‘Multi-arm bandit problem’

FOR

Dr. Basabdatta Sen Bhattacharya

(Department of Computer Science)

BY

Name of the Student	ID Number
Akshay V	***2018B1A70608G***
Aman Jain	***2018B3A70768G***
Anish Mulay	***2018B3A70907G***
Dev Churiwala	***2018B1A70602G***
Neel Bhandari	***2018B1A70084G***

In the partial fulfillment of the requirements of

CS F317 - Reinforcement Learning

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

14 Feb 2022

Overview

In this work, we analyze the multi-armed bandit problem. We understand its implementational details, explore different policies, validate the results discussed in the textbook, and provide a detailed study on the effects of different parameters on the given problem. We consider the 10-armed bandit for our analysis and start by creating a Bandit class as well as the initialization of its member variables.

Notable member variables include:

A. **k_arm**: Number of arms
B. **epsilon**: Probability for exploration in the epsilon-greedy algorithm (**ε**)
C. **initial**: Initial estimation for each action (used for optimistic initializations)
D. **step_size**: Constant step size for updating estimations (**α**)
E. **sample_averages**: If True, we use sample averages to update estimations instead of constant step_size (**1/N(A)**)
F. **UCB_param**: If not None, we use the UCB algorithm to select an action
G. **gradient**: If True, we use the gradient-based bandit algorithm
H. **gradient_baseline**: If True, we use average rewards as a baseline for the gradient-based bandit algorithm

The class has 3 methods of interest:

A. ***reset()*** -- Allows us to reset certain variables of the bandit object after each run (fixed number of timesteps).
B. ***act()*** -- Allows us to get an action for the bandit based on the specified policy.
C. ***step()*** -- Allows us to use the generated action to make a step in the environment, get the corresponding reward, and update the estimated value for that action.

We also define a ***simulate(runs, time, bandits)*** method, which is used to simulate the interaction between the bandit and the environment. Here, we can choose the “runs” and “time” parameters based on the analysis to be performed. This method returns 3 arrays which are later used for evaluating the agent and plotting their performance graphs.

Observations

Rewards Distribution

Our first plot represents a sample distribution of the rewards over the given actions. The action values have been chosen to be normal distributions with mean = 0 and variance = 1, and the actual rewards have been selected with mean = $q_*(a)$ and variance = 1 -- as observed in the plotted graph.

Greedy Agent

The first agent that we explore is a “greedy” agent which is exploitative in nature. It uses its estimated values to select an action that generates the maximum reward in the given state. We also plot an average best possible reward line which provides some context as to how the given agent is performing. The analysis of this greedy agent acts as a baseline for us to compare various other policies and their performance on the same problem.

ε - Greedy Agent

The next agent we explore is the “**ε**-greedy” agent which explores with an **ε** probability and exploits it’s estimated values for the other actions. We observe that the **ε**-greedy agent performs considerably better than it’s greedy counterpart due to it being able to explore more actions and their repercussions. It is clear from the optimal action ratio plot that the **ε**-greedy agent is able to select the optimal action nearly 80% of the time, whereas the greedy agent only selects the optimal action about 33% of the time.

Comparing **ε** Values

Now that it was clear to us that the **ε**-greedy agent performed substantially better than the greedy agent, we moved on the comparing **ε** values. We understood that the exact choice of **ε** values varied from problem-to-problem, but we could infer 2 things from our analysis. Firstly, choosing too low of an **ε** would lead to the agent being biased towards its exploitative nature -- thus often choosing suboptimal actions and not being able to generate high average rewards. Secondly, choosing too high of an **ε** would lead to the agent being biased towards its explorative nature -- thus spending too much time choosing suboptimal actions and in turn not being able to generate high average rewards.

Comparing Step-Size Values

For the above analyses, we used the number of action counts to decay the step size. We now evaluate the performance of constant step-size values. We observe that for the way that our problem is designed, a constant step-size of 0.5 is able to outperform the others. The plot with the corresponding optimal action ratios also paints a similar picture. We hypothesize that the other step-size values are either not able to modify their estimates fast enough, as in the case of small step-sizes or “1/N(A),” or the step-size values do not decay the estimate enough and are susceptible to frequent changes in the estimates -- both of which result in the rewards being inconsistent and low.

Optimistic Initial Values

We now explore optimistic initial values, due to the values being wildly optimistic -- we expect the agent to be more explorative at the beginning of the run. As the estimates tend towards the true values, the agent naturally becomes exploitative, thus generating higher average rewards in the longer run. We observe that the agent with an initial value, q, set to 0, initially performs better than the other agents due to their exploratory nature, but as time progresses, the agents with q set to 5, are able to consistently outperform the other agent. This can be credited to the fact that the agents initially performed worse but gained more knowledge about the expected values and were able to use this knowledge to outperform the other agent in the longer run.

Upper-Confidence-Bound Action Selection

Here, we attempt to explore the UCB Action Selection Policy. This includes taking into account their potential for being optimal, based on their estimate and the uncertainty in those estimates. As the number of times, an action is selected increases, this factor decreases -- which means that it is less likely to be selected unless its estimate trumps the others. Parallelly, if another action is selected then, this action becomes more likely to be selected in the coming runs. All this results in all actions being explored equally, and the actions with lower estimates being selected less over time. This can clearly be seen in the optimal action ration plot -- briefly, the other agent is able to outperform the UCB agent due to its exploration phase, but in the longer run, the UCB agent tends to consistently outperform the other agent, choosing the optimal action nearly 85% of the time.

Gradient Bandits

Intuition : The intuition behind Gradient Bandits algorithm is based on giving different preference to each action and the preference given to each action is updated based on the reward obtained . Initially, all preferences are the same so that all actions have an equal probability of being selected. The average of all the reward upto t serves as a baseline with which the reward is compared. If the reward is higher than the baseline, then the probability of taking that action in the future is increased, and if the reward is below baseline, then probability is decreased. The non-selected actions move in the opposite direction.

Implementation : We average reward is kept as the baseline which is updated after every action. One hot action array is created whose index value for the selected action is one and is zero for the rest of the actions . It respectively updates the formula for updating preference for the selected action and for the non selected actions as well. Thus the action with high preference is selected more often.

Epsilon Decay Strategy

Intuition: The intuition behind the epsilon decay strategy is very simple. Consider a new agent who is just starting to learn the environment and the rewards associated with various actions. It makes sense for the agent to explore more of these actions initially since it does not have a good understanding of how the environment works. After a certain number of iterations however, the agent should focus more on exploiting the knowledge it has gathered since its ultimate goal is to maximize reward.

Implementation: We start with an epsilon like we would with an epsilon-greedy strategy. We set the amount by which the epsilon will decay and the number of time intervals after which the decay will occur. Now, whenever the agent goes through the appropriate number of time intervals, the epsilon value is updated and the following decision the agent takes will use this new updated value of epsilon until the next decay occurs.

Results: On comparison with the greedy strategy, we found that the epsilon decay strategy fares better. The amount of decay and the time interval between decays does not affect the result. Epsilon decay always outperforms the greedy strategy. The epsilon decay strategy also outperforms the simple epsilon greedy strategy. Varying the rate of decay and time intervals between decays only changes the point of divergence between the two strategies but the epsilon decay strategy always outperforms the simple epsilon greedy strategy.

Future: Using the reward to determine epsilon decay is an alternative technique to epsilon decay that could be beneficial in some cases. The value is only lowered after an agent has passed a reward threshold. We wait for confirmation of the agent's learning before lowering the epsilon value, rather than presuming that the agent is learning more every episode. As a result, every time the agent's objective is set, we raise it and wait for the agent to attain the newer target before repeating the process. This strategy is sometimes called Reward Based Epsilon Decay.

Non-Stationary Problem

For this problem , we decided to change the reward distribution of the bandits after half the time step for all the runs , for example, if the the timestep is 1000 so we changed the reward distribution of the bandits after 500 timesteps and decided to see which algorithm works well. We see that using step-size parameter(α) rather than using the step size 1/n gives better performance as In such cases it makes sense to give more weight to recent rewards than
to long-past rewards.
Another fascinating aspect we see from the results is that gradient bandits are not able to recover well after we change the reward distribution as the algorithm learns a preference which causes it to select the higher preference actions more frequently. Hence after we change the reward distribution it takes time for it to learn the preference again so it performs poorly in the initial time steps after the change.

Hyperparameter Testing:

In this section, we are an attempt to do hyperparameter testing for our multi-armed bandit problem to explore different situations in the epsilon greedy context and constraints to take a look at any inferences that can be made.

For our purposes, we assume an agent in a stationary environment and update the estimation for each arm using sample averages.

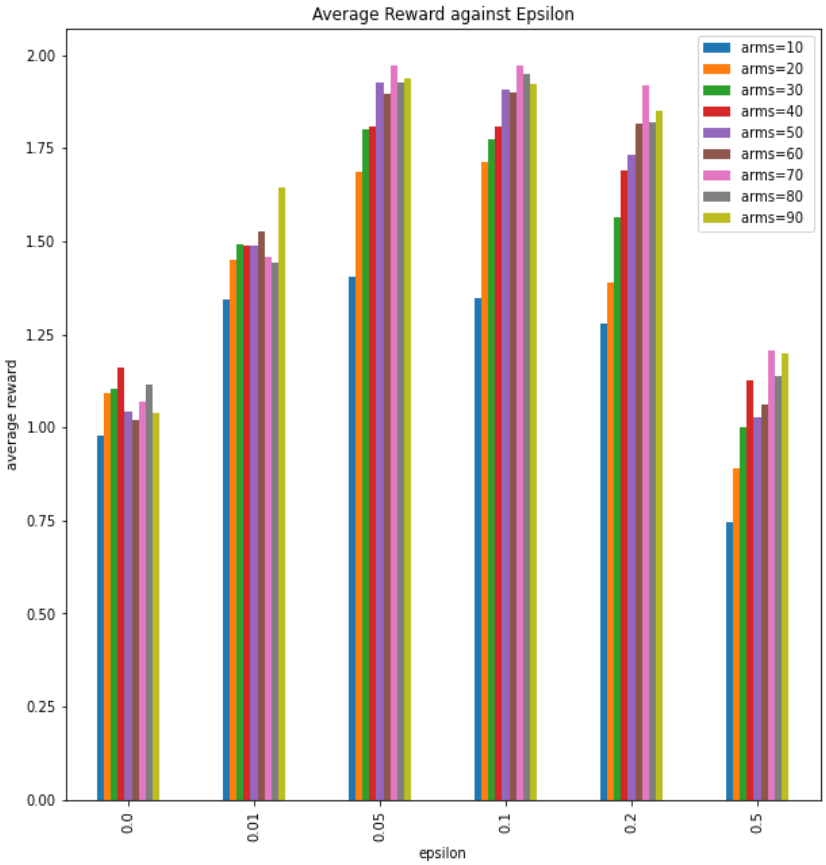
Variables that are taken into consideration:

- Epsilon (ε)
- Arms (k arms)
- Time (run time)

We will view the results of our bivariate hyperparameter testing in 3 steps:

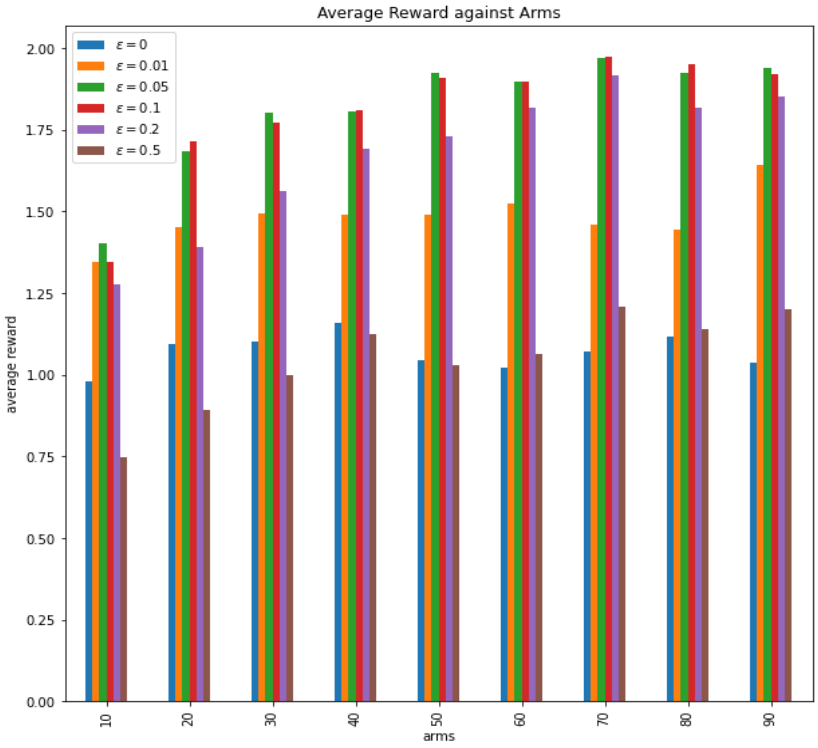
Varying epsilon values and arm size (Time Constant)

Keeping time constant and doing a bivariate analysis over the 2 parameters epsilon (ε) and k-arms, we observed the following graphical distributions.



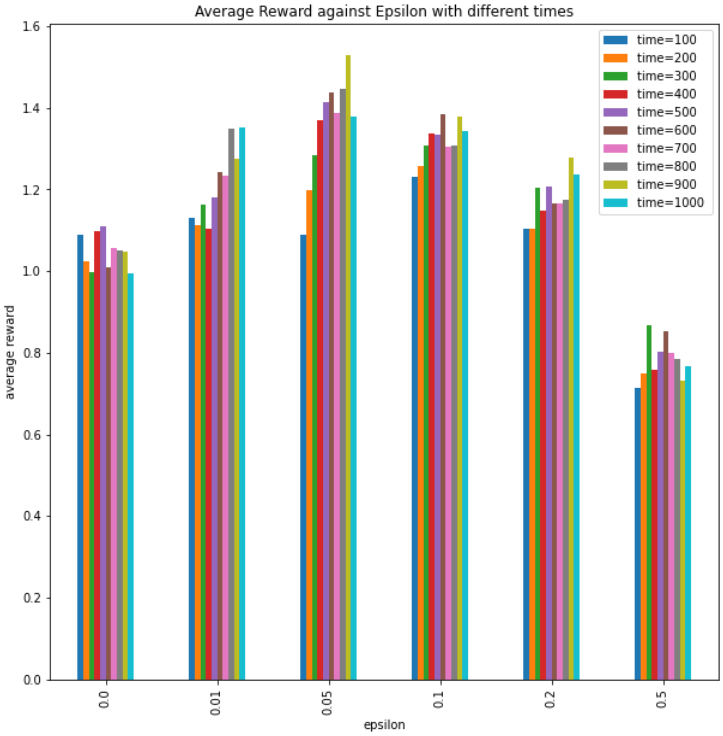
It is important to note that changing the number of arms changes our problem entirely. It can be viewed as changing the number of slots in a casino in the famous analogy. However, we can draw certain conclusions keeping the above factor in mind.

We can see that irrespective of the number of arms, very low (purely greedy) or very high (highly explorative) ε values correspond to a lower average reward. This is a trend we have observed in the sections above.

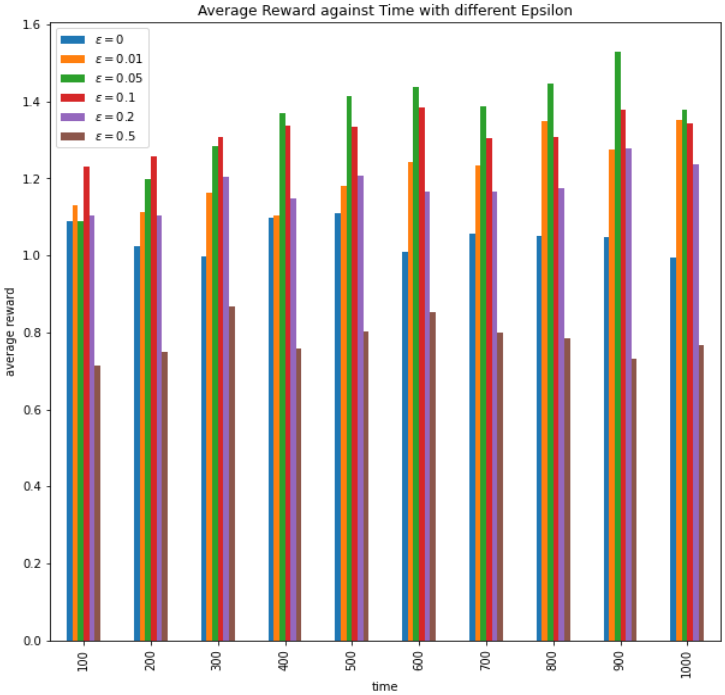


In the above graphical analysis, we can clearly see that increasing the number of slots in a casino will lead to our algorithm performing better till the performance reaches a maxima (approx. 50 arms) and then stagnate. Here we can clearly see that ε=0.05 and ε=0.1 perform the best for our given RL problem.

Varying epsilon values and time duration (Arms Constant)

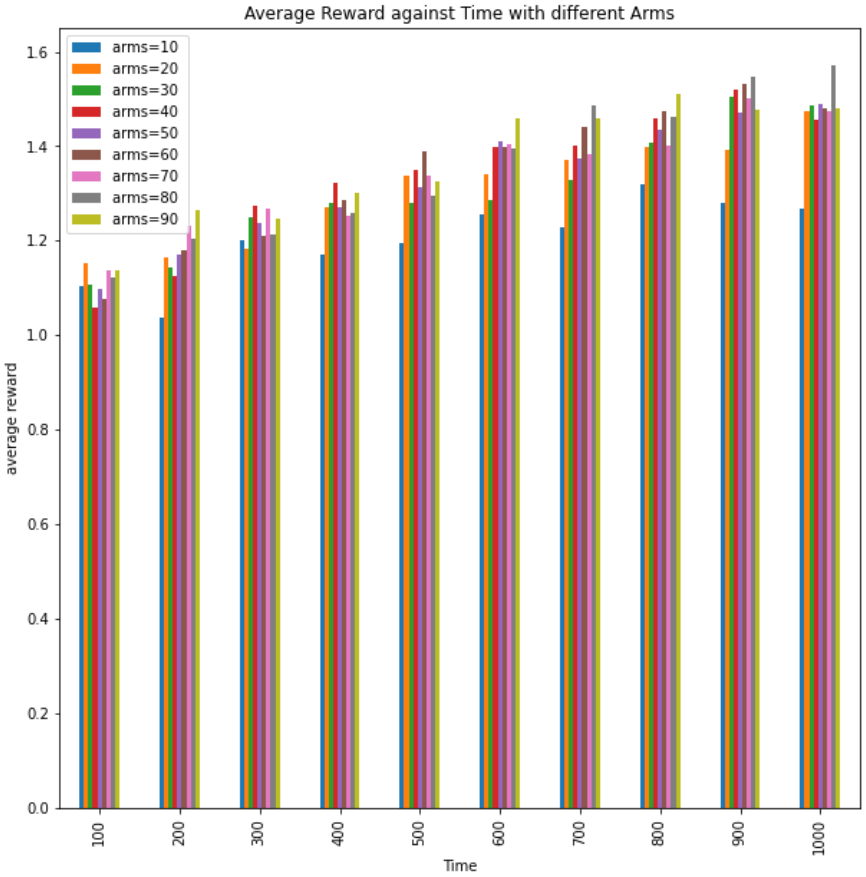


A few conclusions that we could draw from the bivariate analysis between the ε and time variables are:
The performance of the highly explorative (ε=0.5) or highly exploitative (ε=0.0) agents seems to be independent of time and does not increase with time as seen for other epsilon values. We can hypothesize that the highly exploitative agent does not gather more information as time goes on and hence irrespective of the time is stuck with the same priority for actions as initial time.



One interesting observation we can note here is that for time=100, the performance of the greedy agent (ε=0) is comparable to the other agents. This can be attributed to the fact that the other agents have not had enough time to learn and gather information about the optimal choices. Therefore it may make sense to choose agents with low ε values in time-crunch situations.

Varying the number of arms and time duration (Epsilon Constant)



Looking at the bivariate analysis between time and arms we can see that irrespective of the number of arms (ε=0.01), we can see a significant improvement in performance (increased average reward) as time increases. This can be explained as the agent is learning to make more optimal choices as time goes by and its view of the distribution reaches ever closer to the real distribution.