Advanced Machine Learning: Bandits, Reinforcement learning

Ismail BACHCHAR [1]

Abstract

In the first part of this work, we implement three bandit algorithms, Incremental Uniform, UCB, and epsilon-Greedy, in order to compare their behavior under different bandit problem settings and performances by measuring the simple regret and cumulative regret criteria.
In the second part, we consider a reinforcement learning problem or game. First, We provide the game's environment. Secondly, we implement a Q-learning-based approach in order to learn the objective policy. In this latter, we follow the epsilon-Greedy strategy to choose the next action alongside a random approach where the agent randomly chooses the next action. Lastly, we implement a DeepLearning approach instead of Q-learning in order to approximate the Q-table.
All the code presented in this work is written in Python using the following numerical libraries: Numpy, Scipy, and TensorFlow. No RL based-library was used.

Code source: Advanced Machine Learning Bandits-Reinforcement Learning

***Keywords*—** Bandits, Q-Learning, Reinforcement learning

1. Bandit Algorithms

In this first part, we implement three bandit algorithms, Incremental Uniform, UCB, and epsilon-Greedy, from scratch. Each algorithm has its own way of pulling an arm to draw a reward. Their approaches are as follows:

- **Incremental Uniform:** This algorithm repeatedly loops through the arms pulling each arm once each time through the loop. The average rewards for each arm are tracked, and for the simple regret objective, the arm with the best average reward is returned as the best arm.
- **UCB:** The UCB algorithm. At the end of the exploration, the algorithm returns the arm that has accumulated the largest average reward.
- **ϵ-Greedy:** ϵ is a parameter of the algorithm such that the arm that appears to be the best is selected with a probability of ϵ; otherwise, a random arm is selected from among the other arms. In the end, it returns the best arm.

[1]Jean Monnet University, Saint-Etienne, France. Ismail BACHCHAR <isbachchar@gmail.com>.

1.1. Algorithms Behavior Comparison: Experiment settings

For this experiment, the objective is to compare the performance and behavior of every algorithm while searching for the best arm. To do so, we first define the bandit problem proposed in the problem statement.
We consider 8-armed bandits where everyone draws a reward in $[0, 10]$ based on a pre-defined probability distribution as follow: **Arm-0:** $\sim \mathcal{N}(5, 10)$, **Arm-1:** $\sim \mathcal{N}(3, 2)$, **Arm-2:** $\sim \mathcal{N}(10, 3)$, **Arm-3:** $\sim \mathcal{N}(7, 5)$, **Arm-4:** $\sim \mathcal{N}(0, 5)$, **Arm-5:** $\sim \mathcal{N}(1, 1)$, **Arm-6:** $\sim \mathcal{N}(10, 10)$, **Arm-7:** $\sim \mathcal{U}(0, 1)$.
After running every algorithm for 1000 trials and 400 pulls, we aggregate the simple regret, cumulated regret, and the best-founded arm per trial in order to compare the algorithms' performances.

1.2. Algorithms Behavior Comparison: Results

Best-founded arm:

As depicted in Table 1, all the algorithms, including ϵ-Greedy on with different ϵ values, were able after all the trials to agree on the best arm to be **arm-2**. It is also worth mentioning that the 0.5-Greedy algorithm is the one that pinpointed the **arm-2** as the best arm in 611 trials of 1000, more than all the other algorithms.

Simple Regret: In Figure 1, we present the simple regret obtained after 1000 trials with 400 pulls of every algorithm. We see that as the number of pulls increases, the simple regret decreases but none of the algorithms reached 0 regret. The most promising algorithm here is 0.5-Greedy; it actually reached at the end of pulling the minimum simple regret (around 2.5).

Algorithms	arm-2	arm-1	arm-5	arm-3	arm-6	arm-0	arm-7	arm-4
IU	588	169	105	55	31	19	17	16
UCB	598	172	118	43	22	18	18	11
0.5-Greedy	611	187	98	41	22	15	15	11
0.9-Greedy	542	181	123	61	31	29	20	13
0.1-Greedy	562	185	102	63	27	25	21	15

Table 1. Distribution of best-founded arms (8-arms).

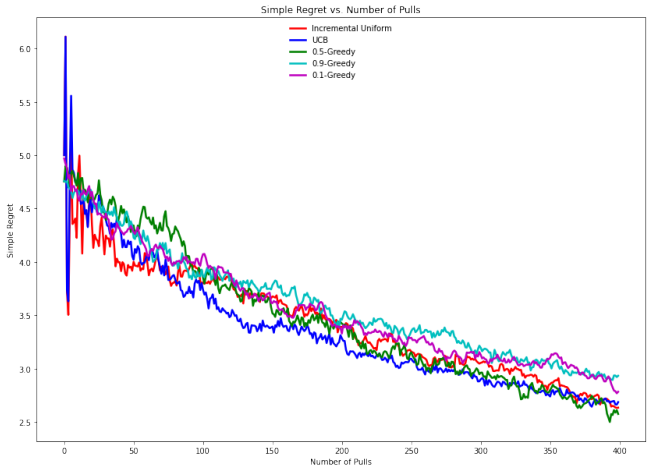

Figure 1. Simple regret comparison (8-arms).

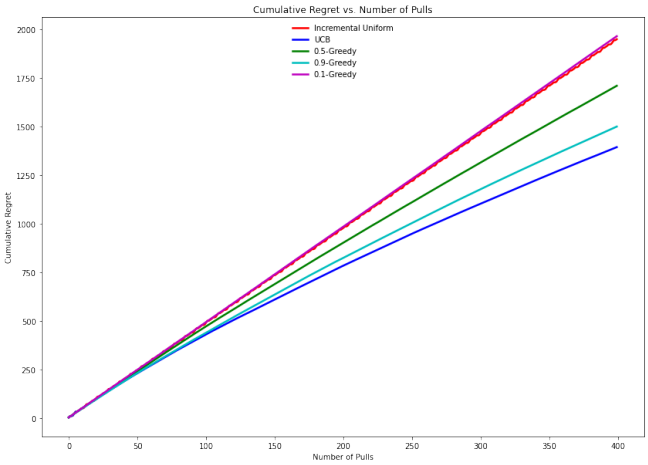

Figure 2. Cumulated regret comparison (8-arms).

Cumulated Regret: Again to continue comparing the algorithms, we also calculated their cumulated regret. The result of this comparison is shown in Figure 2, where the around the 100 pulls, the 0.1-Greedy and Incremental Uniform diverged from the other algorithms; therefore, they cumulated more regret. This is because the Greedy approach performs as the IncrementalUniform algorithm when $\epsilon \sim 0$.

1.3. Influence of Number of Arms: Experiment settings

In this part, we want to see the influence of the number of used arms in the bandit problem. Therefore, we ran two extra experiments using 4-arms for the first and 12-arms for the second with the same previous parameters, such as the trial and number of pulls while changing the distributions as follows:
For 4-arms: Arm-0: $\sim \mathcal{N}(5, 10)$, **Arm-1:** $\sim \mathcal{N}(3, 2)$, **Arm-2:** $\sim \mathcal{N}(10, 3)$, **Arm-3:** $\sim \mathcal{U}(0, 1)$.
For 12-arms: Arm-0: $\sim \mathcal{N}(5, 10)$, **Arm-1:** $\sim \mathcal{N}(3, 2)$, **Arm-2:** $\sim \mathcal{N}(10, 3)$, **Arm-3:** $\sim \mathcal{N}(7, 5)$, **Arm-4:** $\sim \mathcal{N}(0, 5)$, **Arm-5:** $\sim \mathcal{N}(1, 1)$, **Arm-6:** $\sim \mathcal{N}(10, 10)$, **Arm-7:** $\sim \mathcal{N}(9, 6)$, **Arm-8:** $\sim \mathcal{N}(3, 12)$, **Arm-9:** $\sim \mathcal{N}(8, 13)$, **Arm-10:** $\sim \mathcal{N}(8, 5)$, **Arm-11:** $\sim \mathcal{U}(0, 1)$.

Algorithms	arm-2	arm-1	arm-3	arm-0
IU	791	192	9	8
UCB	670	262	34	34
0.5-Greedy	806	183	7	4
0.9-Greedy	705	264	17	14
0.1-Greedy	743	216	27	14

Table 2. Distribution of best-founded arms (4-arms).

Algorithms	arm-2	arm-1	arm-5	arm-10	arm-3	arm-7	arm-6	arm-9	arm-0	arm-11	arm-8	arm-4
IU	442	143	97	61	56	54	36	33	22	21	19	16
UCB	524	160	123	43	36	32	20	14	14	13	12	9
0.5-Greedy	511	147	101	47	44	38	28	23	20	16	13	12
0.9-Greedy	431	178	109	52	49	41	35	28	23	21	20	13
0.1-Greedy	410	163	102	75	57	51	31	25	24	22	21	19

Table 3. Distribution of best-founded arms (12-arms).

1.4. Influence of Number of Arms: Results

As depicted in Tables 2, 3, and 1, all the algorithms regardless of the number of used arms were able to find the same best arm **arms-2** in our experiments which is also the optimal arm. The only difference we can spot from this comparison is that the distribution of founded arms differs based on the number of used arms. When there is a small number of arms, it becomes easy for the algorithm to find discover the best arm and start using it right away, such as in the 0.5-Greedy example shown in Table 2 having 806 times use of the **arm-2** out of 1000 trials.

2. Reinforcement Learning (Walking Game)

In this second part, we are given a Reinforcement Learning problem/game where the task is to guide an agent to find an optimal path to the destination by taking only four directions/walks on a 2D surface. The walks are UP, LEFT, RIGHT, and DOWN. In addition, there are three objects on the surface, Wardrobe (W) with a reward of 0, and when the agent hits it, the game is over (loss); treasure (T) with a reward of +10, and when the agent hits it, the game is over (win); and Poison (P) with a reward of -10, and when the agent hits it, the game is over (loss). The agent can freely use the empty blocks to reach its destination, which is the Treasure, but the agent gets rewarded -1 for every use of the empty block.
Four actions could define this problem: UP, LEFT, RIGHT, and DOWN, and three possible states: the agent could be either on the Empty block, the Treasure block, or the Poison block but cannot be on the Wardrobe block as it's an obstacle.

2.1. QLearning

The Q-learning algorithm is based on the state-action table. It starts with an empty table and continues updating the q-values using in order to improve the learning agent's behavior. This update is done by the Temporal Difference (TD):
$Q(s', a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha \cdot (R(s, a) + \$\gamma valueswithaccuracy$ gamma $\cdot \max_{a'} Q(s', a'))$
In our work, we implemented the algorithm, and for the experiments, we chose to tune the parameters $\$\gamma$ gamma$intheequationaboveequationand\epsilon$ from the ϵ-Greedy approach, which controls the randomness in choosing the next state.
In addition, in the implementation, we added a binary variable **play_random** that specifies either to make the ϵ-Greedy approach when choosing the next state or randomly choose the next state. This is not applied to the training part, only when the agent plays to test its ability to find the Treasure. We start by finding the best γ gamma for $\epsilon = 0.5$, and then we use the best γ gamma to tune ϵ.

γ	Accuracy
0.01	0.9666666666666667
0.2	0.9
0.35	0.9833333333333333
0.9	**1.0**
1.2	0.8166666666666667
1.9	0.7166666666666667
3	0.7333333333333333
10	0.7833333333333333

Table 4. Q-learning: γ gamma tunning.

ϵ	Accuracy
0.01	0.21666666666666667
0.1	0.3
0.2	0.6666666666666666
0.3	0.9
0.4	0.9
0.5	**1.0**
0.6	1.0
0.7	0.9833333333333333
0.8	1.0
0.9	1.0

Table 5. Q-learning: ϵ tunning (while $\gamma = 1$).

By tunning the chosen parameters, we found out that the Q-learning algorithm can achieve an accuracy of 1 as depicted in Tables 4 and 5. The accuracy is achieved in a reasonable execution time.
The algorithm performs well mainly because the chosen game configurations (these are provided in the GitHub repos-

γ	Accuracy
0.01	1.0
0.2	1.0
0.9	1.0
3	1.0
10	1.0

Table 6. Deep QLearning: γ gamma tunning.

itory inside the data folder) are not that hard.

2.2. Deep QLearning

In this part, we followed the same approach as in Q-learning, just the difference is that here we use a basic Neural Network, as presented in the figure 3, to approximate the q-table.

```
Model: "model_14"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_15 (InputLayer)       [(None, 3)]               0

 dense_42 (Dense)            (None, 3)                 12

 dense_43 (Dense)            (None, 3)                 12

 dense_44 (Dense)            (None, 1)                 4

=================================================================
Total params: 28
Trainable params: 28
Non-trainable params: 0
_________________________________________________________________
```

Figure 3. DL model settings.

ϵ	Accuracy
0.01	1.0
0.2	1.0
0.5	1.0
0.8	1.0
1	1.0

Table 7. Deep QLearning: ϵ tunning.

As depicted in Tables 6 and 7, we see that the DL based approach is very powerful as it got an accuracy of 1 from the first parameter and we should've stopped tuning from that moment but to complete the comparison we let it run for all the parameters. The problem with this approach is that it is slow, but since it's very powerful and doesn't require that much tuning of the parameters, it is still preferred to QLearning based on what we discover in this work.

Code and Data

The code implementation of this project is available on GitHub (Advanced Machine Learning Bandits-Reinforcement Learning). Presented in a reproducible manner so that similar near-results (randomization) could be re-obtained. The repository contains three Jupyter Notebooks, Bandit_algorithms: implementation and results of this work's first part, QLearning: the implementation and results of Q-learning-based approach for the RL problem/game, and finally, Deep_QLearning: where the implementation and obtained results of the RL game solution using the DL-based approach.
Considering the data, the first part of this project is more about parameter tunning and comparison, so no data was used here. In contrast to the second part -RL problem/game- we created 6 game configurations in a text file following the same logic provided in the problem statement. These files are provided in the data folder in the repository.

3. Conclusion

In this project, we first implemented three bandit algorithms and ran their experiments in order to compare their behavior and performance using best-chosen arms, simple regret, and cumulated regret criteria. Then secondly, we implemented an RL problem from scratch while defining the game's environment and two algorithms, QLearning and Deep QLearning. As a result, we found out that the latter is way better in terms of accuracy. Its only drawback is the required computation time if the tuning task is required.