Exercise session 3: Approximate Q-Learning

Guillaume Charpiat, Victor Berger

January 25, 2018

Preamble

This exercise session is adapted from a project of the EPFL *Unsupervised and Reinforcement Learning in neural networks* course.

Exercise

In this project, you are asked to solve the mountain car problem, using the approximate Q-Learning algorithm and its TD(λ) generalization. The mountain car problem is that of a car stuck amidst two mountains. A left or right constant thrust can be applied to the car. The goal is to make the car climb the mountain to its right. The thrust that can be applied is not sufficient for the car to directly go uphill. Momentum has to be gathered to climb high enough.

You are provided with an implementation of the mountain car problem. The state space of the agent consists of the x coordinate of the car, and its horizontal velocity v_x. x lies in $[-d_{min}; 0]$ (this range is provided to your agent at the beginning of a game) and v_x lies in $[-20; 20]$. The class of approximating functions you are strongly advised to use proceeds as follow. It maps points (x, v_x) into a high dimensional representation and consider the class of linear functions in this high dimensional space. When the high dimensional space considered is well chosen, this considerably increases expressiveness when compared to directly considering linear functions on the state space. Consider $s^{0,0}...s^{p,k}$ nodes of a discretization of the state space. Typically, dividing both axes evenly into respectively p and k intervals gives $(p+1) \times (k+1)$ points $s^{i,j}$ with coordinates

$$s^{i,j} = (-d_{min} + i\frac{d_{min}}{p}, -20 + j\frac{40}{k}) \tag{1}$$

Each coordinate of the $(p+1) \times (k+1)$ dimensional representation vector ϕ only depends on the closeness of (x, v_x) and $s^{i,j}$. Typically, $\phi_{i,j}$ is 1 if $(x, v_x) = s^{i,j}$ and tends to zero the farther (x, v_x) is to $s^{i,j}$. Formally

$$\phi_{i,j} = e^{-(x-s_1^{i,j})^2} e^{-(v_x-s_2^{i,j})^2} \tag{2}$$

The Q-function approximation then takes the form

$$Q((x, v_x), a) = \sum_{i,j} W^a_{i,j} \phi_{i,j} \tag{3}$$

where the $W^a_{i,j}$ are real numbers.You are asked to design a Q-Learning and a TD(λ) algorithms that learn parameters $W^a_{i,j}$ to solve the mountain car task. (p, k) are to be tunable hyperparameters of your algorithm, along with any other hyperparameter of interest. The provided code gives you access to facilities to visualize your learning progresses.

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 1000 --niter 100 --batch 200
```

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