max_path_length=50*)

Bases: object

Class template to wrap a simulator for interaction with AST.

This class already tracks the simulator options to return the correct observation type. In addition, *max_path_length* and *self._path_length* are handled by this parent class.

Parameters

- **blackbox_sim_state** (*bool, optional*) – True if the true simulation state can not be observed, in which case actions and the initial conditions are used as the observation. False if the simulation state can be observed, in which case it will be used.
- **open_loop** (*bool, optional*) – True if the simulation is open-loop, meaning that AST must generate all actions ahead of time, instead of being able to output an action in sync with the simulator, getting an observation back before the next action is generated. False to get interactive control, which requires that *blackbox_sim_state* is also False.
- **fixed_init_state** (*bool, optional*) – True if the initial state is fixed, False to sample the initial state for each rollout from the observaation space.
- **max_path_length** (*int, optional*) – Maximum length of a single rollout.

clone_state()
Clone the simulator state for later resetting.

This function is used in conjunction with *restore_state* for Go-Explore and Backwards Algorithm to do their deterministic resets.

Returns *array_like* – An array of all the simulation state variables.

closed_loop_step(*action*)
User implemented function to step the simulation forward in time when closed-loop control is active.

This function should step the simulator forward a single timestep based on the given action. It will only be called when *open_loop* is False. This function should always return *self.observation_return()*.

Parameters **action** (*array_like*) – A 1-D array of actions taken by the AST Solver which deterministically control a single step forward in the simulation.

Returns *array_like* – An observation from the timestep, determined by the settings and the *observation_return* helper function.

get_reward_info()
Returns any info needed by the reward function to calculate the current reward.

is_goal()
Returns whether the current state is in the goal set. :returns: *bool* – True if current state is in goal set.

is_terminal()
Returns whether rollout horizon has been reached. :returns: *bool* – True if rollout horizon has been reached.

log()
perform any logging steps

observation_return()
Helper function to return the correct observation based on settings.

Returns *array_like* – An observation from the timestep, which is either from the simulator if *open_loop* is False and *blackbox_sim_state* is True, or else the initial conditions.

render(***kwargs*)
Either renders a simulation scene or returns data used for external rendering.

Parameters **kwargs** – Keyword arguments used in the simulators *render* function.

reset(*s_0*)
Resets the state of the environment, returning an initial observation.

User implementations should always call the super class implementation. This function should always return *self.observation_return()*.

Parameters **s_0** (*array_like*) – The initial conditions to reset the simulator to.

Returns *array_like* – An observation from the timestep, determined by the settings and the *observation_return* helper function.

restore_state(*in_simulator_state*)
Reset the simulation deterministically to a previously cloned state.

This function is used in conjunction with *clone_state* for Go-Explore and Backwards Algorithm to do their deterministic resets.

Parameters **in_simulator_state** (*array_like*) – An array of all the simulation state variables.

simulate(*actions*, *s_0*)
Run a full simulation given the AST solver's actions and initial conditions.

simulate takes in the AST solver's actions and the initial conditions. It should return two values: a terminal index and an array of relevant simulation information.

Parameters

- **actions** (*list[array_like]*) – A sequential list of actions taken by the AST Solver which deterministically control the simulation.
- **s_0** (*array_like*) – An array specifying the initial conditions to set the simulator to.

Returns

- **terminal_index** (*int*) – The index of the action that resulted in a state in the goal set E. If no state is found terminal_index should be returned as -1.
- *array_like* – An array of relevant simulator info, which can then be used for analysis or diagnostics.

step(*action*)
Step the simulation forward in time.

step takes in a the actions that deterministically control a single step forward in the simulation. It checks to see if the rollout horizon has been reached, and then calls *closed_loop_step* if the simulation is set to *open_loop == False*.

Parameters **action** (*array_like*) – A 1-D array of actions taken by the AST Solver which deterministically control a single step forward in the simulation.

Returns *array_like* – An observation from the timestep, which is either from the simulator if *open_loop* is False and *blackbox_sim_state* is True, or else the initial conditions.

ast_toolbox.spaces package

Action and State Spaces to formulate validation as an AST RL problem

class ast_toolbox.spaces.**ASTSpaces**
Bases: object

Class to define the action and observation spaces of an AST problem.

Both the *action_space* and the *observation_space* should be a gym.spaces.Space type.

The *action_space* is only used to clip actions if *ASTEnv* is wrapped by the normalize env.

If using *ASTEnv* with *blackbox_sim_state == True*, *observation_space* should define the space for each simulation state variable. Otherwise, it should define the space of initial condition variables.

If using *ASTEnv* with *fixed_init_state == False*, the initial conditions of each rollout will be randomly sampled at uniform from the observation_space.

action_space
Returns a definition of the action space of the reinforcement learning problem.

Returns

gym.spaces.Space – The action space of the reinforcement learning problem.

observation_space
Returns a definition of the observation space of the reinforcement learning problem.

Returns

gym.spaces.Space – The observation space of the reinforcement learning problem.

class ast_toolbox.spaces.**ExampleAVSpaces**(*num_peds=1*, *max_path_length=50*, *v_des=11.17*, *x_accel_low=-1.0*, *y_accel_low=-1.0*, *x_accel_high=1.0*, *y_accel_high=1.0*, *x_boundary_low=-10.0*, *y_boundary_low=-10.0*, *x_boundary_high=10.0*, *y_boundary_high=10.0*, *x_v_low=-10.0*, *y_v_low=-10.0*, *x_v_high=10.0*, *y_v_high=10.0*, *car_init_x=-35.0*, *car_init_y=0.0*, *open_loop=True*)

Bases: *ast_toolbox.spaces.ast_spaces.ASTSpaces*

Class to define the action and observation spaces for an example AV validation task.

Parameters

- **num_peds** (*int, optional*) – The number of pedestrians crossing the street.
- **max_path_length** (*int, optional*) – Maximum length of a single rollout.
- **v_des** (*float, optional*) – The desired velocity, in meters per second, for the ego vehicle to maintain
- **x_accel_low** (*float, optional*) – The minimum x-acceleration of the pedestrian.
- **y_accel_low** (*float, optional*) – The minimum y-acceleration of the pedestrian.
- **x_accel_high** (*float, optional*) – The maximum x-acceleration of the pedestrian.
- **y_accel_high** (*float, optional*) – The maximum y-acceleration of the pedestrian.
- **x_boundary_low** (*float, optional*) – The minimum x-position of the pedestrian.
- **y_boundary_low** (*float, optional*) – The minimum y-position of the pedestrian.
- **x_boundary_high** (*float, optional*) – The maximum x-position of the pedestrian.
- **y_boundary_high** (*float, optional*) – The maximum y-position of the pedestrian.
- **x_v_low** (*float, optional*) – The minimum x-velocity of the pedestrian.
- **y_v_low** (*float, optional*) – The minimum y-velocity of the pedestrian.
- **x_v_high** (*float, optional*) – The maximum x-velocity of the pedestrian.
- **y_v_high** (*float, optional*) – The maximum y-velocity of the pedestrian.
- **car_init_x** (*float, optional*) – The initial x-position of the ego vehicle.
- **car_init_y** (*float, optional*) – The initial y-position of the ego vehicle.

- **open_loop** (*bool, optional*) – True if the simulation is open-loop, meaning that AST must generate all actions ahead of time, instead of being able to output an action in sync with the simulator, getting an observation back before the next action is generated. False to get interactive control, which requires that *blackbox_sim_state* is also False.

action_space
Returns a definition of the action space of the reinforcement learning problem.

Returns

gym.spaces.Space – The action space of the reinforcement learning problem.

observation_space
Returns a definition of the observation space of the reinforcement learning problem.

Returns

gym.spaces.Space – The observation space of the reinforcement learning problem.

Submodules

ast_toolbox.spaces.ast_spaces module

Class to define the action and observation spaces of an AST problem.

class ast_toolbox.spaces.ast_spaces.**ASTSpaces**
Bases: object

Class to define the action and observation spaces of an AST problem.

Both the *action_space* and the *observation_space* should be a gym.spaces.Space type.

The *action_space* is only used to clip actions if *ASTEnv* is wrapped by the normalize env.

If using *ASTEnv* with *blackbox_sim_state == True*, *observation_space* should define the space for each simulation state variable. Otherwise, it should define the space of initial condition variables.

If using *ASTEnv* with *fixed_init_state == False*, the initial conditions of each rollout will be randomly sampled at uniform from the observation_space.

action_space
Returns a definition of the action space of the reinforcement learning problem.

Returns

gym.spaces.Space – The action space of the reinforcement learning problem.

observation_space
Returns a definition of the observation space of the reinforcement learning problem.

Returns

gym.spaces.Space – The observation space of the reinforcement learning problem.

ast_toolbox.spaces.example_av_spaces module

Class to define the action and observation spaces for an example AV validation task.

class ast_toolbox.spaces.example_av_spaces.**ExampleAVSpaces**(*num_peds=1*, *max_path_length=50*, *v_des=11.17*, *x_accel_low=-1.0*, *y_accel_low=-1.0*, *x_accel_high=1.0*, *y_accel_high=1.0*, *x_boundary_low=-10.0*, *y_boundary_low=-10.0*, *x_boundary_high=10.0*, *y_boundary_high=10.0*, *x_v_low=-10.0*, *y_v_low=-10.0*, *x_v_high=10.0*, *y_v_high=10.0*, *car_init_x=-35.0*, *car_init_y=0.0*, *open_loop=True*)

Bases: *ast_toolbox.spaces.ast_spaces.ASTSpaces*

Class to define the action and observation spaces for an example AV validation task.

Parameters

- **num_peds** (*int, optional*) – The number of pedestrians crossing the street.
- **max_path_length** (*int, optional*) – Maximum length of a single rollout.
- **v_des** (*float, optional*) – The desired velocity, in meters per second, for the ego vehicle to maintain
- **x_accel_low** (*float, optional*) – The minimum x-acceleration of the pedestrian.
- **y_accel_low** (*float, optional*) – The minimum y-acceleration of the pedestrian.
- **x_accel_high** (*float, optional*) – The maximum x-acceleration of the pedestrian.
- **y_accel_high** (*float, optional*) – The maximum y-acceleration of the pedestrian.
- **x_boundary_low** (*float, optional*) – The minimum x-position of the pedestrian.
- **y_boundary_low** (*float, optional*) – The minimum y-position of the pedestrian.
- **x_boundary_high** (*float, optional*) – The maximum x-position of the pedestrian.
- **y_boundary_high** (*float, optional*) – The maximum y-position of the pedestrian.
- **x_v_low** (*float, optional*) – The minimum x-velocity of the pedestrian.
- **y_v_low** (*float, optional*) – The minimum y-velocity of the pedestrian.
- **x_v_high** (*float, optional*) – The maximum x-velocity of the pedestrian.
- **y_v_high** (*float, optional*) – The maximum y-velocity of the pedestrian.
- **car_init_x** (*float, optional*) – The initial x-position of the ego vehicle.
- **car_init_y** (*float, optional*) – The initial y-position of the ego vehicle.
- **open_loop** (*bool, optional*) – True if the simulation is open-loop, meaning that AST must generate all actions ahead of time, instead of being able to output an action in sync with

the simulator, getting an observation back before the next action is generated. False to get interactive control, which requires that *blackbox_sim_state* is also False.

action_space
Returns a definition of the action space of the reinforcement learning problem.

Returns gym.spaces.Space – The action space of the reinforcement learning problem.

observation_space
Returns a definition of the observation space of the reinforcement learning problem.

Returns

gym.spaces.Space – The observation space of the reinforcement learning problem.

ast_toolbox.utils package

Utility functions for running and analyzing AST problems

Submodules

ast_toolbox.utils.analysis_utils module

ast_toolbox.utils.analysis_utils.**render_itr_heatmap**(*samples_data*, *visit_counts*, *fig=None*, *ax=None*)

ast_toolbox.utils.analysis_utils.**render_paths**(*filepath*, *gif_file=None*)

ast_toolbox.utils.analysis_utils.**render_paths_heatmap_gif**(*filepath*, *gif_file=None*, *frames=10*)

ast_toolbox.utils.exp_utils module

ast_toolbox.utils.exp_utils.**log_mean_exp**(*x*, *dim*)
Compute the log(mean(exp(x), dim)) in a numerically stable manner

ast_toolbox.utils.exp_utils.**log_sum_exp**(*x*, *dim*)
Compute the log(sum(exp(x), dim)) in a numerically stable manner

ast_toolbox.utils.exp_utils.**softmax**(*x*, *dim*)
Compute softmax values for each sets of scores in x along dim

ast_toolbox.utils.ga_argparser module

ast_toolbox.utils.ga_argparser.**get_ga_parser**(*log_dir='./'*)

ast_toolbox.utils.go_explore_utils module

ast_toolbox.utils.go_explore_utils.**convert_drl_itr_data_to_expert_trajectory**(*last_iter_data*)

ast_toolbox.utils.go_explore_utils.**convert_mcts_itr_data_to_expert_trajectory**(*best_actions*, *sim*, *s_0*, *reward_function*)

ast_toolbox.utils.go_explore_utils.**get_cellpool**(*filename*, *dbname=None*, *dbtype=<sphinx.ext.autodoc.importer._MockObject object>*, *flags=<sphinx.ext.autodoc.importer._MockObject object>*, *protocol=4*)

ast_toolbox.utils.go_explore_utils.**get_meta_filename**(*filename*)

ast_toolbox.utils.go_explore_utils.**get_metadata**(*filename*)

ast_toolbox.utils.go_explore_utils.**get_pool_filename**(*filename*)

ast_toolbox.utils.go_explore_utils.**get_root_cell**(*pool*, *cell*)

ast_toolbox.utils.go_explore_utils.**load_convert_and_save_drl_expert_trajectory**(*last_iter_filenam*
expert_trajectory_f

ast_toolbox.utils.go_explore_utils.**load_convert_and_save_mcts_expert_trajectory**(*best_actions_fil*
*expert_trajectory_*
sim,
s_0,
reward_function)

ast_toolbox.utils.go_explore_utils.**plot_goal_trajectories**(*filename*, *goal_limit=None*, *sort_by_reward=False*)

ast_toolbox.utils.go_explore_utils.**plot_terminal_trajectories**(*filename*, *terminal_limit=None*, *sort_by_reward=False*)

ast_toolbox.utils.go_explore_utils.**plot_trajectories**(*filename*, *plot_terminal=True*, *plot_goal=True*, *terminal_limit=None*, *goal_limit=None*, *sort_by_reward=False*)

ast_toolbox.utils.go_explore_utils.**render**(*car=None*, *ped=None*, *noise=None*, *ped_obs=None*, *gif=False*)

ast_toolbox.utils.mcts_utils module

class ast_toolbox.utils.mcts_utils.**StateActionNode**
Bases: object

class ast_toolbox.utils.mcts_utils.**StateActionStateNode**
Bases: object

class ast_toolbox.utils.mcts_utils.**StateNode**
Bases: object

ast_toolbox.utils.np_weight_init module

ast_toolbox.utils.np_weight_init.**init_param_np**(*param*, *policy*, *np_random=<module 'numpy.random' from '/home/docs/checkouts/readthedocs.org/user_builds/ast-toolbox/envs/latest/lib/python3.7/site-packages/numpy/random/__init__.py'>*)

ast_toolbox.utils.np_weight_init.**init_policy_np**(*policy*, *np_random=<module 'numpy.random' from '/home/docs/checkouts/readthedocs.org/user_builds/ast-toolbox/envs/latest/lib/python3.7/site-packages/numpy/random/__init__.py'>*)

ast_toolbox.utils.seeding module

ast_toolbox.utils.seeding.**create_seed**(*a=None*, *max_bytes=8*)

Create a strong random seed. Otherwise, Python 2 would seed using the system time, which might be non-robust especially in the presence of concurrency.

Args: a (Optional[int, str]): None seeds from an operating system specific randomness source. max_bytes: Maximum number of bytes to use in the seed.

ast_toolbox.utils.seeding.**hash_seed**(*seed=None*, *max_bytes=8*)

Any given evaluation is likely to have many PRNG's active at once. (Most commonly, because the environment is running in multiple processes.) There's literature indicating that having linear correlations between seeds of multiple PRNG's can correlate the outputs:

http://blogs.unity3d.com/2015/01/07/a-primer-on-repeatable-random-numbers/ http://stackoverflow.com/questions/1554958/how-different-do-random-seeds-need-to-be http://dl.acm.org/citation.cfm?id=1276928

Thus, for sanity we hash the seeds before using them. (This scheme is likely not crypto-strength, but it should be good enough to get rid of simple correlations.)

Args: seed (Optional[int]): None seeds from an operating system specific randomness source. max_bytes: Maximum number of bytes to use in the hashed seed.

ast_toolbox.utils.seeding.**np_random**(*seed=None*)

ast_toolbox.utils.tree_plot module

ast_toolbox.utils.tree_plot.**add_children**(*s*, *s_node*, *tree*, *graph*, *d*)

ast_toolbox.utils.tree_plot.**get_node_num_next**(*s*, *tree*, *depths*, *nodeNums*, *d*)

ast_toolbox.utils.tree_plot.**get_root**(*tree*)

ast_toolbox.utils.tree_plot.**plot_node_num**(*tree*, *path*, *format='svg'*)

ast_toolbox.utils.tree_plot.**plot_tree**(*tree*, *d*, *path*, *format='svg'*)

ast_toolbox.utils.tree_plot.**s2node**(*s*, *tree*)

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W