CS 240 Final Project Report

Zongxiang Wu, Jing Guo, Jiaqiang Huang

December 29, 2022

Abstract

This report summarizes several algorithms mentioned in *Multi-armed Bandit Algorithms and Empirical Evaluation*. Focusing on the examples of multi-arm slot machines, this paper analyzes and compares several algorithms involved using the idea of greedy algorithms. Among them, we reproduce the code implementation of ϵ-**greedy** algorithm, **SoftMax** algorithm and **SoftMix** algorithm, and describe the **POKER** algorithm. We compared the effects of these three algorithms implemented through code experiments, and have a deeper understanding of multi-arm slot machines.

1 Introduction

The gambler's multi-armed bandit problem is to decide which arm of the K-slot machine to pull to maximize his total reward in a series of trials. Many real-world learning and optimization problems can be modeled in this way. The key point to this question is how to balance the distribution of the two operations in the game, maximizing the reward based on existing knowledge and trying new actions to acquire new knowledge, that is, the so-called exploitation and exploration trade-off of reinforcement learning.

This paper analyzes and compares several algorithms proposed in the last three decades, describes and analyzes a new algorithm **POKER**.

We have implemented three algorithms, including ϵ-**greedy**, **SoftMax** and **SoftMix**. Meanwhile, the experiments basically confirms the authors' view that the ϵ-**greedy** algorithm is still a simple and high-yield algorithm that is difficult to defeat.

2 Implemented Algorithms

Intuitively, in the problem of multi-arm slot machines, we should find the levers with higher expected reward as efficiently as possible, so that as the number of rounds increases, we become more and more clear how to pull the levers to achieve the highest reward, so as to maximize the reward of the player. This greedy thought of choosing the lever with the highest expected reward per round runs through various algorithm implementations of multi-armed bandit game.

2.1 ϵ-greedy

In this algorithm, we have a parameter ϵ, in which the gambler will pull the levers completely randomly with ϵ-frequency , and otherwise pull the lever with the current maximum estimated mean. ϵ must be in the open interval $(0, 1)$ and its choice is left to the user.

```
# ---------- epsilon greedy algorithm ----------
import numpy as np
def run_epsilon_greedy_experiment(k_num, m_bandits, epsilon, N):
    player_choice = np.empty(N)
    for i in range(N):
        p = np.random.random()
        if p < epsilon:  # exploration exploitation trade-off
```

```
            j = np.random.choice(k_num)
        else:
            j = np.argmax([a.estimated_mean for a in m_bandits])
        reward = m_bandits[j].pull()
        # updates the estimated reward
        m_bandits[j].epsilon_greedy_update(reward)
        player_choice[i] = reward
```

This simple algorithm is sub-optimal, because the constant ϵ prevents the strategy from asymptotically approaching the optimal lever. One solution is to let a reduce the number of rounds that has been carried out, so that the strategy is small enough when the number of rounds is large enough to gradually select the optimal pole. If ϵ is multiplied by the factor $\frac{1}{t}$, it is **ϵ-decreasing** algorithm. And we implement **GreedyMix** algorithm, a variant of the **ϵ-decreasing** algorithm, which multiplies ϵ by the factor $\frac{log(t)}{t}$. Where t is the current number of rounds.

```
# ---------- GreedyMix ----------
import numpy as np
def run_greedy_mix_experiment(k_num, m_bandits, epsilon, N):
    player_choice = np.empty(N)
    for i in range(N):
        p = np.random.random()
        # decreasing epsilon
        if p < (epsilon * math.log(i + 1) / (i + 1)):
            j = np.random.choice(k_num)
        else:
            j = np.argmax([a.estimated_mean for a in m_bandits])
        reward = m_bandits[j].pull()
        # update the estimated reward
        m_bandits[j].epsilon_greedy_update(reward)
        player_choice[i] = reward
```

Both of these two algorithms have time complexity $O(N)$. We can use a temporary variable to remember the current maximum estimated mean.

2.2 softmax

2.2.1 Algorithm

Softmax strategy is to update the probability that we may choose for each lever each time we pull a lever and get a reward. We update the corresponding probability of a lever using a softmax function according to the estimated mean which can be calculated by what we have just observed. The update function may look like this $p_i = \frac{e^{\frac{\hat{\mu}_i}{\tau}}}{\sum_{k=1}^{n} e^{\frac{\hat{\mu}_k}{\tau}}}$. Where τ is the temperature, a super parameter defined by the user and $\hat{\mu}_i$ represents the estimated mean which can be observed by calculating the average reward we have got in the turns before this turn. So at first, $\hat{\mu}_i$ is initialized to 0 by definition, while p_i should be $\frac{1}{n}$ calculated by the softmax function. In each turn, p_i for each lever i will be modified since the chosen one's estimated mean reward will be updated (usually increased). Generally, a lever which may return a better reward will have a larger p_i so that the algorithm will choose the certain lever more often.

While the temperature τ controls how the estimated mean reward $\hat{\mu}_i$ increases the probability p_i that a lever will be in which probability be chosen. A too-low temperature will cause a too-high learning rate, which may cause the algorithm to choose a comparably better lever instead of the best lever since the exploration times will not be high enough. On the other hand, a too-high temperature will cause a too-low learning rate, which may cause the algorithm to choose many comparably-better-lever randomly and reap a low total reward.

2.2.2 Pseudo code

Algorithm 1 Softmax algorithm

for $i = 0$ to n **do**
    para[i]=exp($\hat{\mu}_i/\tau$)
end for
for $i = 0$ to n **do**
    sum+=para[i]
end for
for $i = 0$ to n **do**
    p[i]=para[i]/sum
end for=0

2.2.3 Time complexity

Play t times and the multi-armed bandit has n arms. In each turn, update all the softmax parameter of each lever, then the time complexity is $O(tn)$.

2.3 Softmix

2.3.1 Algorithm

According to the disadvantage of the softmix algorithm, consider an improvement named softmix. In softmax, the temperature is a constant number. As a reinforcement algorithm, we generally want to explore more in the beginning while exploit more in the end. But a constant temperature will cause a too-less exploration in the beginning and a too-less exploitation in the end. So we consider add a time multiplier to the temperature to make it higher at first to decrease the learning rate. While with the time increases, the temperature will start falling to form a higher exploitation rate. A $\frac{log(t)}{t}$ seems good. So the p_i now looks like $p_i = \frac{e^{\frac{\hat{\mu}_i}{\tau * \frac{log(t)}{t}}}}{\sum_{k=1}^{n} e^{\frac{\hat{\mu}_k}{\tau * \frac{log(t)}{t}}}}$.

2.3.2 Pseudo code

Algorithm 2 Softmix algorithm

for $i = 0$ to n **do**
    para[i]=exp($\hat{\mu}_i/(\tau * log(t)/t)$)
end for
for $i = 0$ to n **do**
    sum+=para[i]
end for
for $i = 0$ to n **do**
    p[i]=para[i]/sum
end for=0

2.3.3 Time complexity

Similar with which of softmax algorithm, $O(tn)$.

3 Experiments

In the experimental environment, we set ten levers that obey the normal distribution, whose mean is from one to ten, with a variance of 4. Intuitively, the ideal algorithm should asymptotically select the lever with the maximum mean. We introduce the variable *cumulative average*, the value of which is the average value of the currently obtained rewards. The cumulative average of each round is reflected in the line chart generated by the experiment.

For the four algorithms described above, we have tried their hyper-parameters several times, and finally selected three parameters to be reflected in the result table respectively.

Strategeties	Round 100	Round 1k	Round 10k
Greedy, 0.01	0.398	0.859	0.965
Greedy, 0.05	0.326	0.858	0.964
Greedy, 0.1	0.841	0.862	0.938
GreedyMix, 1	0.877	0.983	0.992
GreedyMix, 10	0.664	0.819	0.952
GreedyMix, 100	0.561	0.597	0.875
SoftMax, 0.1	0.869	0.905	0.909
SoftMax, 1	0.579	0.673	0.67
SoftMax, 5	0.631	0.674	0.718
SoftMix, 1	0.619	0.579	0.606
SoftMix, 20	0.786	0.971	1.002
SoftMix, 100	0.622	0.866	0.986

Table 1: Experimental results for several bandit algorithms. Float numbers in each entries are the real total reward divided by the ideal total reward. Algorithms with the higher ratio have better performance at the specific number of rounds. The numbers following the strategy names is are the tuning parameters used in the experiments.

3.1 result line charts

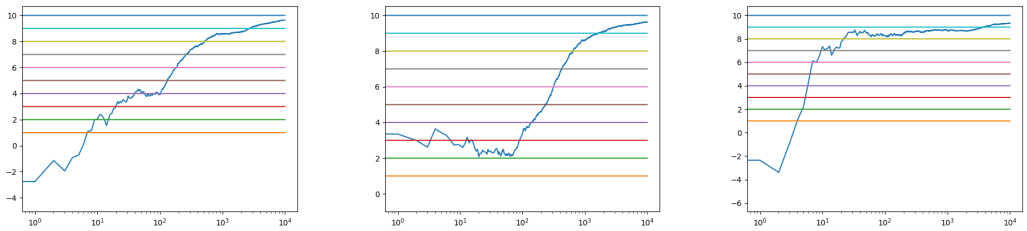

Figure 1: Result of ϵ-greedy with $\epsilon = 0.01, 0.05, 0.1$ respectively

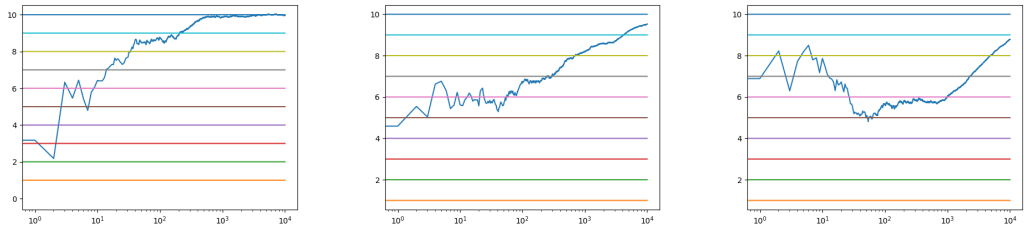

Figure 2: Result of **GreedyMix** with $\epsilon = 1, 10, 100$ respectively

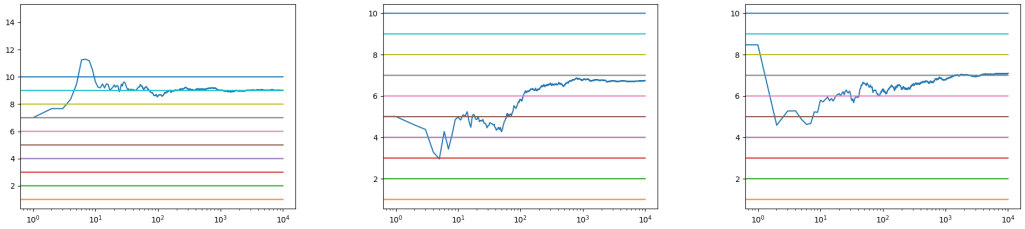

Figure 3: Result of **SoftMax** with $\tau = 0.1, 1, 5$ respectively

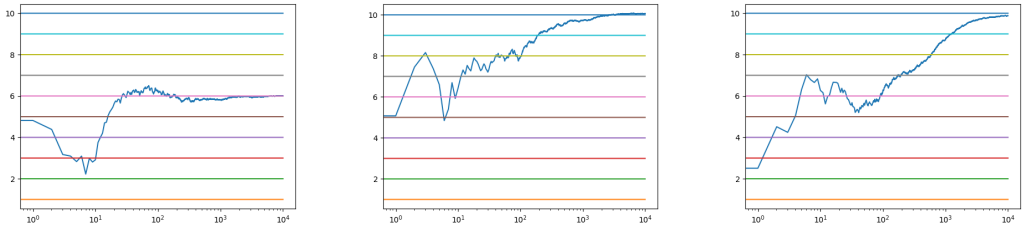

Figure 4: Result of **SoftMix** with $\tau = 10, 20, 100$ respectively