Exercise session 4: Policy gradient

Guillaume Charpiat, Victor Berger

January 25, 2018

Preamble

This exercise session is a followup of the previous one.

Exercise

In this project, you are asked to solve a variation of the mountain car problem, using a policy gradient algorithm.

We are considering an environment similar to the one of last week. Once again, this is a mountain car problem, however we have changed some parts of the problem:

- The possible actions for your car are now continuous: any force value in $[-F_{max}; F_{max}]$ (if you output values outside of this range they will be clamped).
- You receive at each time step a reward of $-0.1 - \lambda |F_t|^2$, where F_t is the force you last applied.
- λ and F_{max} are parameters of the environment that you do not know in advance (and will be randomized on the test bed).
- You receive a reward of 100 for reaching the top of the hill.

This means your agent must find an appropriate balance between getting out of the valley as quickly as possible and not accelerating too much (to save fuel).

Given the action space is now continuous, using an approximate Q-learning algorithm like last time becomes difficult. A possibility would be to discretize the force values, but this does not combine well with the fact that F_{max} and λ are unknown. Instead, we will use a policy gradient method, which can easily model continuous action spaces.

We suggest the following model:

- The policy $\pi_{\theta_t}(a_t|s_t)$ is always a normal distribution, of mean μ_t and standard deviation σ_t;
- μ_t is a linear function of pre-defined descriptors of the state:
  $\mu_t = \mu_{\theta_t}(s_t) = \sum_{i,j} \theta^t_{i,j}\phi_{i,j}(s_t)$,
  where $\phi_{i,j}$ is the same kernel basis as for previous exercise;
- σ_t is annealed: it starts with a rather high value (to encourage exploration) and decreases over time as the model μ_{θ_t} gets better.

(The following questions are here to guide you though the implementation, and do not require a written answer.)

Question 1: What is the expression of $\log \pi_{\theta_t}(a_t|s_t)$? Of $\nabla_\theta \log \pi_\theta(a_t|s_t)$?

We consider implementing the actor-critic policy gradient algorithm to solve this problem. For this, we also need an estimation of the value function $V(s_t)$. We will learn it using a linear parametric model as well:

$$V_{\psi_t}(s_t) = \sum_{i,j} \psi^t_{i,j}\phi_{i,j}(s_t)$$

Question 2: What is the update rule for the parameters $\psi_{i,j}$ of V_ψ ?

Question 3: What is the update rule for the parameters $\theta_{i,j}$ of π_θ ?

Question 4: How does the value of σ impact the update of $\theta_{i,j}$? Design an annealing rule for σ appropriately.

Question 5: Seeing the update rule for $\theta_{i,j}$, would it make sense to do an ϵ-greedy-like version of this algorithm (acting completely randomly with probability ϵ)?

Question 6: Then, how can we go into "exploitation mode" with this policy? What else should we take care of when doing so?

Implement the actor-critic policy gradient for this problem and submit your solution on the platform.

Template description

The template is a zip file that you can download on the course website. It contains several files, two of them are of interest for you: `agent.py` and `main.py`.

`agent.py` is the file in which you will write the code of your agent, using the `RandomAgent` class as a template. Don't forget to read the documentation

it contains. In particular, note that for this exercise, at the beginning of a new game, the `reset` function called returns to the agent information about the range of possible x coordinates. As usual you can have the code of your several agents in the same file, and use the final line `Agent = MyAgent` to choose which agent you want to run.

`main.py` is the program that will actually run your agent. You can run it with the command `python main.py`. It also accepts a few command-line arguments:

- `--ngames N` will run your agent for N games against in the same environment and report the total cumulative reward
- `--niter N` maximum number of iterations allowed for each game
- `--batch B` will run B instances of your agent in parallel, each against its own bandit, and report the average total cumulative reward
- `--verbose` will print details at each step of what your agent did. This can be helpful to understand if something is going wrong.
- `--interactive` will train your agent `ngames` times, then run an interactive game displaying informational plots. You need to have `matplotlib` installed to use it.

The running of your agent follows a general procedure that will be shared for all later practicals:

- The environment generates an observation
- This observation is provided to your agent via the `act` method which chooses an action
- The environment processes your action to generate a reward
- this reward is given to your agent in the `reward` method, in which your agent will learn from the reward

This 4-step process is then repeated several times.

Grading

The final performance of your agent will be evaluated by running the following command on a pseudo-random[1] testbed:

```
python main.py --ngames 200 --niter 400 --batch 10
```

Once you think your implementation is good, create a zip file containing your `agent.py` file and the `metadata` file provided in the template, and upload it to the platform. Your score will be computed and you can compare yourself to the rest of the class using the leaderboard. Your grade for this exercise will be based on this score.

[1] This means that uploading two times the exact same code will generate the exact same score