Hybrid System Falsification and Reinforcement Learning

Home assignment

David Sprunger and Clovis Eberhart

July 29, 2019

To hand in at the latest by: August 19, 2019, 00:00.
How to hand in: email both at `first_name[dot]family_name[at]gmail[dot]com`.

1 Overview

1.1 Preliminaries

To run the code in this repository, you need a version of Python 3. You can install the packages called here automatically with pip, call `pip install -r requirements.txt` from the root of this repository. If you have any troubles getting things to work, please don't hesitate to email.

(*) Below, any time we talk about running a script, we are assuming you are using the root of this repository as the current working directory.

1.2 Big picture

In this assignment, we will be taking a look at a falsification benchmark from "Linear Hybrid System Falsification Through Descent" by Abbas and Fainekos. You can get an idea of how this benchmark works by running `python -m models.navigation` from the root of this repository*. This should launch an animation with a 4x4 grid and show the trace of three random particles moving through the space one after the next.

In this benchmark, we are allowed to create a particle in the green box in the upper left hand corner with some bounded initial velocity—after this particle is created it flows entirely according to the dynamics of the the environment, no further control is possible.

Our goal is to find an initial condition on the particle (position and velocity in x and y) so that the particle traces out a curve that comes as close to the red circle in the bottom right hand corner as possible. In order to do this, we defined a formula saying that if the particle starts at a position in the green box, then it does *not* enter the red circle within 20 seconds. Falsifying this formula would therefore accomplish our goal.

2 What to do

You will need to do complete three pieces of code for this assignment.

1. The robustness_semantics function in stl/semantics.py (line 77). This function computes the robustness semantics of a formula on a given signal at a given time. You may draw inspiration from the boolean_semantics function.
2. The nelder_mead_generate function in optimisation.py. This function generates the next sample to try according to the Nelder-Mead algorithm. You may draw inspiration from the nelder_mead_loop function in nelder_mead.py, from the rest of nelder_mead_generate and from nelder_mead_update. Looking at other complete optimisation procedures in optimisation.py might also help.
3. The policy function for AssignmentSolutionAgent in rl/part3.py (line 46). You should both set self.prev_action and return it to cause the agent to use that action. You may want to consult the implementation of other agents from rl/agent.py.

3 Global architecture

The zipfile at `http://group-mmm.org/~eberhart/` is organized into several packages and files. We describe these below, *italicising* files that may be useful to understand for the assignment, and **bolding** files that need your work.

1. The `models` package contains our implementation of several CPS models. Of these, the main one used in this assignment is navigation.py, but you may be interested to play with some of the others. In more detail, it has the following files:
   - aircraft.py: contains the implementation of an aircraft model, a formula to falsify (the input is a signal), and an example of output signal with a simple input signal when launched as the main module.
   - graphics.py: a graphics library, written by John Zelle (no need to understand), used to plot trajectories in the navigation benchmark.
   - *navigation.py*: the main benchmark for this assignment. It defines a model to run, a formula to falsify (the input is a position), and some example trajectories are drawn when it is launched as the main module.
   - rail.py: contains the implementation of a simple robot on a rail, a formula to falsify (the input is a signal), and example behaviours when launched as the main module.
   - robot.py: contains the implementation of a free-flying robot, a formula to falsify (the input is a signal), and example behaviours when launched as the main module.
2. The `rl` package contains infrastructure for running reinforcement learning experiments, with an eye towards using CPS models as environments for agents. In more detail, it has the following files:
   - *agent.py*: implements several reinforcement learning agents.
   - environment.py: implements several environments in which to perform reinforcement learning
   - *gadget.py*: state-tracking devices used by agents to keep track of statistics in environments.
   - gpi.py: some generalized policy iteration code for a policy iterator agent.
   - gym.py: facilities for automatically running agents in environments.
   - **part3.py**: more agents, including the agent we created in class and the agent you will finish for this assignment.
3. The `stl` package includes definitions for the principal items in signal temporal logic, including formulas, signals, and semantics.
   - *formula.py*: a library defining formulas. It prints some formulas when launched as the main module.
   - **semantics.py**: contains the implementations of the boolean and robustness semantics of signals with respect to formulas. It gives examples of robustness values when launched as the main module. The robustness_semantics function must be completed.
   - timed_signal.py: a library for signals.
4. the `falsification` package includes definitions for optimisation methods and the falsification loop.
   - *falsification.py*: the main file for the falsification part. Contains a complete falsification loop that takes as arguments a model to run, a formula to falsify, an optimisation method to draw samples, and a timeout. It also runs the falsification with different optimisation methods if launched as the main module. The file only contains examples of falsifying models that work with initial positions.
   - *nelder_mead.py*: a simple implementation of the Nelder-Mead algorithm. When launched as the main module, applies Nelder-Mead to find the minimum of two functions.
   - **optimisation.py**: a library that defines some optimisation methods: random, confidence bound, and Nelder-Mead. The Nelder-Mead method has to be completed (or another optimisation method written).
   - point.py: contains an implementation of points that may contain a "mode" (some quantity that should not be added when two points are added, nor scaled when the point is scaled).
5. assignment.pdf: this pdf.
6. requirements.txt: a list of packages for pip to install.
7. utils.py: utility functions.

4 More details

4.1 Robustness semantics

In the file stl/semantics.py around line 77 there is a function stub called `robustness_semantics`. The first task is to complete this function so that it returns the robustness semantics of a given formula on the given signal at the given time (called `start_time` in the argument list).

You may get ideas from the function immediately before, which gives boolean semantics. We have also provided a partial test suite in the file itself, which is run every time it is called as the main module (i.e. via `python -m stl.semantics`).

Though this is not necessary to complete this part, you may find it helpful to review our definitions of Formula objects (which also include Terms) from formula.py and our definition of a Signal from timed_signal.py in order to understand the objects your robustness_semantics will be provided with. Some useful things to know are that Signal objects come with a `.timeRange(...)` helper method for enumerating the control points they use, and also that they have `.upper` and `.lower` attributes to delimit the range they are defined on.

4.2 Optimisation

Optimisation methods as defined in optimisation.py possess three methods.

1. *initialise*, which is called only once when the falsification process starts. It initialises all the parameters and structures needed by this specific optimisation method.
2. *generate*, which generates a new sample when the falsification algorithm needs one.
3. *update*, which updates some values that are specific to this algorithm based on the robustness of the signal, computed by the falsification algorithm.

The falsification loop will call these methods and interact with the optimisation method through them and a global structure called `struct`, which contains information about how falsification has worked until now (the tested input signals, the best robustness found, etc.). In general, to make sure names do not clash, variables that must be stored between two calls to the optimisation method are stored in `struct[''for-optimisation'']`. In the case of the Nelder-Mead method, two fields of `struct` are particularly important:

1. `struct[''struct'']` contains a substructure similar to `struct` itself, used by the algorithm that samples the initial. Whole simplices are stored in `struct[''struct''][''simplex'']`.
2. `struct[''for-optimisation''][''memory'']` is a dictionary that maps the names of variables used in the Nelder-Mead algorithm to their values. For example, if variable i is 1 at some point in the algorithm, then we will have `struct[''for-optimisation''][''memory''][''i'']` $= 1$.

4.3 Reinforcement learning

In the file rl/part3.py, your task is to implement an agent, namely AssignmentSolutionAgent, which will experience an environment designed to help falsify the formula in this benchmark. In this file, you will also see the implementation we did in class for the Agent Foo-bar.

States in the environment are 4-tuples of the form (x, y, v_x, v_y), indicating the starting positions and velocities of the particle. Actions are similarly 4-tuples which indicate how that 4-tuple will be changed, adding/subtracting either 0.05 or 0.02 to/from the current value in one coordinate. You can run your Agent in the target environment by calling `python -m rl.gym`. This will write to the console the best trace your Agent was able to find, as well as the best traces found by some of the other Agents we discussed in class. It should also plot the performance of the agents over time.

You may wish to consult the implementation of the agents we discussed in class, which are available in rl/agent.py. Those Agents sometimes use Gadgets to help keep track of past (state, action, reward) triples. Those Gadgets are available for inspection in rl/gadget.py.

5 Those who fight further

While none of these are required for the assignment, if you are interested in doing more, here is a list of ideas to get you started. Don't hesitate to be creative.

1. In the robustness semantics, equality between modes often "kills" the value of the robustness, either by making it exactly 0 or by making it bigger than it was by an arbitrary factor (why should modes 0 and 1 be "closer" than modes 0 and 2?). Think of a way to circumvent this problem, implement it, and compare with your previous implementation.
2. Implement an optimisation method of your choice and compare with the other ones. Hill climbing might be the easiest one to code, but it only a local maximisation (hence minimisation) method, so it probably does not perform as well. Simulated annealing is probably the easiest global search method to code (much simpler than Nelder-Mead, as it does not need to take care of the "memory" of the algorithm, or any structure).
3. Try using some of the other CPS models and formulas provided with them. Do these falsification methods work equally well with other models?
4. Make a new RL environment based on the SimpleNavEnvironment (from rl/environment.py) where agents can use interval action spaces rather than discretely enumerated actions, then design an agent which is able to take advantage of this feature. (The other agents won't work properly in this environment.) Does it achieve better falsification results?