Experiments on Multi-Armed Bandit Problems

Ajay Subramanian

May 2019

Definition of the Problem

The multi-armed bandit problem is a classic reinforcement learning example wherein pulling each arm of a slot machine returns us a stochastic reward $q_*(a)$, generated from an unknown probability distribution. Our objective is to identify the sequence of arms that achieves an optimality of some kind. This optimality condition can be defined in various ways, for instance, asymptotic correctness, regret, or PAC. There are some interesting observations that we can make about Bandit problems.

1. Bandit problems are a subset of immediate RL problems in which we receive an undelayed reward R_t for every action a_t . Also, R_t is a pure function of a_t and independent of any state variables.

2. A major challenge in solving such problems is handling the explore-exploit trade-off. More exploitation maximizes the obtained reward, but at the cost of identifying possible better arms. Excessive exploration can result in a more confident estimated reward $Q(a)$ but also increases the regret i.e. payoff lost out on while picking sub-optimal arms. I have experimented and analyzed the ϵ-Greedy, Softmax action selection, and UCB-1 algorithms on the 10-armed testbed described in [1]

Experiments

ϵ-Greedy

The ϵ-Greedy algorithm leaves the explore-exploit decision to a hyperparameter ϵ. Exploiting involves pulling the arm corresponding to the maximum value of $Q_t(a)$ while exploring involves pulling an arm, uniformly at random.

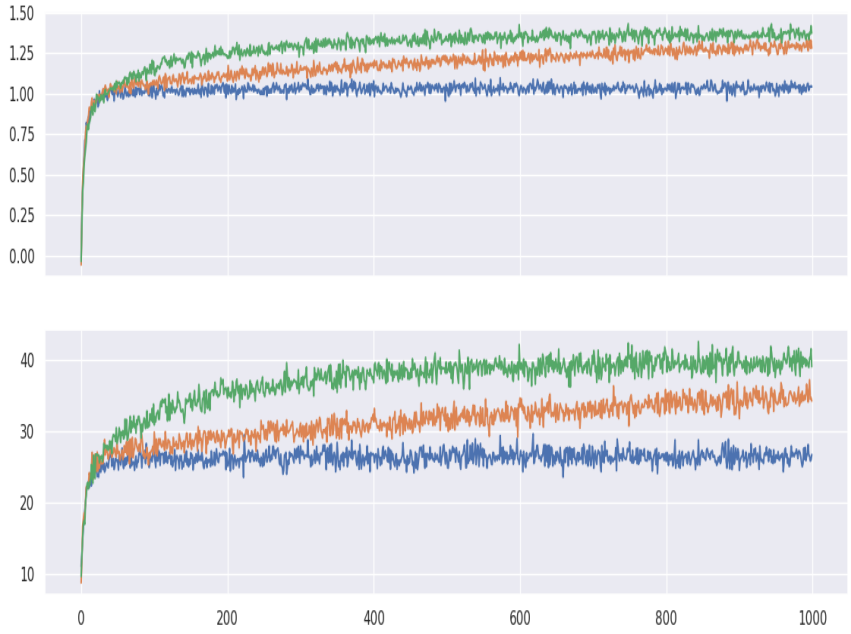

Figure 1: reward vs time (top) and % optimal action vs time (bottom) for ϵ-greedy for ϵ taking values 0 (blue), 0.01 (orange), 0.1 (green).

As seen in first plot of Figure 1, when the algorithm is set to pure exploitation i.e. $\epsilon = 0$, the reward vs time plot sharply increases on the first trial and then stays almost constant throughout. This is because any arm that gives us a positive reward will become the only positive valued arm and hence the 'optimal' one. We will then continue pulling only that arm till the end of the process. This process of learning is entirely random and hence is not very reliable. Now, if ϵ is increased to 0.01, we expect to take a random action 1% of

the time. Due to this, we see a sharp increase, followed by a gradual improvement in reward over time. The same trend holds for $\epsilon = 0.1$, but with the maximum possible reward being reached towards the final timesteps. This happens because, with a higher chance of exploration, it becomes easier to find the optimal arm. Hence, the average reward starts becoming more or less constant after timestep $\approx$750. Due to this 'reward saturation', we see the gap between the $\epsilon = 0.01$ and $\epsilon = 0.1$ curves decreasing towards the end.

This observation could also be explained from the perspective of explore-exploit tradeoffs. Whereas $\epsilon = 0.1$ curve identifies the optimal arm faster and continues to randomize its actions even after that, the $\epsilon = 0.01$ plot takes time to figure out the best arm, but almost always pulls that arm once it does.

The difference between these two ϵ values is all the more visible in the second plot of Figure 1 where we see the best arm being learned by the $\epsilon = 0.01$ model gradually over time while it is identified much earlier(at $\approx$400) by $\epsilon = 0.1$.

Softmax Action Selection

Using a softmax action selection makes the probability of pulling each arm proportional to its estimated reward $Q_t(a)$. This makes the exploration process more efficient and helps us to eliminate the clearly undesirable arms.

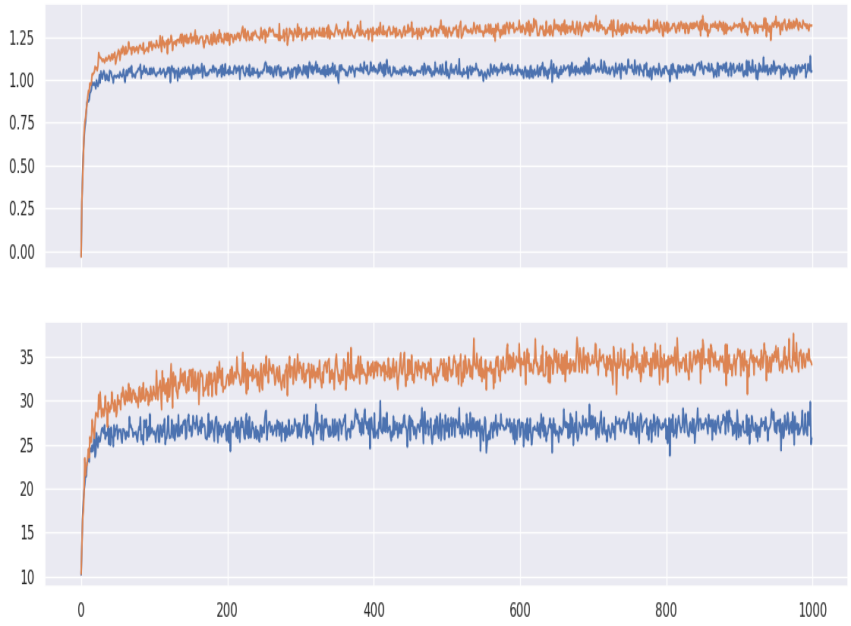

Figure 2: reward vs time (top) and % optimal action vs time (bottom) for softmax action selection for β taking values 0.01 (blue), 0.1 (orange)

When we compare the curves in Figure 2 with those in Figure 1, we can clearly make out that the maximization of reward occurs much earlier when we use softmax action selection. This is because of the more efficient action selection process. Arms that consistently return a lower reward will have a lower value function, and hence a lower probability of being selected. This is in contrast to the ϵ-greedy case where exploring meant giving equal chances to each sub-optimal arm irrespective of how consistently badly or well they had performed in the past.

We would also like to observe the variation of the plots when the value of β is changed, in this case for values 0.01 and 0.1. The most important difference is that a higher value of β correlates to a more uniform probability distribution, and therefore more randomness in selection, while a lower β value corresponds to a more biased distribution and hence more exploration.

Upper Confidence Bound (UCB-1)

While the two algorithms seen until now aim to achieve asymptotic correctness (choosing an arm that gives maximum payoff), the UCB-1 algorithm tries to, in addition, minimize regret i.e. pull least possible number of arms prior to finding the best one. Hence the algorithm goes as follows:

1. **Pull each arm once:** This helps us get an early reward estimate of each arm. Asymptotically, pulling each arm atleast once is a necessity to make a decision about the optimum one.

2. **Choose the arm j that maximizes** $Q(j) + \sqrt{\frac{2\ln(n)}{n_j}}$**:** UCB works on the principle of 'uncertain optimism'. This means that the algorithm hopes for an arm better than what is observed from previous data. This is clear in the above expression where the second term can be understood as an exploration confidence interval which shrinks as we pull more arms. This happens since as we pull more arms, we become more confident about the estimated reward values.

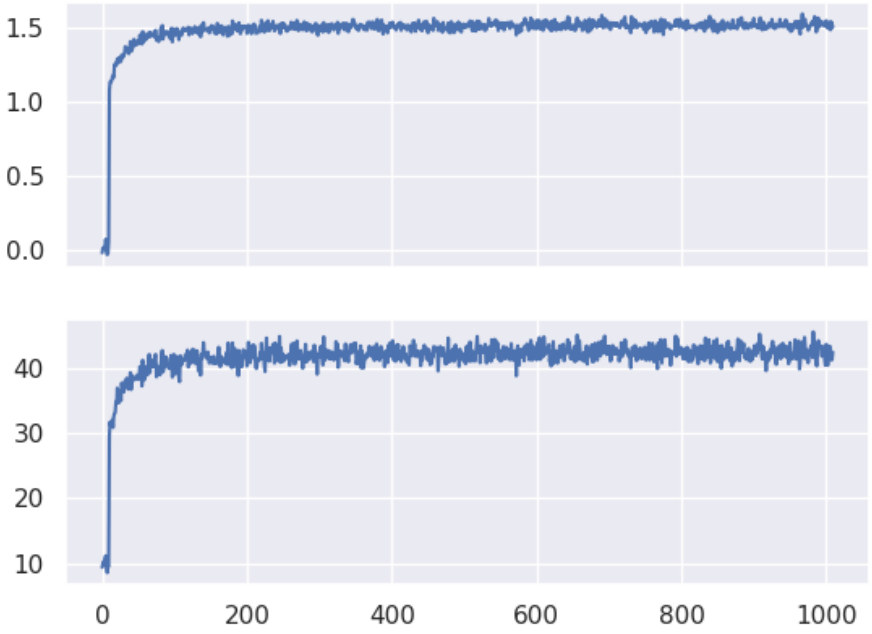

Figure 3: reward vs time (top) and % optimal action vs time (bottom) for UCB-1

It is easily observable from Figure 3 that UCB-1 achieves the near-optimum action the fastest amongst the three algorithms. An important reason for this is that UCB, unlike ϵ-greedy and softmax, combines the explore and exploit phases into a single operation. This is explained as follows. We have already seen the following action selection metric.

$$argmax_j \left(Q(j) + \sqrt{\frac{2\ln(n)}{n_j}} \right)$$

The first term of the expression accounts for exploitation. We know that, in order to maximize our reward and minimize the regret, we have to take an action with a reward somewhere in the vicinity of the known optimum. Next, at each step, we try to explore in the region by aiming for a slightly higher reward. The second term accounts for this exploration effort. Hence,

due to each step of UCB being productive, we can expect it to learn much faster than both ϵ-greedy and softmax, both of which need more time due to separate explore and exploit phases.

Effect of a Larger Action Space [1]

The major difference in these three algorithms is action selection i.e. the method of deciding and optimizing the explore-exploit decision. However, in the big picture, they work towards a common objective that is arriving at a good estimation for the true reward distribution. This distribution becomes harder to estimate, as the number of arms increases since the agent is left with more options per decision. Therefore all three algorithms share the characteristic that they take more timesteps to figure out the optimal action, as the number of arms increases.There are however, a few differences in the trend followed by each algorithm

ϵ-Greedy

As the number of arms increases, the trained bandits perform much better (on average) than the case with fewer arms, though they take longer to get trained. This could be because having more arms (with values picked from a common distribution) results in a lower probability of a few bad arms being picked. But in the few arms case, one bad arm could easily be picked since the sample size is much smaller.

Softmax

The effect discussed above is prevalent in softmax too. But since the softmax distribution biases us towards picking a better arm even in the few-arms case, the effect of more arms is not as pronounced.

UCB-1

The trend seen in the UCB case is quite different from the other algorithms. In the first 1000 (number of arms) timesteps, it averages a reward of 0 be-

[1] Graphs corresponding to this section can be accessed here

cause UCB initially picks each arm once. After that, it starts pulling arms with an optimistic reward expectation. But in a more-arms case, the algorithm behaves more aggressively because at initial timesteps, the $\frac{2\ln(n)}{n_j}$ value is smaller. Due to this, the increase in reward is slower but for more significant over a large number of timesteps. Another interesting observation in the UCB graph is that it has a higher optimal action percentage than the other algorithms. This could be because of the simultaneous exploration-exploitation by UCB i.e taking random actions from an estimated confidence interval rather from the whole action space.

REINFORCE on Contextual Bandits

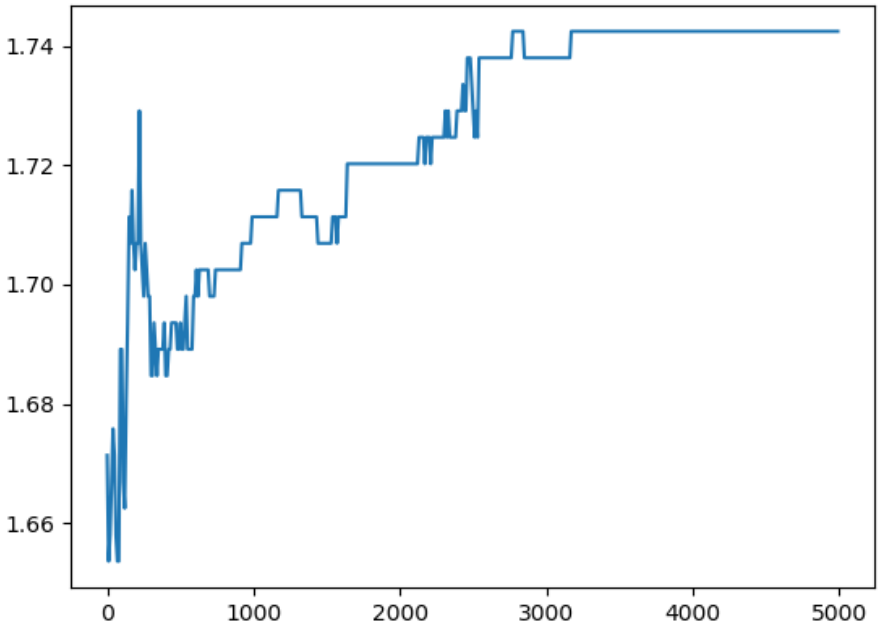

Figure 4: reward vs time for REINFORCE on a Contextual Bandit problem

A contextual bandit problem is a problem that is modelled using different bandits (different distributions) for different states. When encountered with

a particular state, we pick an action corresponding to our estimation of the expected reward of only that bandit (state). Contextual bandits are an acceptable formulation when we have states that are independent of each other i.e which is a more extreme assumption than the Markov assumption, and not always true in real world environments.
REINFORCE works by ascending the gradient of the estimated reward function. However, it may encounter local maxima or saddle points in its journey to the global maximum. Hence, the reward vs time graph as seen in Figure 4 may not always be a monotonically increasing graph (unlike the algorithms seen so far). The speed of convergence is determined by the step size. Too small a step size makes it slow while a very large one might result in an oscillation in the reward function.
An advantage of REINFORCE (or any policy gradient method) is that they can be modelled using a MLP, whence a sequence of linear operations with non-linear activation functions can result in complex function approximation capabilities.

Conclusion

From the detailed analyses of three perspectives at solving bandit problems, we have now a better understanding of the deep intuitions involved in their design. We have also analyzed the role of parameters in the functioning of these algorithms and have learnt the expected behavior of these models with varying parameter values.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, second ed., 2018.