EXERCISE 8
Exploration and Bandits

Tue Herlau
tuhe@dtu.dk

21 March, 2025

Objective: Today's exercises will deal with a core issue in reinforcement learning, known as the exploration-exploitation dilemma. To introduce the topic, we start by considering a slightly modified (nonstationary) version of the standard (stationary) k-armed bandit problem. You'll implement your own agents and observe how different approaches to balancing exploration and exploitation perform in this setting. (44 lines of code)
Exercise code: `https://lab.compute.dtu.dk/02465material/02465students.git`
Online documentation: 02465material.pages.compute.dtu.dk/02465public/exercises/ex08.html

Contents

1 Theoretical question: Simple bandits

☞

(a.) Solve [SB18, Problem 2.2]

(b.) The average regret associated with an arm a is defined as:

$$\Delta_a = \max_{a'} \mathbb{E}[R_t | A_t = a'] - \mathbb{E}[R_t | A_t = a]$$

What is the average regret for arms $a = 1, 2, 3$ in the 10-armed testbed depicted in [SB18, Figure 2.1]?

(c.) Why do you think it is called the "average regret"?

2 Getting started (`bandits.py`)

My experience is that Bandits can feel a bit disconnected from the rest of [SB18]. This is really a shame, since bandits are the primary mechanism by which the agent explores. To enforce this connection, we will therefore implement Bandits using the same `Agent` / `Environment` -distinction we are have already seen. Specifically:

- The bandit is the *environment* (i.e. the *slot machine*) where your actions corresponding to pulling a particular arm (labeled from 0 to $k-1$) and get a reward. There are two specific things to keep in mind:
  - The reset-function has to completely reset the environment, i.e. it will randomize which arm is the optimal arm.
  - For plotting, we want to track the average regret, which is defined as the expected difference in reward between the optimal arm and the currently selected arm. I.e., if the regret is zero, we have selected an optimal arm. Since only the bandit-environment knows which arm is optimal, we have to compute it in the environment.

To simplify this, our bandit-environment will all inherit from the same `BanditEnvironment` (defined below) and simply implement the `reset` and `bandit_step` methods:

```python
# bandits.py
class BanditEnvironment(Env):
    def __init__(self, k : int):
        super().__init__()
        self.observation_space = Discrete(1)  # Dummy observation space with a single observation.
        self.action_space = Discrete(k)       # The arms labelled 0,1,...,k-1.
        self.k = k  # Number of arms


    def reset(self):
        raise NotImplementedError("Implement the reset method")

    def bandit_step(self, a):
        reward = 0 # Compute the reward associated with arm a
        regret = 0 # Compute the regret, by comparing to the optimal arms reward.
        return reward, regret

    def step(self, action):
        reward, average_regret = self.bandit_step(action)
        info = {'average_regret': average_regret}
        return None, reward, False, False, info
```

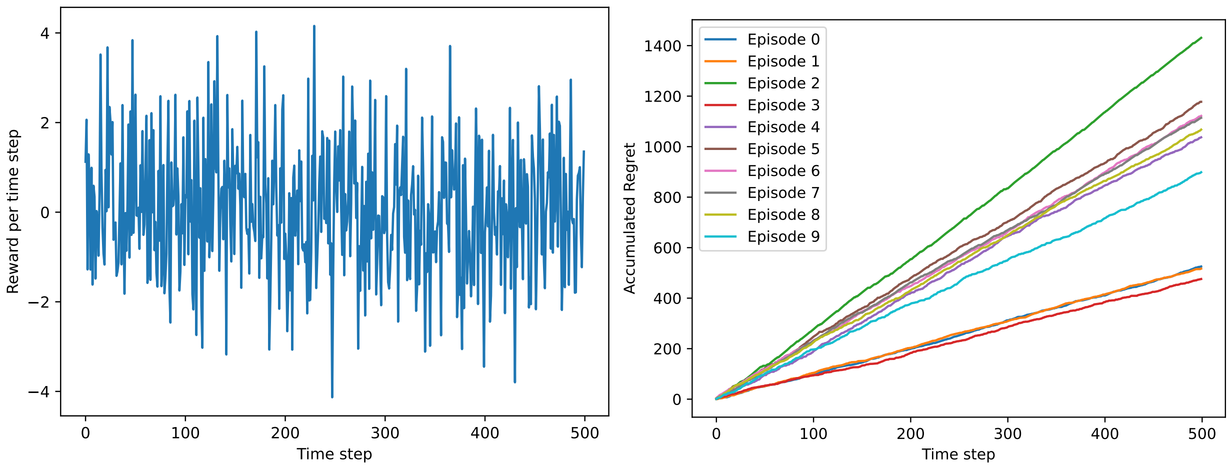


Figure 1: Simple bandit examples

- The agent will just be our regular `Agent` class, where the policy function (`def pi(s,t):`) should ignore `s` and return the arm to pull:

$$\pi : \emptyset \mapsto \{0, 1, \ldots, k-1\}.$$

2.1 Running agents and the testbed

Since a bandit is just a special kind of agent we can train it using the methods we have already seen. As a first example, we will instantiate a bandit environment and plot the reward obtained over 500 time steps:

```python
# bandit_example.py
from irlc import Agent, train, savepdf
from irlc.ex08.bandits import StationaryBandit
bandit = StationaryBandit(k=10) # A 10-armed bandit
agent = Agent(bandit)  # Recall the agent takes random actions
_, trajectories = train(bandit, agent, return_trajectory=True, num_episodes=1, max_steps=500)
plt.plot(trajectories[0].reward)
plt.xlabel("Time step")
plt.ylabel("Reward per time step")
```

This code should be familiar from the previous weeks, and the result can be found in fig. 1 (left). This plot is not very informative at all since the reward is just random. To get closer to the plots in [SB18], we need to compute the regret, and average the reward over multiple runs where the optimal arm is reset. A more elaborate example show 10 runs and plot the accumulated regret (see fig. 1 (right)):

```python
# bandit_example.py
agent = Agent(bandit)  # Recall the agent takes random actions
for i in range(10):
    _, trajectories = train(bandit, agent, return_trajectory=True, num_episodes=1, max_steps=500)
    regret = np.asarray([r['average_regret'] for r in trajectories[0].env_info[1:]])
    cum_regret = np.cumsum(regret)
    plt.plot(cum_regret, label=f"Episode {i}")
```

```python
plt.legend()
plt.xlabel("Time step")
plt.ylabel("Accumulated Regret")
```

In our real experiments, we will be using the so-called 10-armed-testbed as described in [SB18]. The 10-armed testbed is more or less what you have already seen, but with a few niceties:

- The 10 regret-lines should be averaged into a single line
- It allows easy plotting of multiple bandit-algorithms in one plot
- It saves results as they are computed so plots can happen quickly

Caching results is very helpful in machine learning, but it can a pain when you develop your methods. You can disable cache using `use_cache = False` (see exercise scripts) and delete all cached results by removing the `ex08/cache` directory. The following example illustrate what it looks like in practice:

```python
# simple_agents.py
env = StationaryBandit(k=10)
agents = [BasicAgent(env, epsilon=.1), BasicAgent(env, epsilon=.01), BasicAgent(env, epsilon=0) ]
eval_and_plot(env, agents, num_episodes=100, steps=1000, max_episodes=150, use_cache=use_cache)
savepdf("bandit_epsilon")
plt.show()
```

This code set up three different agents (in this case we vary the parameter `epsilon`), test them for 100 episodes, and plot the average performance. The parameter `max_episodes` control the maximum number of episodes to compute, i.e. if you run the script again it will only compute an additional 50 episodes and show results of all 150 runs, and if you run it a third time it will compute no additional episodes but show all 150 results. The result is shown in problem 1.

Warning: Remember to reset your agent Your bandit agent is supposed to learn the best arm in the current environment. Since the environment is randomized after each episode, the bandit algorithm too has to reset itself. The way this is handled is in the policy-function: Whenever the agent is in time-step 0, it should reset itself and train anew:

```python
# simple_agents.py
def pi(self, s, t, info=None):
    """ Since this is a bandit, s=None and can be ignored, while t refers to the
    ↪ time step in the current episode """
    if t == 0:
        # At step 0 of episode. Re-initialize data structure.
        self.Q = np.zeros((self.k,))
        self.N = np.zeros((self.k,))
    # compute action here
```

3 The ε-greedy agent (`simple_agents.py`)

Our first objective will be to re-produce the results in [SB18, Section 2.3] for the epsilon-greedy agent using the 10-armed-testbed with the **simple bandit agent** described in the box in [SB18, Section 2.4]. The testbed environment will be implemented as the class `StationaryBandit` in `bandits.py`.

Problem 1 *The first bandits*

Implement the basic, stationary environment and the epsilon-greedy basic agent to test an epsilon-greedy bandit strategy. When done, you should get a plot showing the influence of (ε-greedy exploration)

Info: When done, you should get the same output as in [SB18]

4 Conceptual question: UCB Bandits

Consider a bandit problem with exactly two arms $A_t = a \in \{0, 1\}$.

- The first arm, $A_t = 0$, always gives a reward of $R_t = 5$
- The second arm, $A_t = 1$, always gives a reward of $R_t = 10$.

In this problem, we imagine we apply UCB1 to this agent and we are interested in the behavior of the upper-confidence bound

$$f_a(t) = Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \tag{1}$$

for the two arms (recall the upper-confidence bound is used to select the next action). You can assume that $c = 2$.

(a.) Suppose we try each of the arms $N = 500$ times. What are the two upper-confidence bounds at time $t = 1000$?

(b.) After these 1000 pulls, which of the two arms will UCB1 select? Explain your result both as a consequence of how UCB1 select actions and intuitively.

(c.) In the next question, you should assume the agent begins to always select arm $A_t = 1$ in each successive timestep t. What is the upper-confidence bound $f_a(t)$ as a function of time for arms $a = 0, 1$?

(d.) Make a rough sketch (i.e., without using maple or python) of how the upper-confidence bounds $f_0(t)$ and $f_1(t)$ change as a function of time. The plot should start after the initial $N = 500$ attempts at each arm and you should try to get the overall shape of the curves correct based on your derivation.

(e.) Will the two curves eventually cross? Why/why not? What does crossing signify?

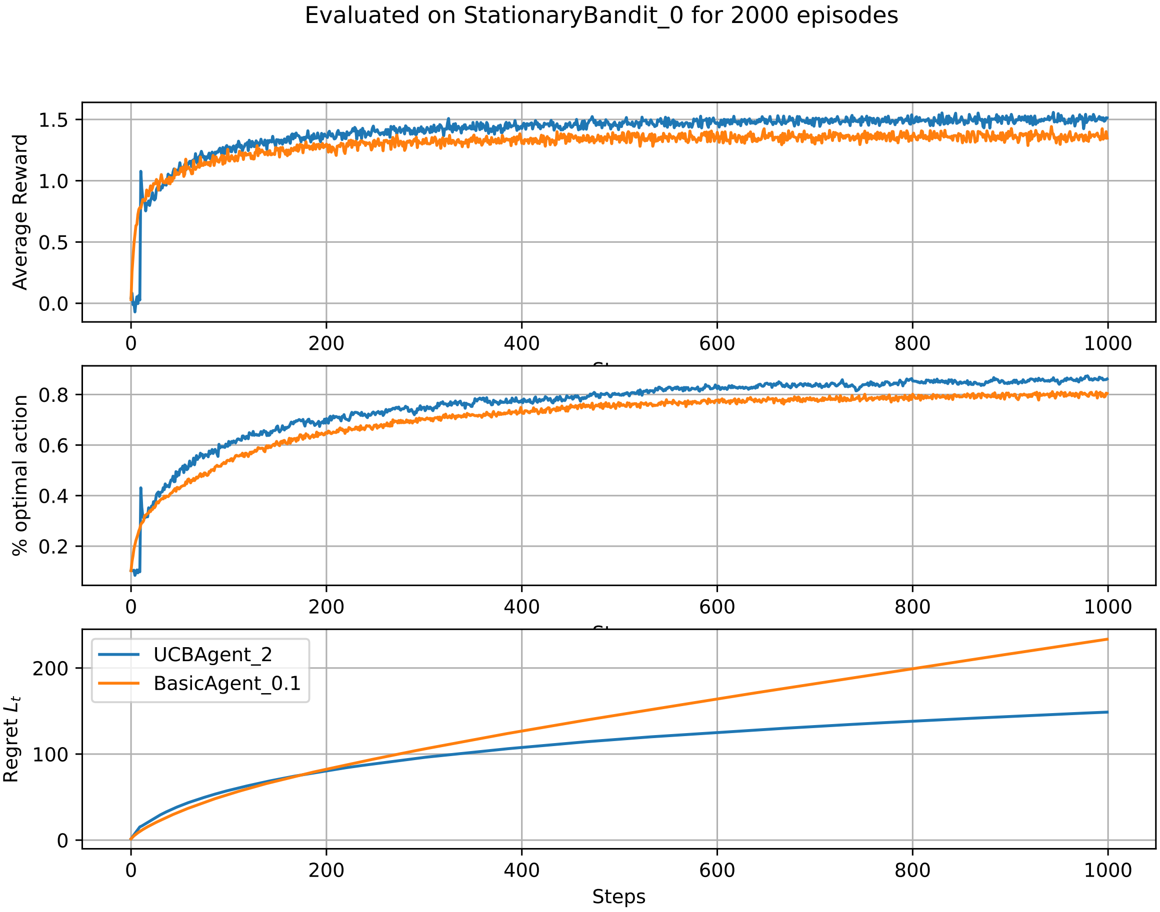
5 Upper Confidence Bound (UCB) (`ucb_agent.py`)

In contrast to the random exploration we've seen so far, the UCB method provides a systematic approach by taking into account the uncertainty about the underlying probabililty distributions of the levers. This takes into account that actions that are less explored have a reasonably high potential for optimality and they're as a consequence picked more often initially.

Problem 2 *UCB action potential*

Implement the UCB bandit algorithm and compare it against an epsilon-greedy bandit algorithm. When done, you should re-produce [SB18, Fig. 2.4]

Info: When done, you should get the same output as in [SB18]



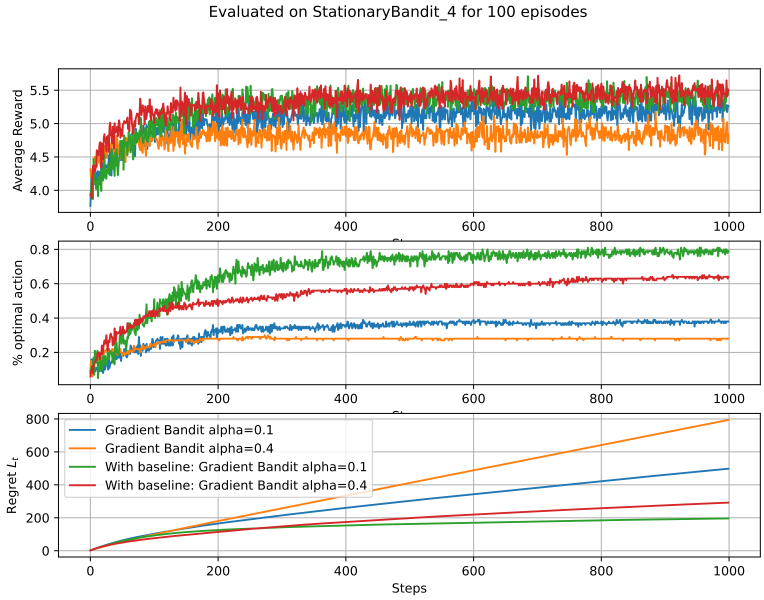
6 Gradient bandits (`gradient_agent.py`) ✶✶

This problem will be about the gradient bandit algorithm where we will re-produce [SB18, Fig. 2.5], where the bandit problem is assumed to (optionally) include a baseline reward of four. Note that the gradient bandit is not part of the reading for today, so this is an optional exercise.

Problem 3 *Gradient Bandits*

Update the simple bandit environment to include the baseline if you have not already done so. Then implement the gradient bandit algorithm by picking actions with probability [SB18, Eqn. (2.11)] and update the action preference vector using [SB18, Eqn. (2.12)] during training.

Info: When done, you should get the same output as in [SB18]



7 A nonstationary bandit problem (`nonstationary.py`) ✸

This exercise is based on [SB18, Exercise 2.5]. Implement the new non-stationary bandit class which changes the mean of the reward distribution by adding a small normally distributed variable to each coordinate which has mean 0 and standard deviation of 0.01.

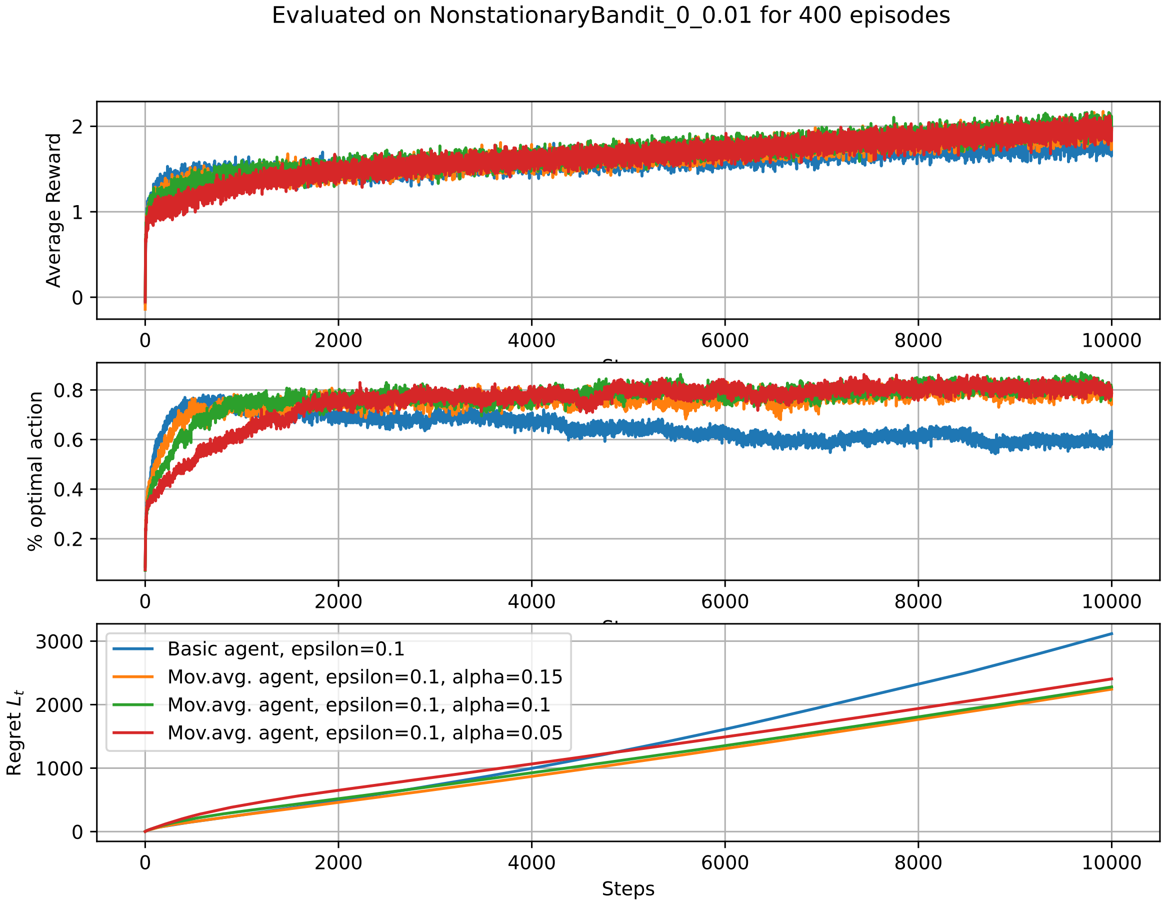
When done, implement a new bandit agent which should be similar to the basic one described in [SB18, Section 2.4], but with moving average α parameter as described in [SB18, Eqn. (2.3)] which allows it to forget the past and therefore adapt.

We will use $10'000$ time steps, but it is recommended you start out with less runs/steps first to test your methods.

Problem 4 *Nonstationary bandits (Ex. 2.5)*

Solve [SB18, Exercise 2.5] using the hints above. We will use a few more values of α, but feel free to only try one value.

Info: When done, you should get the same output as in [SB18]



8 The grand bandit race (`grand_bandit_race.py`) ✸✸

Let's try to summarize what we have seen today by comparing all our agents; note this time around the cache can be turned on.

Problem 5 *Big comparison*

Complete the script to run all bandit algorithms on the principal settings we have seen today. It will take some time, so consider using the cache system.

References

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. (Freely available online).