Reinforcement Learning Exercise 0

Team: Nico Ott 4214197, Lior Fuks 4251285, Hendrik Vloet 4324249

Date: November 9, 2017

Due Date: November 9, 2017

1 Introduction to RL

- **Model**: A model is an agent's internal representation of the environment that predicts its behavior, i.e. forecasts the environment's evolution, regarding an action of the agent.
  A model may help a policy to find the best next action and it consists of 2 components:
  - State-Transition model $\mathcal{P}$. It predicts the next state of the agent after executing an action. Formalized:

$$\mathcal{P}^a_{SS'} = \mathbb{P}\left\{S_{t+1} = s' | S_t = s, A_t = a\right\}$$

  - Reward model $\mathcal{R}^a_s$. It predicts the next expected (immediate) reward for the agent after performing some action, formalized:

$$\mathcal{R}^a_s = \mathbb{E}\{R_{t+1} | S_t = s, A_t = a\}$$

- **Policy**: the policy is the behavior of the agent, i.e. it is a mapping function from state to an action. Or in other words: it is a strategy how the agent will choose its next action, taking into account in which state it is. Nevertheless, a policy does not guarantee optimal performance, since it just represents one possible behavior of the agent out of infinitely many. Briefly, it can be good or bad and thus increase or decrease performance.
  We can divide them into two overall classes:
  - deterministic policy, where the outcome of an evaluation of the state is exactly known:

$$a = \pi(s)$$

  - stochastic policy, where the outcome of an evaluation of the state is not exactly known but can be described with the help of probabilistic calculus:

$$\pi(a|s) = \mathbb{P}(A_t = a | S_t = s)$$

- **Value-function**: predicts the future reward of the agent and is used to evaluate the goodness/badness of states, i.e. the value function can be used to pick actions, depending of the outcome. The value functions "unrolls" future possible rewards under some policy π. Usually, the value function cannot predict the future with absolute certainty. Therefore, expected rewards of the future will have less influence on the value of the actual state("Myopic Evaluation"):

$$v^{(s)}_\pi = \mathbb{E}_\pi\{R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... | S_t = s\}$$

2 GYM

The main objective of this task was, to get familiar with the GYM environment, provided by openAI.
We decided to try different approaches in order to increase familiarity with GYM. As a result of these approaches we came up with two basic ideas:

1. A simple demonstration of how the cartpole will behave under a **random** policy.
2. A more detailed analysis of two different policies:
   - Random Search
   - Hill Climbing

2.1 Setup

- filename: main.py
- ONLY WORKS WITH PYTHON2
- Depending on the configurations in the user interface, the execution can take a while. A configuration with 1000 RUNS and 1000 TESTRUNS and the POLICY_ANALYSIS,SHOW_PARAMETER, SHOW_PLOTS, TEST_PARAM flags set to true took approx. 10-15 minutes (results are shown in figure 2).

2.2 User Interface

The program is completely controllable by a few parameters and flags. The default values have been set to show a short demonstration but we encourage you to play with the
POLICY_ANALYSIS flag! The other parameters have following functions (refer also to figure 1):

- POLICY_ANALYSIS
  - True: enable the full analysis program, i.e. performing a random search and hill climbing (Warning: Increases runtime)
  - False: disable full analysis and just show a short demo with a randomized policy within the cartpole environment
- RUNS: set the total amount of runs that are used in the different parameter searches. Increasing it beyond the default(100) may increase runtime drastically
- RENDER: if set true, all episodes of any cartpole interaction is rendered and thus displayed. (Warning: Can increase runtime drastically!)
- SHOW_PARAMETERS: if true, the program will display several additional information about used weights, rewards, observations etc. (Warning: activating it, may result in a blown up output display)
- NOISE_SCALING: affects the Hill Climbing search, which uses noise injection in order to escape local minima. The higher it is, the more the Hill Climbing behaves like pure Random Search
- TESTRUNS: sets the amount of testruns, hat will be used to find the best parameter pairs, that have been found by Random Search and Hill Climbing
- SHOW_PLOTS: if set true, several plots will show up in the end, regarding some statistics of the Random Search and the Hill Climbing (Histogram, ECDF, Success-Rate, see figure 2)
- TEST_PARAM: in case POLICY_ANALYSIS and TESTRUNS is set to true, the program will try to find the best parameter pairs, which have been found by Random Search and Hill Climbing

```
#### STATICS ############################################################################


### all values are only for a short demonstration, you can play with them as much as
### you want, but it can increase runtime drastically

### set this to true, to run the full random search and hill climbing analysis
### if it is false, a simpler version of a random policy will execute
POLICY_ANALYSIS = False



# choose amount of total runs for the parameter search
RUNS = 100
# set true for enabling rendered cartpole, default= False, WARNING: INCREASES RUNTIME!
RENDER = False
# set true to show successful parameters that are found in one successfull run
# default = False
SHOW_PARAMETERS = False

# set noise scaling for noise injection in hill climbing
# a high noise scaling results in similar performance as random search!
NOISE_SCALING = 0.1

# set amount of test runs for validating found averaged parameters
TESTRUNS = 100

# show plots
SHOW_PLOTS = False

# look for the most successful parameters in random search and hill climbing
TEST_PARAM = True
```

Figure 1: Screenshot of the user configuration part of the program

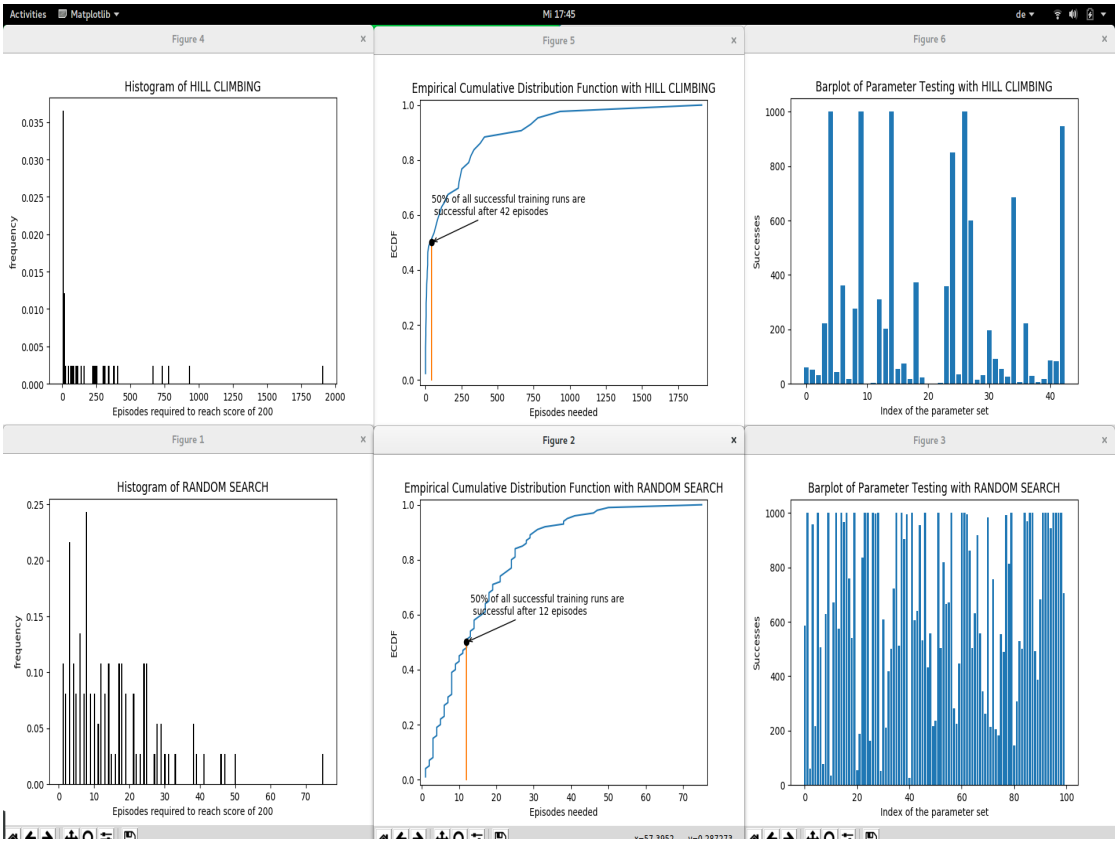

Figure 2: Plotting results of a detailed policy analysis.