ASSIGNMENT 1: EXPLORATION/EXPLOITATION ON BANDITS

Daniël Zee
s2063131

Noëlle Boer
s2505169

1 INTRODUCTION

For this assignment, we studied three different bandit algorithms: ϵ-greedy, optimistic initialization with greedy action selection and upper confidence bounds (UCB). Each of the algorithms was implemented in Python using the provided skeleton code. We then used our implementations to investigate the effect of different exploration parameters for each algorithm and compare the performance of the three models.
A bandit is defined by a set of discrete actions $\mathcal{A}$ (also called arms) and conditional probability distributions which define the probability of receiving a reward when selecting an action. The probability of each action being selected at a certain time step is defined in what we call the policy π, where $\pi(a)$ is the policy for action $a \in \mathcal{A}$. The goal is the find a policy that maximises the total rewards after a given number of time steps. A Bandit algorithm follows the following steps:

1. Specify the number of time steps T.
2. Initialise the policy $\pi(a), \forall a \in \mathcal{A}$.
3. For every time step until T, do the following:
   3.1. Sample an action a from $\pi(a)$.
   3.2. Observe the reward r after taking action a.
   3.3. Update $\pi(a)$ based on the reward r given by action a.

In the final step we use the received reward of an action to update our policy and hopefully make a better action decision on later time steps. To do this we keep track of the action value $Q(a)$ of each action, which is the expected reward pay-off. After receiving the reward of an action, we update our estimate of $Q(a)$ in the direction of the received reward. This updating can be done multiple ways, as will be discussed later. The updated estimates of $Q(a)$ can then be used by the policy to make a new action decision at the next time step. For a bandit algorithm we therefore have to decide on three aspects: the initial estimates of $Q(a)$, the policy (how we select an action based on the values of $Q(a)$) and the update (how we update $Q(a)$ after receiving reward r from action a).
A well functioning bandit algorithm makes these decisions in a way to find a balance between exploitation and exploration. Exploitation involves selecting the action a with the current highest $Q(a)$ in order to hopefully continue receiving high rewards from it. Exploration on the other hand involves selecting a seemingly sub optimal action a in the hope that the rewards increase $Q(a)$ enough as to overtake the current highest $Q(a)$. Most bandit algorithms therefore have exploration parameters which can be tweaked to adjust the balance between exploitation and exploration.

2 METHODOLOGY

We were provided with skeleton code for implementing our algorithms. The file `BanditPolicies.py` contained pre made classes for all three algorithms. These classes have three methods that correspond to the three design aspects of a bandit algorithm as discussed before: `__init__` (initialise the values of $Q(a)$), `select_action` (implement the policy $\pi(a)$ and return the chosen action) and `update` (implement the update function for updating $Q(a)$ after receiving the reward).
We were also provided with code for generating a bandit environment, which we used to investigate the performance of our three algorithm implementations. The file `BanditEnvironment.py` contained a pre made class `BanditEnvironment` that after initialisation computes the mean

reward pay-off of each action, which is randomly drawn from a uniform distribution: $\mu_a \sim$ Uniform(0,1). The `act` method can then be used to compute the reward for chosing action a, which is a random 0/1 variable with mean μ_a.
For our experiments we wrote our code in the provided `BanditExperiment.py` file. We wrote a function `run_repititions`, which receives the following parameters:

- `n_actions`: The number of actions the bandits should have.
- `n_timesteps`: The number of time steps the algorithm should perform.
- `n_repetitions`: The number of repetitions we should perform.
- `policy_type`: Which of the three algorithms we want to experiment on.
- exploration parameters (`epsilon`, `initial_value`, `c`): Parameters used to set the level of exploration for each algorithm.

The function runs the specified bandit algorithm on a generated bandit environment for the specified number of time steps. Every repetition initializes a clean policy and bandit environment. The rewards for every time step at each repetition are stored in a NumPy array of size `n_repetitions` $\times$ `n_timesteps`. After running all repetitions we return the average reward over all repetitions at each time step r_t as follows:

$$r_t = \frac{1}{N} \sum_{n=1}^{N} r_{t,n} \tag{1}$$

where N is the number of repetitions. These average results will be used to plot the learning curves of each algorithm using different exploration parameter values and to compare the algorithms.
For all experiments we used the following settings: `n_actions` $= 10$, `n_timesteps` $= 1000$, `n_repetitions` $= 500$.

3 ϵ-GREEDY

The ϵ-greedy algorithm uses an extension of the greedy policy to introduce exploration. A greedy policy always selects the action a with the current highest $Q(a)$. This is a great way to exploit the current best action but has the risk of never finding the optimal action as we don't explore by selecting sub optimal actions. ϵ-greedy addresses this by introducing the exploration parameter ϵ, which is a value between 0 and 1 that indicates the probability of a sub optimal action being selected by the policy. The algorithm initializes the Q(a) values for each action to 0 and uses an incremental update of the means to update Q(a) after receiving the rewards from action a. Using this strategy, $Q(a)$ will always be equal to average of all the rewards previously received by action a. Earlier and more recent time steps will therefore have equal weight in computing $Q(a)$.

3.1 METHODOLOGY

For the ϵ-greedy algorithm we implemented the methods for the `EgreedyPolicy` class as follows:

1. `__init__`: We initialise two NumPy arrays of size `n_actions` with zeros. One is used to store $Q(a)$ for every action and the other for $n(a)$, the number of times an action has been chosen by the policy.
2. `select_action`: We implemented the following ϵ-greedy policy:

$$\pi_{\epsilon-greedy}(a) = \begin{cases} 1-\epsilon, & \text{if } a = \arg\max_{b \in \mathcal{A}} Q(b) \\ \frac{\epsilon}{(|\mathcal{A}|-1)}, & \text{otherwise} \end{cases} \tag{2}$$

3. `update`: We implemented the following function for incremental update of the means:

$$n(a) \leftarrow n(a) + 1 \tag{3}$$

$$Q(a) \leftarrow Q(a) + \frac{1}{n(a)}[r(a) - Q(a)] \tag{4}$$

For our experiment we ran the `run_repetitions` function using the following values for `epsilon` : $[0.01, 0.05, 0.1, 0.25]$. We then compared the performance of the different settings using the `LearningCurvePlot` class provided in `Helper.py`. The learning curves were smoothed using the `smooth` function in `Helper.py` with `smoothing_window` $= 31$.

3.2 Results

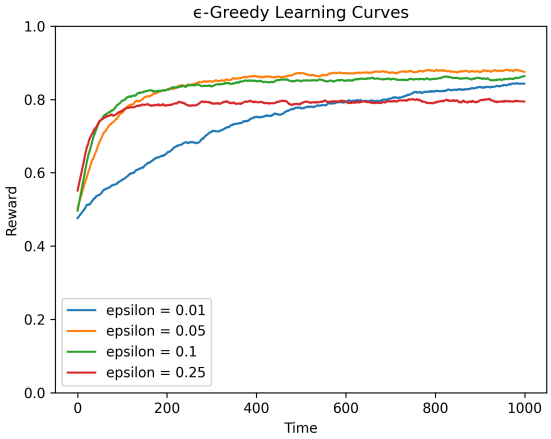

Figure 1: Average performance of the ϵ-greedy algorithm on a 10-armed bandit environment over 5000 repetitions.

The results show a clear distinction in performance based on the value of ϵ. Setting ϵ to 0.01, we see a slowly increasing learning curve that does not seem to plateau after 1000 time steps. This can be explained by the fact that the exploration rate is relatively low using this setting, so the algorithm is not eager to try sub optimal actions, making the learning process slower. Setting ϵ to 0.25, we see a quickly increasing learning curve that plateaus at an average reward just shy of 0.8. In this case the exploration rate is relatively high, which results in an algorithm that is very eager to try sub optimal actions, but does not have enough exploitative power to find the action with the highest mean reward. Setting ϵ to 0.05 or 0.1, we see similar learning curves. This indicates that between these values for ϵ we found a good balance between exploration and exploitation.

4 Optimistic Initialization

The next algorithm we implemented is called "optimistic initialization with greedy action selection" (OI). We previously stated that we do not have any exploration when we have a greedy action selection policy. This is true when we initialise our $Q(a)$ values to 0, as we did in the ϵ-greedy algorithm. In that case the policy will select a random action at first (when all $Q(a)$ values are 0) until it receives a reward and then continue to only select that action in the future. A way to introduce exploration without changing the policy is by initialising our $Q(a)$ values to a number higher than the highest expected average reward of an action. By doing this, our $Q(a)$ values will initially only decrease in time. Actions which have been explored less will therefore stay at higher $Q(a)$ values which incentivises the greedy policy to select those actions at future time steps. The exploration parameter is thus the initial value for each $Q(a)$. This algorithm uses a learning-based update rule for the mean to update $Q(a)$. This tragedy weights more recent observations more heavily, which results in the optimistic initial values of $Q(a)$ losing weight over time.

4.1 Methodology

For the OI algorithm we implemented the methods for the `OIPolicy` class as follows:

1. `__init__`: We initialise a NumPy array of size `n_actions` with the specified `initial_value`. These are the initial $Q(a)$ values for every action.
2. `select_action`: We implemented the following greedy policy:

$$\pi(a) = \begin{cases} 1, & \text{if } a = \arg\max_{b \in \mathcal{A}} Q(b) \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

3. `update`: We implemented the following function for a learning-based update rule for the mean:

$$Q(a) \leftarrow Q(a) + \alpha[r - Q(a)] \tag{6}$$

where α is the learning rate.

For our experiment we ran the `run_repetitions` function using the following values for `initial_value` : $[0.1, 0.5, 1.0, 2.0]$ and fixed the learning rate $\alpha = 0.1$. We compared the performance of the different settings using the same setup as the previous algorithm.

4.2 Results

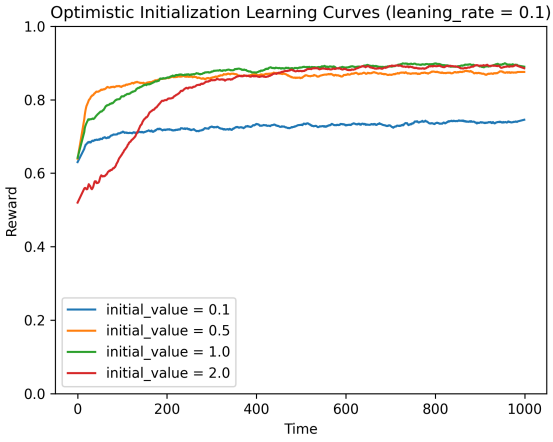

Figure 2: Average performance of the OI algorithm on a 10-armed bandit environment over 5000 repetitions.

The results show a clear distinction in performance based on the chosen initial value. Setting the initial value to 0.1 results in a very poorly performing algorithm compared to the other tested settings. When interpreting these results we need to keep in mind that the actual mean reward pay-off of each action is uniformly drawn between 0 and 1. So the mean reward pay-off over all actions lies at 0.5. Since 0.1 is much smaller than is, we do not get the exploratory behaviour as described before. Increasing the initial value to 0.5 shows a big boost in performance. Using this setting, we see that the average reward jumps up rapidly. This is because the initial value is already quite close the actual mean reward pay-off for most actions so $Q(a)$ does need many time steps to move. A downside of this that the exploratory power of the initial values also quickly decreases and thus does not reach the optimal policy after all time steps. Increasing the initial value to 1.0 or 2.0 shows us that our learning curve increases more slowly, as the higher initial values need more time to adjust. We do however see that this gives us more exploratory power and reaches a higher average reward after all time steps.

5 Upper Confidence Bounds

The Upper Confidence Bounds algorithm (or UCB for short) tries to find a balance between exploitation and exploration by calculating a UCB-score to decide the next action. The score combines the

average reward and the degree of confidence on this average reward. A lower confidence gets a higher reward. This because this arm of the bandit has a potential higher reward. When the algorithm has done several rounds, the confidence will increase and will focus more on exploitation. The exploration parameter here is $c \in \mathbb{R}^+$, also called the exploration constant.

5.1 Methodology

For the UCB algorithm we implemented the methods for the `UCBPolicy` class as follows:

1. `__init__`: We initialise two NumPy arrays of size `n_actions` with zeros. One is used to store $Q(a)$ for every action and the other for $n(a)$, the number of times an action has been chosen by the policy.
2. `select_action`: We implemented the following UCB policy:

$$\pi(a) = \begin{cases} 1, & \text{if } a = \arg\max_{b \in \mathcal{A}} \left(Q(b) + c\sqrt{\frac{\ln t}{n(b)}} \right) \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

   When $n(b) = 0$, we evaluate the expression between the brackets as ∞.
3. `update`: We implemented the function for incremental update of the means as shown in equaltions 3 and 4.

For our experiment we ran the `run_repetitions` function using the following values for `c` : $[0.01, 0.05, 0.1, 0.25, 0.5, 1.0]$. We compared the performance of the different settings using the same setup as the previous two algorithms.

5.2 Results

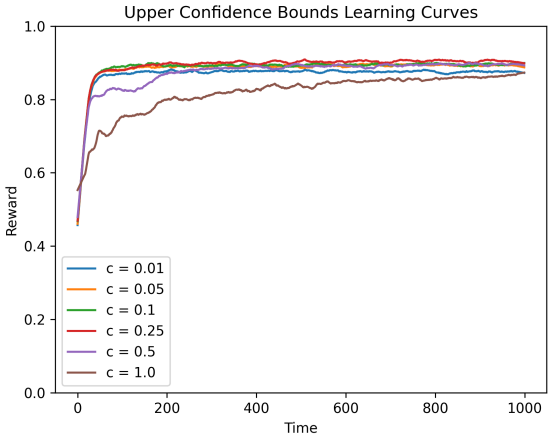

Figure 3: Average performance of the UCB algorithm on a 10-armed bandit environment over 5000 repetitions.

A setting of c equal to 1 performs the worst of all settings. We can also see that for a while the line for c equal to 0.5 does also not perform well. The line for c equal to 0.01 seems to perform about equally well to the other settings. but when examined closely it does not converge to the same reward. A c of 0.5 or higher will probably lead to less exploration and will therefore find the high reward much slower. The setting of 0.01 will lead to a lot of exploration and will therefore not exploit enough to find the optimal reward.

6 Comparison

We compared the three different algorithms with their optimal settings in one figure to see how the algorithms perform with respect to each other.

6.1 Methodology

To compare the algorithm we first average the rewards over all time steps and repetitions $\bar{r}$ for each algorithm as follows:

$$\bar{r} = \frac{1}{N \cdot T} \sum_{n=1}^{N} \sum_{t=1}^{T} r_{t,n} \tag{8}$$

where where N is the number of repetitions and T the number of time steps. We plotted the $\bar{r}$ values against the exploration parameter settings using the `ComparisonPlot` class provided in `Helper.py`. For each algorithm we then choose the parameter setting with the highest average reward and compared their learning curves using the same setup as the previous experiments.

6.2 Results

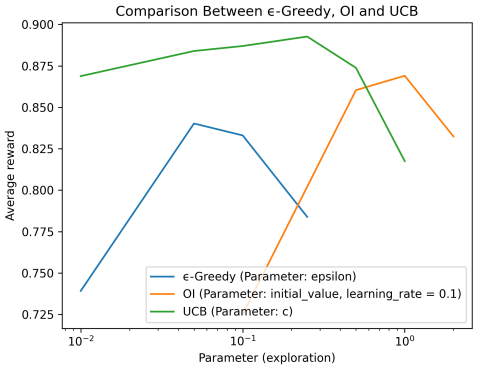

Figure 4: Average rewards over all time steps for three algorithms: Greedy, Optimistic Initialization and Upper Confidence Bounds

Figure 5: Performance of the three algorithms with optimal parameter settings.

From figure 4 we can deduce the optimal parameter settings for each algorithm by finding the highest average reward and reading the corresponding value on the x-axis parameter value. From the graph can be derived that most of the time the optimum parameter setting does not tend to lie close to bounds of the parameter domain. This indicates that the balance between exploration and exploitation is important to each algorithm. From figure 5 we can deduce which algorithm performs best using optimal parameters. We see that the UCB algorithm starts with the steepest learning curve. After a number of time steps the algorithms start to converge to the same reward value. The UCB algorithm has the steepest learning curve and then flattens. It stays above all the other lines. We can therefor conclude that this algorithm reaches the optimal reward first and performs better than the other algorithms.

7 Conclusion

We investigated three different bandit algorithms – ϵ-greedy, Optimistic Initialization, and Upper Confidence Bounds (UCB) – and how they behave with different parameter settings. ϵ-greedy exhibited a clear trade-off between exploration and exploitation, with lower values of ϵ favoring exploitation and higher values favoring exploration. Optimistic Initialization showed the importance of selecting appropriate initial values, with overly optimistic or pessimistic initialisation leading to sub optimal performance. UCB showed promising results, particularly with moderate values of c. To make sure the selected parameters are optimal we could have sampled more values of the parameter that we are tuning. For example for OI we could have tried more initial values to ensure we took the optimal parameter for comparing the algorithms. But with the selected values we get a good enough picture of what the optimal parameter should be. For future work more bandit algorithms can be chosen to compare with these three bandit algorithms. Also the influence of the parameters that were fixed in our experiment could be investigated.