Tópicos especiais em Inteligência Computacional II:
Aprendizado por Reforço
Exercício 4: Actor-Critic

Wouter Caarls
wouter@ele.puc-rio.br

May 15, 2022

In all previous exercises, the action space was discrete (North, South, East, West; or -3V, 0V, 3V). However, for many applications in robotics this leads to high-frequency chattering. To deal with continuous action spaces, the policy has to be represented explicitly, instead of being derived from the value function. One class of methods that do this are the *Actor-Critic* methods. In this exercise, you will implement two of these and compare them.

The prerequisites are the same as for the previous exercise, so should already be installed. If not, open the Anaconda Prompt and run

```
(base) C:\Users\You> conda install swig
(base) C:\Users\You> pip install tensorflow gym[box2d]
```

1 Understanding the code

The Python code for this exercise (`ex4.py`) contains one function and three classes, see Table 1.

Table 1: Main functions and classes

`ex4.rbfprojector`	Gaussian Radial Basis Function projector.
`ex4.DDPG`	Deep Deterministic Policy Gradient network.
`ex4.Memory`	Replay memory.
`ex4.Pendulum`	Pendulum swing-up environment.

Exercise 1.1 Whereas to evaluate the DQN network, a state and action were required, the DDPG network's `critic` function allows us to evaluate a state by itself. What is the return value in that case?

2 Linear state-value actor-critic

In linear approximations, a function $f(x)$ is approximated as $\hat{f}(x;\theta) = \theta^T\phi(x)$[1], where θ is the parameter vector and ϕ is a function that projects the state x onto a feature vector. In this exercise, ϕ projects onto Gaussian radial basis functions, and is implemented in `rbfprojector`.

Exercise 2.1 Create an RBF projector `feature` that projects onto 5 basis functions per dimension, with width 0.1. Create a feature vector `theta` of all ones with the same size as the parameter vector (use `len(feature([0, 0, 0]))` and plot the resulting approximation using `env.plotlinear(theta, feature)`, where `env` is a `Pendulum` object. It should look something like Figure 1.

Exercise 2.2 Count the number of basis functions in the figure and compare it to the feature vector size. Explain the result. How does this relate to the sine-cosine representation used by the Pendulum environment?

Exercise 2.3 Alter the main simulation loop from Exercise 3 to solve the Pendulum problem using linear state-value actor-critic with Gaussian radial basis functions. Use 21 functions per dimension with width 0.25. Start with an initial exploration rate of $\sigma = 1$ and decay it by 0.99 every episode. Set the critic learning rate $\alpha = 0.2$, actor learning rate $\beta = 0.01$, and discount rate $\gamma = 0.99$.

After training, plot the learning curve and approximators. For the latter, use `env.plotlinear(w, theta, feature)`. The results should be similar to Figure 2.

Exercise 2.4 Measure the performance in three ways:

[1] in Python: `np.dot(theta, feature(x))`.

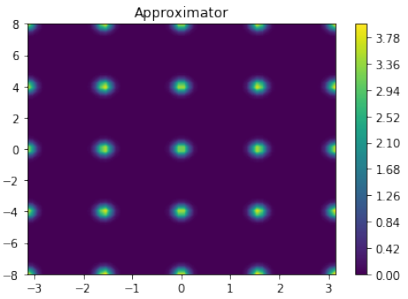

Figure 1: Gaussian radial basis functions

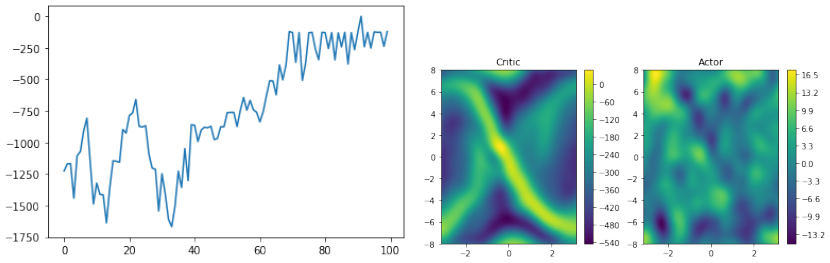

Figure 2: Results of training linear state-value actor-critic on the Pendulum problem.

- *rise time*: how many episodes does it take for the learning to converge? Use the point at which the episode reward is consistently above -500.
- *end performance*: the reward obtained after training. Use a separate run of 100 episodes where exploration and learning is disabled, and average the episode rewards.
- *computation time*: how long it took to train 100 episodes. Use `time.time()`.

Run the training a few times and average the results to get statistically meaninful information.

3 Deep Deterministic Policy Gradient

Now we move to a nonlinear representation, with the required stabilization mechanisms (replay memory, target network). The Deep Deterministic Policy Gradient algorithm estimates the value function (critic) in the same way as Deep Q learning you implemented in Exercise 3, but additionally trains a policy (actor) using the gradient

$$\nabla_\theta J_\mathrm{r}(\theta) = E_{\pi_\theta}\left\{\nabla_\theta Q^{\pi_\theta}(s, \pi(s;\theta))\right\}, \tag{1}$$

that is, moving the policy parameters such that they maximize the Q function.

Exercise 3.1 Implement DDPG. Use `DDPG.critic` to read the value function and `DDPG.actor` to read the policy. Note that the `DDPG.train` function already handles the training of the actor using the gradient in Equation 1. Update the target network every 200 steps.

Note that the actor network's output is limited to $[-1, 1]$, while the pendulum's actions are in the range $[-2, 2]$. To make sure the results are comparable, multiply the actor's value by 2 before passing it to `env.step`, and halve the noise.

After training, plot the learning curve and approximators. For the latter, use `env.plotnetwork(ddpg)`. The results should be similar to Figure 3.

Exercise 3.2 Measure the same statistics as for linear state-value actor-critic, and compare the results. Is it what you expected? Explain.

Upload your report in pre-executed ipynb format to Moodle.

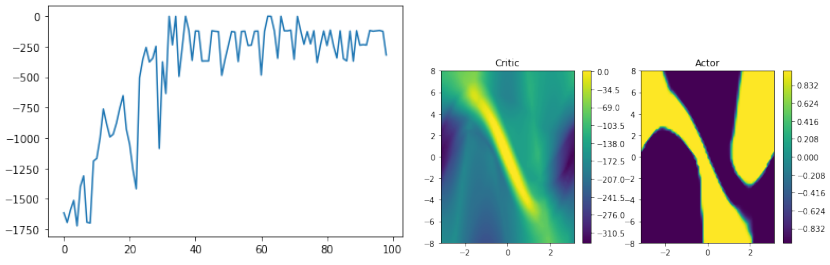

Figure 3: Results of training DDPG on the Pendulum problem.