COMP3667: Reinforcement learning practical 5

Exploration vs exploitation with Monte Carlo methods

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.
- Learn about the heuristic: "optimism in the face of uncertainty".
- 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 only need the following packages:

```
import numpy as np
import rldurham as rld

import matplotlib.pyplot as plt
from IPython import display as disp
```

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

```
# import the frozen lake gym environment
env = rld.make('FrozenLake-v1', is_slippery=False)
rld.seed_everything(42, env)
```

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.

```
# plotting
ep_reward = 0
reward_list = [[]]
plot_data = [[]]
plot_labels = []
auc = [0]
experiment_id = 0
```

You can evaluate the following (ϵ-greedy MC) multiple times to plot in different colours:

```
Q = np.zeros([env.observation_space.n, env.action_space.n])
n_s_a = np.zeros([env.observation_space.n, env.action_space.n])
num_episodes = 15000
epsilon = np.random.rand()*0.2+0.35 # change this each run
```

```
plot_labels.append("eps: {:.3f}".format(epsilon))

for episode in range(num_episodes):
    state, info = env.reset()
    done = False
    results_list = []
    result_sum = 0.0
    while not done:
        if np.random.rand() > epsilon:
            action = np.argmax(Q[state, :])
        else:
            action = env.action_space.sample()
        new_state, reward, term, trunc, info = env.step(action)
        done = term or trunc
        results_list.append((state, action))
        result_sum += reward
        state = new_state

    for (state, action) in results_list:
        n_s_a[state, action] += 1.0
        alpha = 1.0 / n_s_a[state, action]
        Q[state, action] += alpha * (result_sum - Q[state, action])

    reward_list[experiment_id].append(result_sum)
    auc[experiment_id] += int(result_sum)

    if episode % 1000 == 0:
        plot_data[experiment_id].append([episode, np.array(reward_list[experiment_id]).mean(),
            np.array(reward_list[experiment_id]).std()])
        reward_list[experiment_id] = []

        for i in range(len(plot_data)):
            plt.plot([x[0] for x in plot_data[i]], [x[1] for x in plot_data[i]],
                     '-',
                     # color=colors[i],
                     label=plot_labels[i] + ', AUC: '+str(auc[i]))
        plt.xlabel('Episode number')
        plt.ylabel('Episode reward')
        plt.legend()
        disp.clear_output(wait=True)
        plt.show()


env.close()

# advance to next experiment (so when we run the cell again we retain previous results)
experiment_id += 1
reward_list.append([])
plot_data.append([])
auc.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 AUC (area under the curve) measuring here?

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

```python
env = rld.make('FrozenLake8x8-v1', is_slippery=False)
rld.seed_everything(42, env)
```

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:

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

   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:

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

   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 AUC > 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 AUC.

- 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 AUC across those 5 trials. Is it stable or does it only sometimes work?
- Now, lets 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 AUC 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 UCB1[1] (upper confidence bounds) and it defines an optimistic (upper confidence) bound on the estimated action value (average return) as

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

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

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

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

```python
...
# define exploration constant and plot label
c = .1
plot_labels.append(f"UCB1: c={c}")
...

for episode in range(num_episodes+1):
    ...
    while not done:
        if np.any(n_s_a[state] == 0):
            # first, try every action at least once
            action = np.random.choice(np.argwhere(n_s_a[state] == 0).flatten())
        else:
            # then, use upper bounds to choose actions
            R_s_a = Q[state, :]
            n_s_a_ = n_s_a[state]
            n_s = n_s_a_.sum()
            B_s_a = R_s_a + 2 * c * np.sqrt(2 * np.log(n_s) / n_s_a_)
            action = np.random.choice(np.argwhere(B_s_a==np.max(B_s_a)).flatten())
    ...
```

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?

---

[1] Auer 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