

COMP3667: Reinforcement learning practical 5

Exploration vs exploitation with Monte Carlo methods

christopher.g.willcocks@durham.ac.uk & robert.lieck@durham.ac.uk

1 Overview

Welcome to the fifth reinforcement learning practical. In this practical, we will be focusing on the fundamental exploration vs exploitation problem with the aim to build intuition and develop good practices in conducting experiments. In particular, we will:

- Evaluate different ϵ -greedy strategies for Monte Carlo reinforcement learning.
- Evaluate different GLIE strategies.
- A mini competition—who can design the fastest converging and most sample efficient agent, just by tuning the ϵ -schedule?

Setup

For this practical, you will need a basic Python environment with `numpy`, `matplotlib`, and OpenAI `gym` (version 0.20.0). You will not need a GPU.

```
1 !pip install 'gym[box2d]==0.20.0'
2 import gym
3 import time
4 import numpy as np
5 import matplotlib.pyplot as plt
6 import matplotlib.font_manager
7 from IPython import display as disp
8 %matplotlib inline
```

As before, we will also use the frozen lake gym environment:

```
1 # actions
2 LEFT, DOWN, RIGHT, UP = 0,1,2,3
3
4 # import the frozen lake gym environment
5 name = 'FrozenLake-v1'
6 env = gym.make(name, is_slippery=False) # warning: setting slippery=True results in very complex enviro
7 env.seed(742)
8 env.action_space.seed(742)
```

Monte Carlo methods with ϵ -greedy exploration

When tuning RL agents, it's important to evaluate convergence across multiple experiments. You can use tools like 'weights & biases' for this. For now, we'll simply keep track of our past experiments in a list and plot them.

```
1 # different colors for each experiment
2 cmap = get_cmap('tab10')
3 colors = cmap.colors
4
5 # plotting
6 ep_reward = 0
7 reward_list = [[]]
```

```

8 plot_data = [[]]
9 plot_labels = []
10 experiment_id = 0

```

Evaluate the following cell (ϵ -greedy MC) multiple times to plot the results in different colours: [copy code](#)

```

1 Q = np.zeros([env.observation_space.n, env.action_space.n])
2 n_s_a = np.zeros([env.observation_space.n, env.action_space.n])
3 num_episodes = 15000
4 epsilon = np.random.rand()*0.2+0.35 # change this each run
5 plot_labels.append("eps: {:.3f}".format(epsilon))
6
7 for episode in range(num_episodes):
8     state = env.reset()
9     done = False
10    results_list = []
11    result_sum = 0.0
12    while not done:
13        if np.random.rand() > epsilon:
14            action = np.argmax(Q[state, :])
15        else:
16            action = env.action_space.sample()
17        new_state, reward, done, _ = env.step(action)
18        results_list.append((state, action))
19        result_sum += reward
20        state = new_state
21
22    for (state, action) in results_list:
23        n_s_a[state, action] += 1.0
24        alpha = 1.0 / n_s_a[state, action]
25        Q[state, action] += alpha * (result_sum - Q[state, action])
26
27    reward_list[experiment_id].append(result_sum)
28    aoc[experiment_id] += int(result_sum)
29
30    if episode % 1000 == 0:
31        plot_data[experiment_id].append([episode, np.array(reward_list[experiment_id]).mean(),
32                                         np.array(reward_list[experiment_id]).std()])
33        reward_list[experiment_id] = []
34
35        for i in range(len(plot_data)):
36            plt.plot([x[0] for x in plot_data[i]], [x[1] for x in plot_data[i]],
37                    '-', color=colors[i], label=plot_labels[i] + ', AOC: '+str(aoc[i]))
38            plt.xlabel('Episode number')
39            plt.ylabel('Episode reward')
40            plt.legend()
41            disp.clear_output(wait=True)
42            plt.show()
43
44
45    env.close()
46
47    # advance to next experiment (so when we run the cell again we retain previous results)
48    experiment_id += 1
49    reward_list.append([])
50    plot_data.append([])
51    aoc.append(0)

```

Take your time to study the code and relate it to what was covered in the lecture. If you run the cell multiple times, you will see it keeps track of the previous results and plots them in different colours.

Questions

- Evaluate the above cell several times for different ϵ values [0.6, 0.5, 0.4, 0.3, 0.2] (it should display previous experiment results on the same graph in different colours):
- What behaviour do you observe with small ϵ vs with large ϵ agents?
- Does changing ϵ impact the expected episode length (how long it takes to run)?
- What is the AOC (area under the curve) measuring here?

Now change the environment to the more difficult 8x8 frozen lake environment:

```
1 name = 'FrozenLake8x8-v1'
2 env = gym.make(name, is_slippery=False)
3 env.seed(742)
4 env.action_space.seed(742)
```

Question

- Experiment with ϵ values in this new environment. What do you observe in contrast to the previous environment?

2 Evaluating GLIE strategies

A learning policy is considered GLIE (Greedy at the Limit with Infinite Exploration) if these two conditions are met:

1. The agent continues to explore everything:

$$\lim_{k \rightarrow \infty} N_k(s, a) = \infty,$$

where $N_k(s, a)$ is the number of times an action a was taken in state s at step k .

2. The policy converges on a greedy policy:

$$\lim_{k \rightarrow \infty} \pi_k(a|s) = \mathbf{1}(a = \arg \max_{a' \in \mathcal{A}} Q_k(s, a')).$$

In other words, after training for ever and ever, the final policy will be greedy with respect to Q .

Exercise

- Return to the previous simple environment and design a GLIE policy that achieves an AOC $> 12,000$. It helps to design these in desmos.com—make sure to have the y-axis maximum at 1, and the x-axis maximum at 15,000 for this environment.

Competition

Now return to the more difficult 8x8 frozen lake environment and try to optimise the AOC.

- Compete with those around you (not allowed more than 15,000 episodes). When you are satisfied with your best schedule, evaluate it 5 times and write down the mean AOC across those 5 trials. Is it stable or does it only sometimes work?
- Now, let's try the 'hardcore' version of this environment. Enable `is_slippery=True` and try to design a schedule that performs well over 5 trials of this slippery 8x8 environment. Again, write down the mean AOC across those 5 trials. Is it stable or does it only sometimes work? Why?
- Who can design the best schedule for the deterministic and the slippery environments respectively? What are the challenges here?

3 Optimism in the face of uncertainty

A guiding principle to address the exploration-exploitation dilemma is “optimism in the face of uncertainty”. The basic idea is that if you are very certain, you can safely exploit, but if you are uncertain about the value of an action you should act “optimistically”, so basically “assume rewards everywhere” (unless you know better), which leads to exploration. This principle can be mathematically formulated in different ways and it is even possible to prove certain guarantees about convergence rates (all good policies of this kind are also GLIE, of course). One of the most simple and popular policies is called UCB1¹ (upper confidence bounds) and it defines an optimistic (upper confidence) bound on the estimated action value (average return) as

$$B_{(s,a)} = \underbrace{\widehat{R}_{(s,a)}}_{\text{unbiased value estimate}} + \underbrace{2 C_p \sqrt{\frac{2 \log n_s}{n_{(s,a)}}}}_{\text{bonus for little explored actions}}, \quad (1)$$

where $\widehat{R}_{(s,a)}$ is the average return of action a in state s ; n_s is the number of visits to state s ; $n_{(s,a)}$ is the number of times action a was taken in state s ; the returns are assumed to be in $[0, 1]$; and the constant $C_p > 0$ controls exploration. Actions are then chosen by maximising this optimistic upper bound instead of the unbiased value estimate

$$a^* = \arg \max_a B_{(s,a)}. \quad (2)$$

You can try out this policy by making some simple modifications to your code

[copy code](#)

```
1  # define exploration constant and plot label
2  c = .1
3  plot_labels.append(f"UCB1: c={c}")
4  ...
5
6  for episode in range(num_episodes+1):
7      ...
8      while not done:
9          # define UCB1 policy
10         if np.any(n_s_a[state] == 0):
11             # first, try every action at least once
12             action = np.random.choice(np.argwhere(n_s_a[state] == 0).flatten())
13         else:
14             # then, use upper bounds to choose actions
15             R_s_a = Q[state, :]
16             n_s_a_ = n_s_a[state]
17             n_s = n_s_a_.sum()
18             B_s_a = R_s_a + 2 * c * np.sqrt(2 * np.log(n_s) / n_s_a_)
19             action = np.random.choice(np.argwhere(B_s_a==np.max(B_s_a)).flatten())
20     ...
```

Exercise

Play around with the constant C_p and see how the UCB1 policy performs in comparison to the static ϵ -greedy policies and your best scheduled ϵ -greedy policy. What differences can you observe?

¹Auer P, Cesa-Bianchi N, Fischer P (2002) Finite-time Analysis of the Multiarmed Bandit Problem. Machine Learning 47:235–256. <https://doi.org/10.1023/A:1013689704352>; Kocsis L, Szepesvári C (2006) Bandit based monte-carlo planning. In: Machine Learning: ECML 2006. Springer, pp 282–293