CS6700 : Reinforcement Learning Programming Assignment 1

Butruille Solène

February 3, 2020

1 Epsilon-greedy

Epsilon-greedy : this algorithm is the most simple. For each step, it takes a random value, if it is superior to the epsilon value, then we chose the arm with the best average. Otherwise, we pick a random arm. From the experiments, I observe that more the epsilon value increase, more it takes time to "find a good arm". Indeed, if we compare the 3 epsilon curves we have, more epsilon value is close to 0, more first steps rewards are good. But very quickly, curves order change. Epsilon-greedy stops evolving. On a bigger value, reward is getting better and better. Between the 3 epsilon value we took (0, 0.01, 0.1) the bigger value has the better reward average in the end. Epsilon equals 0 doesn't really have a chance, indeed, it is very quickly going to choose always the same arm but it is not necessarily the best one. The best average reward we finally have is around 1.4. Execution time for 10 arms, 1000 steps and 2000 bandit problem and one epsilon value is 30 seconds, to get the average reward and 41 seconds to have optimal percentage.

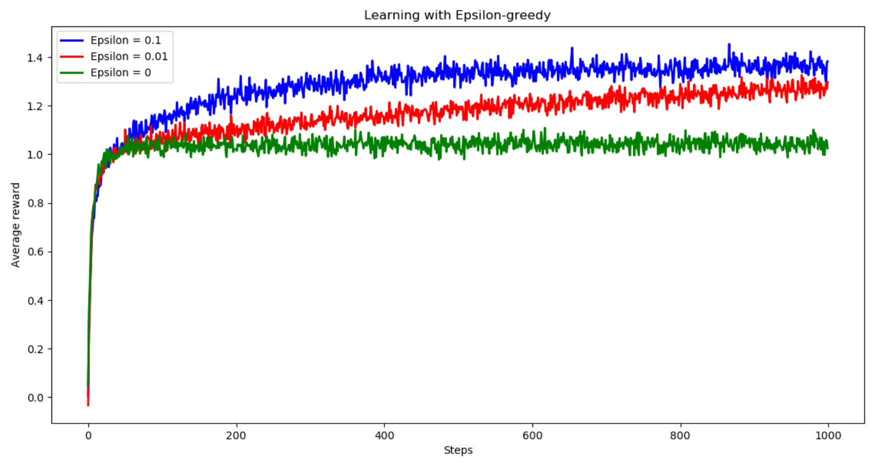

Figure 1: E-Greedy : Reward Average

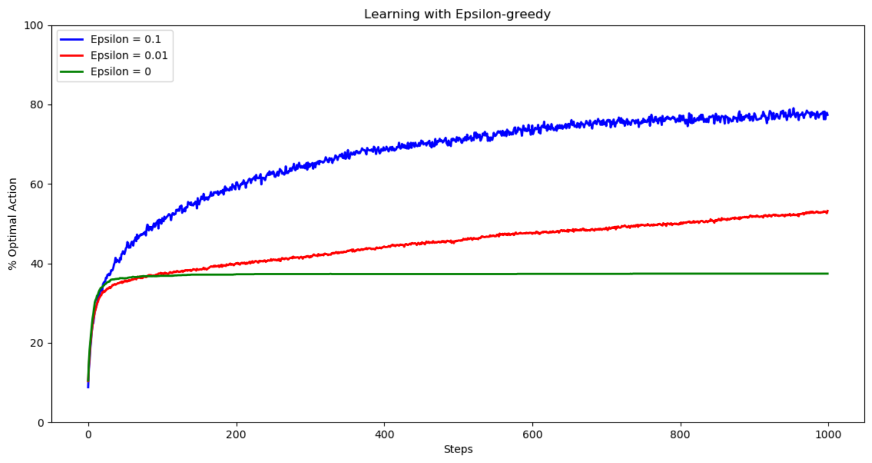

Figure 2: E-Greedy : Percentage Optimal Action

2 Softmax

SoftMax : this algorithm is more complicated. It is changing the probability to take each arm. Using the formula mu = exp(T*arm-reward)/average(mu), it will choose randomly an arm but distribution will not be linear, it will depend on the mu calculated. The parameter T (temperature) changes a lot the reward we have. If temperature equals 0, algorithm is useless because every time every arm have the same chance to be chosen, indeed, percentage of optimal action is 10 percent, probability of each arm to be chosen. From the experiments, I observe that more temperature is increasing, more it accentuates the chance to be chosen for an arm which have a good average reward. This algorithm works fine, we can obtain good average reward but it very quickly stays on the same value (no matter what is the temperature value), and stops evolving. The best average reward we finally have is around 1.4, which is not better has epsilon algorithm. It is disappointing because this algorithm is more complicated. Maybe it could be better with other values of the temperature. Execution time for 10 arms, 1000 steps and 2000 bandit problem and one temperature is 160 seconds, to get the average reward and 200 seconds to have optimal percentage.

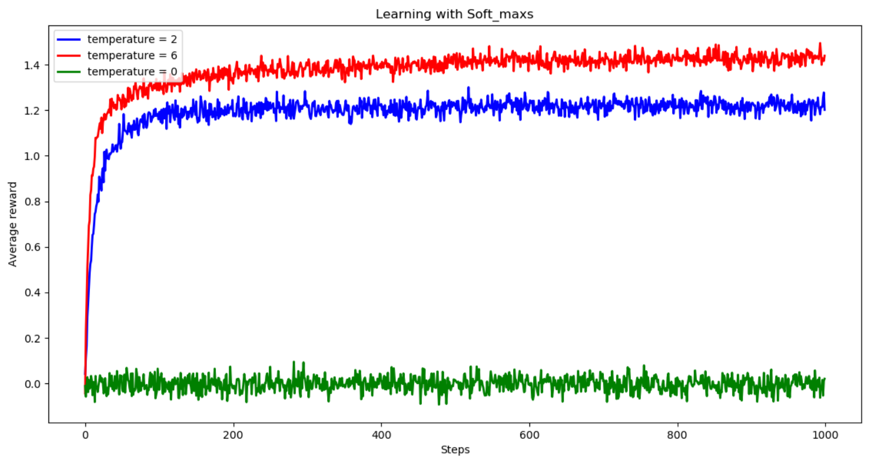

Figure 3: SoftMax : Reward Average

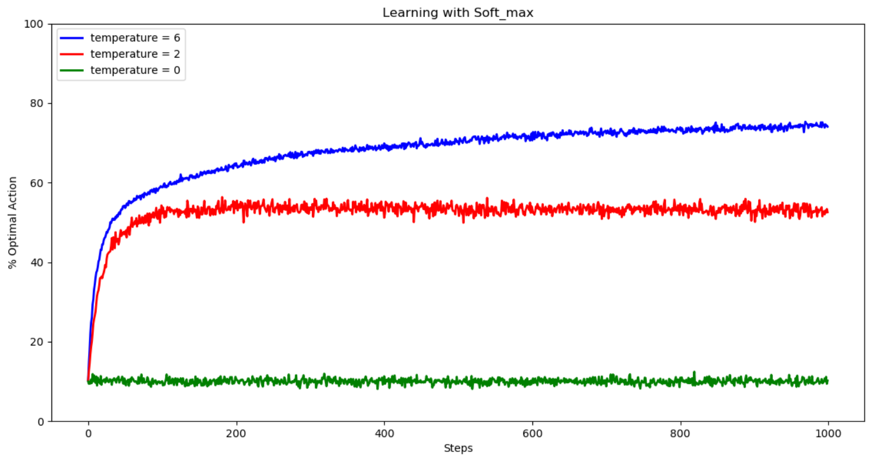

Figure 4: SoftMax : Percentage Optimal Action

3 UCB1

UCB1 : this algorithm calculates for every step the high 95 percent confidence interval (depending on each arm mean, number of time the arm was chosen and step number). It will then always choose the highest value, which means the highest probability to have next time a good reward. If we compare the performance of UCB1 with the one of epsilon greedy and softmax. We observe that UCB1 is faster than softmax and slower as epsilon. Indeed, I think that it is hard to be as fast as epsilon because this algorithm does few calculation (it only compares a random value and epsilon value). This algorithm needs to calculate each time the high 95 percent confidence interval. Whoever, it is faster than softmax. Indeed, Softmax needs the mean on every arm each time which is very long to calculate. UCB1 only needs information about the given arm to calculate the high 95 percent confidence interval (and the step number but it doesn't need to be calculate). From the experiments, I observe that the average reward we have is way more better with this algorithm compared to the 2 first ones. I think it is because this algorithm seems to find the good arm before the end and only pick the good arm in the end. Execution time for 10 arms, 1000 steps and 2000 bandit problem is 130 seconds, to get the average reward and 150 seconds to have optimal percentage.

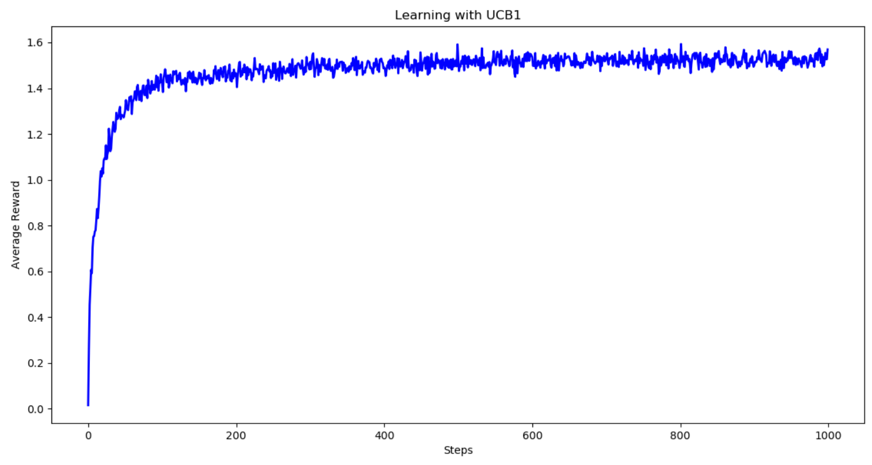

Figure 5: UCB1 : Reward Average

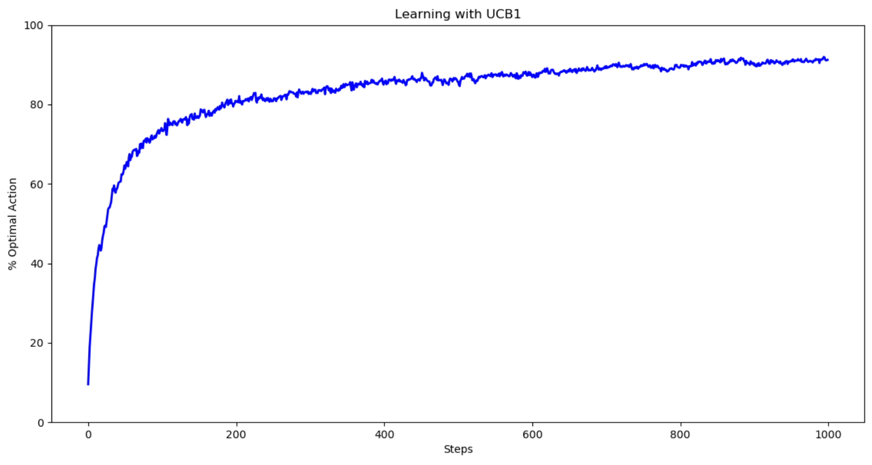

Figure 6: UCB1 : Percentage Optimal Action

4 Median Elimination

Median Elimination is an algorithm which is going to do the average reward on a certain number of times (depending on delta and epsilon) for each arm. It will every step keep going on with the arms having average reward on top of the median. Then it will again to the average reward for the remaining arms. Its performance is really slower to the other ones. Indeed, with delta and epsilon

= 0.1, it is going each time to do average reward on 295 steps. Then, it will need to calculate each arm's meaning and finally calculate the median which takes a lot of time (2950 iterations for the first round). Moreover, it is not finish because it then needs to do the process again with the remaining arms until there is just on arm left. The computational cost of computing median depends on the sorting algorithm use, but the best that we can get is O(nlog(n)). It is the rate-determining step because after calculation of the median, the algorithm will choose which arm will keep going and which one wont. It would be possible to make it faster if we could have a sorted or almost sorted list. We could try to sort the arms depending on they average and hope that the order won't change to much. We could also use the fastest sorting algorithm that we find. From the experiments, I observe that algorithm is very slow, compared to the others with epsilon and delta equals 0.1. Indeed, it really needs to do a lot of calculations. It is obtaining however good average reward (it ends with more than 1.5). If we reduce epsilon and delta (see on the figures below), we can see that there is more noise but it seems to give the same result (but they are less trustable). The average reward increases a little bit as epsilon and delta decrease. His best average reward is as good as UCB1 which was the current best score. There is no need to display percentage optimal action on the algorithm, indeed, it is always equals to ten percent because there 1 chance out of 10 to choose each arm. Execution time for 10 arms, 2000 bandit problem and epsilon=delta=0.1 : 600 seconds Execution time for 10 arms, 2000 bandit problem and epsilon=delta=0.3 : 37,5 seconds Execution time for 10 arms, 2000 bandit problem and epsilon=delta=0.5 : 11,5 seconds Execution time for 10 arms, 2000 bandit problem and epsilon=delta=1: 4 seconds

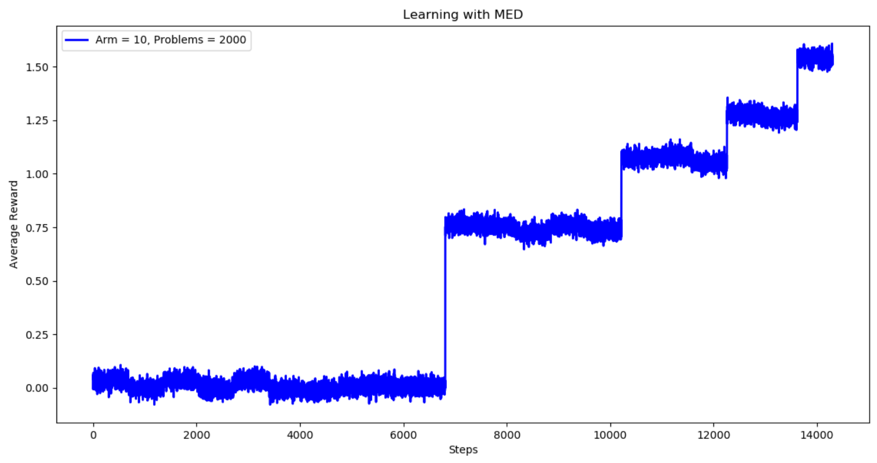

Figure 7: Reward Average Epsilon = Delta = 0.1

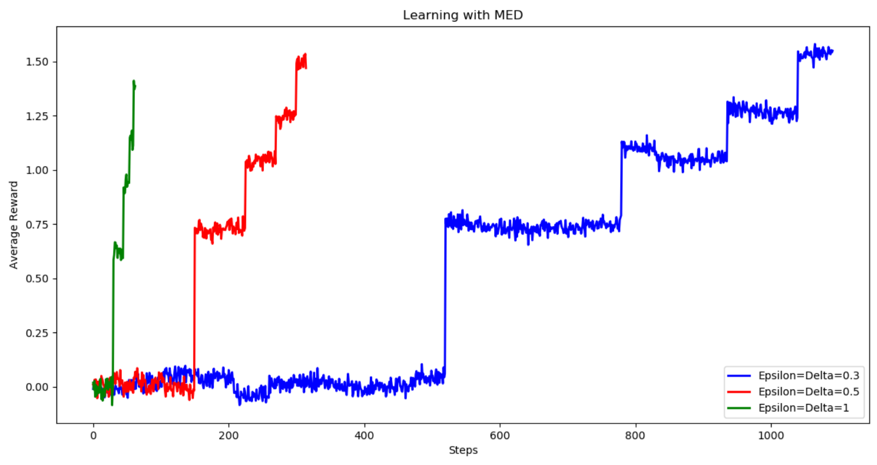

Figure 8: Reward Average other Values

5 1000 arm bandit

As the number of arms grows, it takes for every algorithm longer to execute. I tought it was going to give better results because we increase the chance of having arms with a high value for the Gaussian repartition. For MEA algorithm, it works very well because the average double from 1.5 to 3. But actually, results are not much better for the others, we really have more noise and average rewards very close (even worth in the case of UCB1). I think it is because we got every precedent result with 2000 bandit problem. Every result is already a meaning of a lot of possible arms. Having more arms does not really count compare to the number of arms we already treated. It is finally just longer because there are more calculation. For MEA, I only display for each step the average meaning, otherwise the list size is too big. Comparison of performance : Performance of epsilon : 103 seconds Performance of softmax : 6 567 seconds Performance of UCB1 : 9 422 seconds Performance of MEA : 12 678 seconds

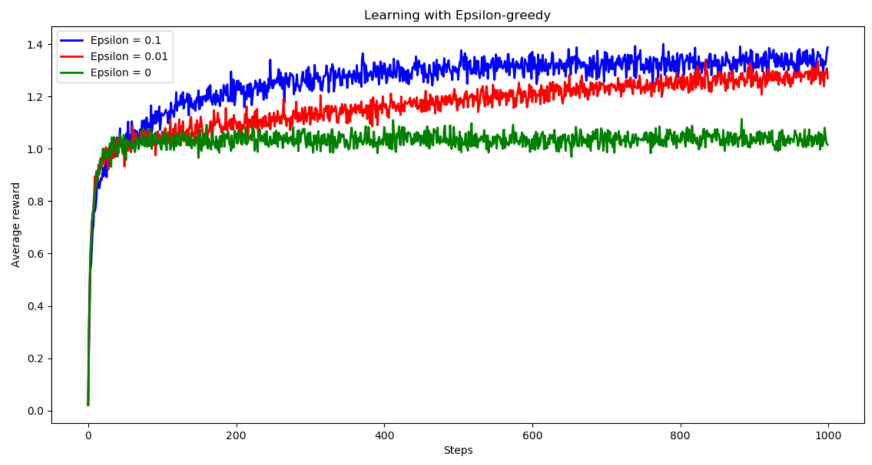

Figure 9: Reward Average E-greedy

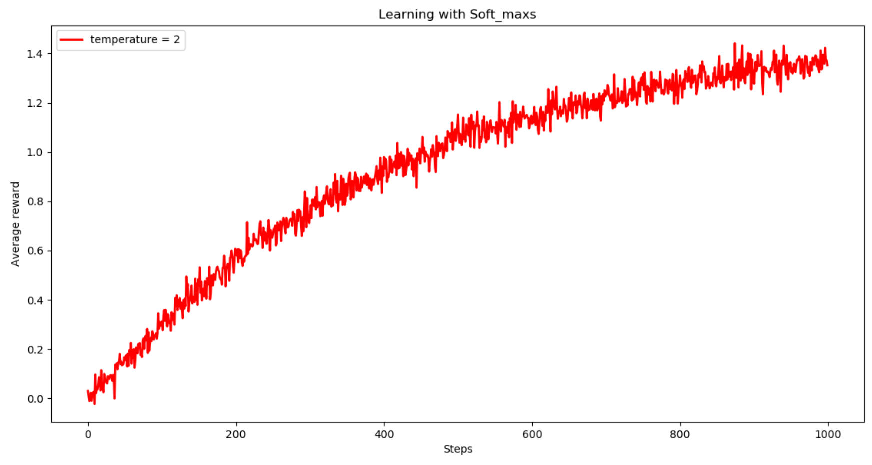

Figure 10: Reward Average Soft-Max

References

As I struggled a lot at the beginning, I had a look at this code for question 1:
https://github.com/FedorSulaev/NArmedBandit/blob/master/NArmedBandit.py

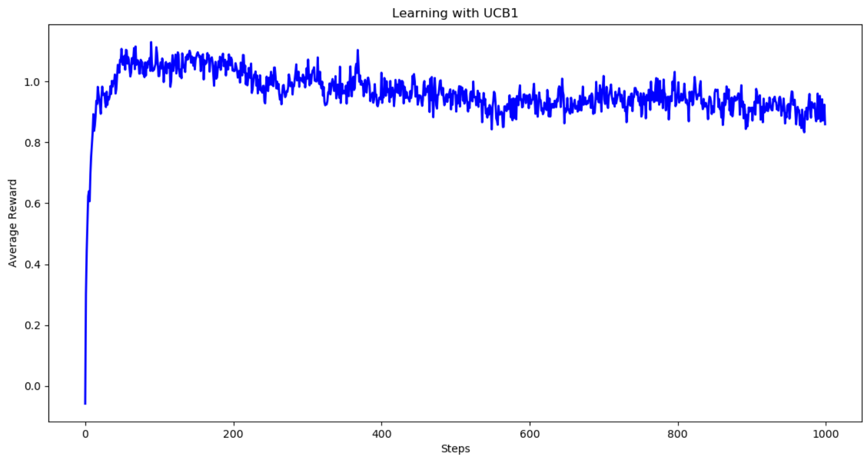

Figure 11: Reward Average UCB1

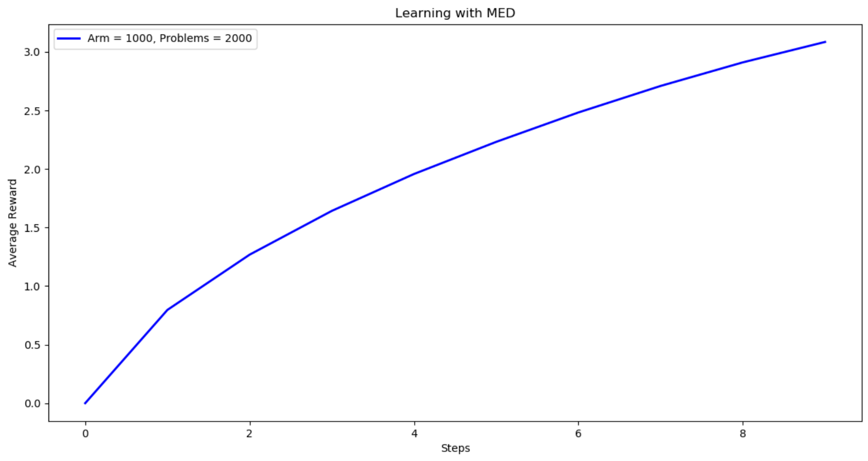

Figure 12: Reward Average MEA