Assignment: Exploration/Exploitation on Bandits
Course: Reinforcement Learning, Leiden University
Written by: Thomas Moerland

In this assignment, you will study three different Bandit algorithms:

- ϵ-greedy (exploration parameter: `epsilon`)
- Optimistic initialization with greedy action selection (exploration parameter: `initial_value`)
- Upper confidence bounds (UCB) (exploration parameter: `c`)

Your research question Your goal is to investigate these three algorithms. In particular, you will:

1. Implement these algorithms.
2. Investigate the effect of different exploration parameters for each algorithm.
3. Compare the performance of the three methods.

The bandit environment We will study these algorithms on a procedurally generated set of bandits. Each bandit instance is initialized as follows:

- The bandit has 10 arms.
- The mean pay-off of each arm is randomly drawn from a uniform distribution: $\mu_a \sim \text{Uniform}(0, 1)$.
- The reward we get at each pull of an arm is a 0/1 variable with mean μ_a, i.e., $r_a \sim \text{Bernouilli}(\mu_a)$.

First carefully read the preparation instructions, and good luck!

Preparation

Python You need to install Python 3, the packages `Numpy`, `Matplotlib`, and `SciPy`, and an IDE of your choice.

Files You are provided with four Python files:

- `BanditEnvironment.py`: This file procedurally generates a Bandit instance. Inspect the code and make sure you understand it. Run the file to sample from each action once, and observe a reward.
- `BanditPolicies.py`: This file contains placeholder classes for the three Bandit algorithms you will implement: `EgreedyPolicy`, `OIPolicy`, and `UCBPolicy`. Currently, each policy randomly returns an action. Your goal is to implement the correct `init()`, `select_action()`, and `update()` methods for each class. Run the file and verify that they work for random action selection.
- `BanditExperiment.py`: In this file you will write your experiment code.
- `Helper.py`: This file contains some helper classes for plotting and smoothing. You can choose to use them, but are of course free to write your own code for plotting and smoothing as well. Inspect the code and run the file to verify that your understand what they do.

Handing in You need to hand in:

- A **report** (pdf) of **maximum 6 pages!!**. Be sure your report:
  - Describes your methods (include equations).
  - Shows results (figures).
  - Interprets your results.
- All **code** to replicate your results. Your submission should contain:
  - The original `BanditEnvironment.py` and `Helper.py`
  - Your modified `BanditPolicies.py`.
  - Your modified `BanditSolution.py`, which upon execution should produce all your plots, and save these to the current folder.

Be sure to verify that your code runs from the command line, and does not give errors!

Warning: common errors (with statistical experiments).

- Average your results over repetitions (since each runs is stochastic)!
- In each repetition, really start from scratch, i.e., randomly initialize a new Bandit environment, and initialize your policy from scratch.
- Do not fix any seeds within the loop over your repetitions! Each repetition should really draw a new Bandit.
- Average your curves over repetitions. If necessary, apply additional smoothing to your curves.

1 ϵ-greedy

You first decide to study an ϵ-greedy exploration strategy for this problem. You proceed in four steps:

a Correctly complete the class `EgreedyPolicy()` in the file `BanditPolicies.py`.

- Initialize the means $Q(a)$ and counts $n(a)$ for each action to 0.
- Implement the following ϵ-greedy policy:

$$\pi_{\epsilon-\text{greedy}}(a) = \begin{cases} 1-\epsilon, & \text{if } a = \arg\max_{b\in\mathcal{A}} Q(b) \\ \frac{\epsilon}{(|\mathcal{A}|-1)}, & \text{otherwise} \end{cases} \tag{1}$$

- Implement the following update (using an incremental update of the means):

$$n(a) \leftarrow n(a) + 1 \tag{2}$$

$$Q(a) \leftarrow Q(a) + \frac{1}{n(a)}\Big[r(a) - Q(a)\Big] \tag{3}$$

Verify that your code works by running `BanditPolicies.py`.

b Write a function `run_repetitions()` in `BanditExperiment.py`. Your function should repeatedly test the policy `EgreedyPolicy()` on an instance of `BanditEnvironment()`.

- Run a single Bandit experiment for `n_timesteps=1000` steps (Algorithm 1).
- Run `n_rep=500` repetitions of this experiment. **Be sure to initialize a clean policy and environment instance for each repetition** (without fixing a seed)!
- Average the learning curves over your `n_rep=500` experiments, smooth your curve with `smoothing_window=31`, and plot the result.

Algorithm 1: Pseudocode for single Bandit experiment.

Input: Exploration parameter, maximum number of timesteps T.
Initialization: Initialize policy $\pi(a)$
for $t = 1...T$ **do**
  Sample next action: $a_t \sim \pi(a)$
  Sample reward from environment: $r_t \sim p(R|a_t)$
  Update π based on observations (a_t, r_t)
end

Experiment a bit with varying your hyperparameters: `epsilon`, `n_timesteps`, `n_rep`, and `smoothing_window`. Get a feeling for how they affect your results.

c You decide to run a more structured experiment.

- You will test the following values for `epsilon`: `[0.01,0.05,0.1,0.25]`.
- Run your function from 1b for these different values of epsilon, where you again average over `n_rep=500` repetitions of `n_timesteps=1000` steps, and smooth your curve with `smoothing_window=31`.

- Plot the average performance for each setting of epsilon in a single graph. **Add a legend, and label the x and y-axis appropriately**. You could use the `LearningCurvePlot()` class for this.

d Write the first section of your report. Describe:

- Your methodology (with equations).
- Your results (graphs).
- Interpret the results, give possible explanations.

2 Optimistic Initialization

You decide to try another bandit algorithm: optimistic initialization with greedy action selection. You follow the same scheme as for your previous experiments.

a Correctly complete the methods of `OIPolicy()` in the file `BanditPolicies.py`.

- Initialize the estimates for the mean $Q(a)$ of each arm to `initial_value`.
- Implement the greedy policy:

$$\pi(a) = \begin{cases} 1, & \text{if } a = \arg\max_{b \in \mathcal{A}} Q(b) \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

- For updating, we will this time try a learning-based update rule for the mean (useful for non-stationary problems):

$$Q(a) \leftarrow Q(a) + \alpha \Big[r - Q(a) \Big] \tag{5}$$

  for learning rate α. We will fix $\alpha = 0.1$ in these experiments.

Verify that your code works by running `BanditPolicies.py`.

b Test your `OIPolicy()` policy over repetitions, like in question 1b. You have two options:

- Write a completely new function (e.g., `run_repetitions_oi()`) in `BanditExperiment.py`.
- Modify your previous `run_repetitions()` to take an argument `policy_type`, to make it work for either `policy_type='egreedy'` or `policy_type='oi'`.

Again, **be sure to initialize a clean policy and environment instance for each repetition** (without fixing a seed)! Verify that your code runs and produces reasonable curves, and play around with a few settings of `initial_value`.

c Run a structured experiment like question 1c.

- Test different values for `initial_value`: `[0.1,0.5,1.0,2.0]`.
- Again average over `n_rep=500` repetitions of `n_timesteps=1000` steps, and smooth your curve with `smoothing_window=31`.
- Plot the average performance for each initial value in a single graph. **Add a legend, and label the x and y-axis appropriately**. You could use the `LearningCurvePlot()` class for this.

d Write a results section in our report with:

- Your methodology (with equations).
- Your results (graphs).
- Interpret the results, give possible explanations.

3 Upper Confidence Bounds

You are getting warmed up now, and decide to repeat the procedure once more for the Upper Confidence Bounds (UCB) algorithm, that incorporates uncertainty into its decision making.

a Correctly complete the methods of `UCBPolicy()` in the file `BanditPolicies.py`.

- Initialize a vector with means $Q(a)$ and counts $n(a)$ of 0 for each action.
- Implement the UCB policy:

$$\pi(a) = \begin{cases} 1, & \text{if } a = \arg\max_{b \in \mathcal{A}} \left[Q(b) + c \cdot \sqrt{\frac{\ln t}{n(b)}} \right] \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

  where t is the timestep and $c \in \mathbb{R}^+$ is the exploration constant. **Importantly, when $n(a) = 0$, we treat the estimate for that action as infinity** (untried actions are always preferred over actions that have at least a single try).
- For updating, use the incremental learning rule of the first part again:

$$n(a) \leftarrow n(a) + 1 \tag{7}$$

$$Q(a) \leftarrow Q(a) + \frac{1}{n(a)} \Big[r - Q(a) \Big] \tag{8}$$

b Verify that your code works by running `BanditPolicies.py`.

- Again test your policy over repetitions, either modifying the function `run_repetitions()` for argument `policy='ucb'`, or writing a new function.

Again, **be sure to initialize a clean policy and environment instance for each repetition** (without fixing a seed)! Verify that your code runs, and that you get a reasonable learning curve. Play around with a few different values of `c`, and observe the effect.

c Run a more structured experiment:

- Test values for `c`: `[0.01,0.05,0.1,0.25,0.5,1.0]`.
- Again average over `n_rep=500` repetitions of `n_timesteps=1000` steps, and smooth your curve with `smoothing_window=31`.
- Plot the average performance for each initial value in a single graph. **Add a legend, and label the x and y-axis appropriately**. You could use the `LearningCurvePlot()` class for this.

d Write a results section in our report with:

- Your methodology (with equations).
- Your results (graphs).
- Interpret the results, give possible explanations.

4 Comparison

Finally, you decide to structurally compare your three approaches. You proceed in two steps:

a First, you compute the average reward over all runs for each of the above settings. In equations, you compute the mean return $\bar{r}$:

$$\bar{r} = \frac{1}{(N \cdot T)} \sum_{n=1}^{\mathrm{N}} \sum_{t=1}^{T} r_{t,n}$$

where N is the number of repetitions, and T is the number of timesteps, for each of the following settings :

- `epsilon`: `[0.01,0.05,0.1,0.25]`.
- `initial_value`: `[0.1,0.5,1.0,2.0]`.
- `c`: `[0.01,0.05,0.1,0.25,0.5,1.0]`.

(You already have these results from the previous runs, and only need to compute the means.) Then, you make a plot like Figure 2.6 from Sutton and Barto, where the horizontal axis plots the exploration parameter value, and the vertical axis plots the associated mean reward over the first 1000 steps. **Use a logarithmic scale for the x-axis!**. **Add a legend, and labels to your x and y-axis**. You could use the `ComparisonPlot()` class for this.

b Based on your results from questions a, you pick the best setting for each method, i.e., `epsilon_optimal`, `initial_value_optimal` and `c_optimal`. Then, you plot the learning curves of these three optimal cases in a single graph (you may use the `LearningCurvePlot()` class). **Add a legend that properly explains each line, and labels to your x and y-axis**.

c Write a results section in your report, where you:

- Describe your methods.
- Show both result graphs of a and b.
- Interpret the results, give possible explanations. Which of these algorithm do you prefer? Why? How could the others be improved?