ME 493: Project α

Shane Lawson

February 16, 2017

1 Introduction

In this project an action-value learner was created to solve the multi armed bandit problem. The agent in this case had to identify the arm with the greatest probability of delivering the highest payout, in essence minimizing losses, or maximizing payout, depending on how the multi armed bandit was configured. As a first introduction to reinforcement learning, this project was intriguing, and turned out to be very rewarding[1]. This report will go over the structure of the program, along with the important, relevant functions, and end with results from running multiple tests using the action-value learner to solve the multi armed bandit problem.

2 Specifics

2.1 Program

In addition to the main program file, I made three classes for this project. An `Arm` class, a `MAB` (multi armed bandit) class, and an `Agent` class.

2.1.1 Arm

The `Arm` class is simple, and as one might expect, represents an arm of the multi armed bandit. This class contains two `double` values, the mean, and the standard deviation of the arm. This class contains a constructor which creates an arm with a random (0,100] mean, and random (0,25] standard deviation. The `Arm` class also contains a function that creates a normal distribution of values using the Box-Muller transformation, and a function that returns the mean of the arm[1].

2.1.2 MAB

The `MAB` class contains a vector of arms. It has a constructor that creates a multi armed bandit with a specified number of arms, a function that pulls a specific arm and returns the reward value, and functions that return the number of arms and a specified arm's mean.

[1]Puns!

2.1.3 Agent

The `Agent` class is where the important functionality of the program exists. It contains the variables α and ϵ, a vector that contains the calculated arm values, and a pointer to the multi armed bandit the agent will interact with. In addition to the constructor which creates and intializes the agent, this class contains many functions:

- `getMaxArm()` : returns the arm with the highest perceived value.
- `executeCycle()` : this function executes a single iteration (or multiple iterations) of the sense, decide, act, react cycle using the functions of the same name. Sense is not used in this application.
  - `int decide()` : This function determines which arm to pull based on which has the highest value, or random, based on the ϵ value.
  - `double act(int)` : This function pulls the arm and gets the reward. The reward then gets passed into the `react()` function.
  - `void react(int,double)` : This function updates the value of the arm using the equation containing α.
- `runTest()` : Depending on the number of arms the multi armed bandit has, this function calls either `testA()` (the test for single arms) or `testB()` (the test for multiple arms).
  - `testA()` : This function runs the test for single arms which tests that the average from many pulls converges to the mean value of that arm. The test starts with a loop where the arm is pulled 1,000,000 times and every reward that is generated is summed up inside the loop. Once the loop ends, the sum is divided by the number of pulls (in this case 1,000,000) which yields the average reward of 1,000,000 pulls. This average won't exactly match the mean of the arm, so a threshold around the mean is created with an upper limit and a lower limit, and the average is compared to this range. An `assert` is then used to make sure the found average is within the threshold (in this case $\pm 1\%$) of the arm's actual mean.
  - `testB()` : This function runs the test for multiple arms which ensures that the agent will converge to "clearly" the best arm if given enough pulls. The simplest way to test for the "best" arm was determined to be a comparison of the means of the arms, and the highest mean would be the "best". There are issues with this assumption, namely if the means are close, but have very different standard deviations. In this case, when the agent receives a couple extreme rewards at the end of its runs this has the potential to influence the values enough to make a decision where an arm without the highest mean may be *perceived* to be more beneficial. This has been know to cause this test to fail, though rarely.

Once the agent has executed all the cycles (in most cases 10,000), `testB()` is run. The test starts with a simple loop that loops through all the means of the multi armed bandit's arms to identify the maximum. Once the max is found, an `assert` is used to check that the arm with the max mean is the same as the agent's max value arm.

- `reset()` : This function resets the agent to its starting state so that it can be run multiple times on the same multi armed bandit. This simply resets all arm values to 0.
- `setEpsilon(double)` : This function sets the value of ϵ, allowing for the value to be changed. `setAlpha(double)` accomplishes the same thing, but for α, however it is not used in this program.

2.1.4 Main

The main program consists of a single function which prompts the user for a number of arms for the multi armed bandit and makes sure the user inputs a value that is numeric and positive, changing the prompt to ask for a positive integer if it is not. Once this number is captured, a multi armed bandit is declared with the specified number of arms. Next, and agent is declared with the address of the multi armed bandit so that it can interact with it later. A loop is executed 30 times for the statistical runs. Inside this loop, the `Agent`'s `executeCycle(int)` function is called with 10,000 iterations. The `runTest()` function is called to ensure the agent have been successful, and then the agent is reset to start the next statistical iteration.

3 Results

The action-value learner was used to solve the multi armed bandit problem multiple times with varying scenarios to ensure that it was performing as expected. The majority of these results are omitted here since they're not very useful.

The first interesting result comes from testing the action value learner against a three armed bandit. The outcomes of 30 statistical runs of this scenario were averaged and the results are shown in Fig. 1.

This figure is very cool because it's clear that the agent begins by pulling arms with low payouts, but after a while, the random exploration pulls begin to produce higher values at which point the agent switches to better paying arms, until it converges to the arm with the highest mean. This convergence happens in under 800 pulls.

Figure 2 shows the action curve for this same scenario with the three armed bandit. The action curve shows the % likelihood that an arm will get picked. Initially the agent picks the poor paying blue arm, which early on has a higher value than the others. As a result, there is an almost 90% likelihood that the bad arm will be chosen. Once the exploration pulls

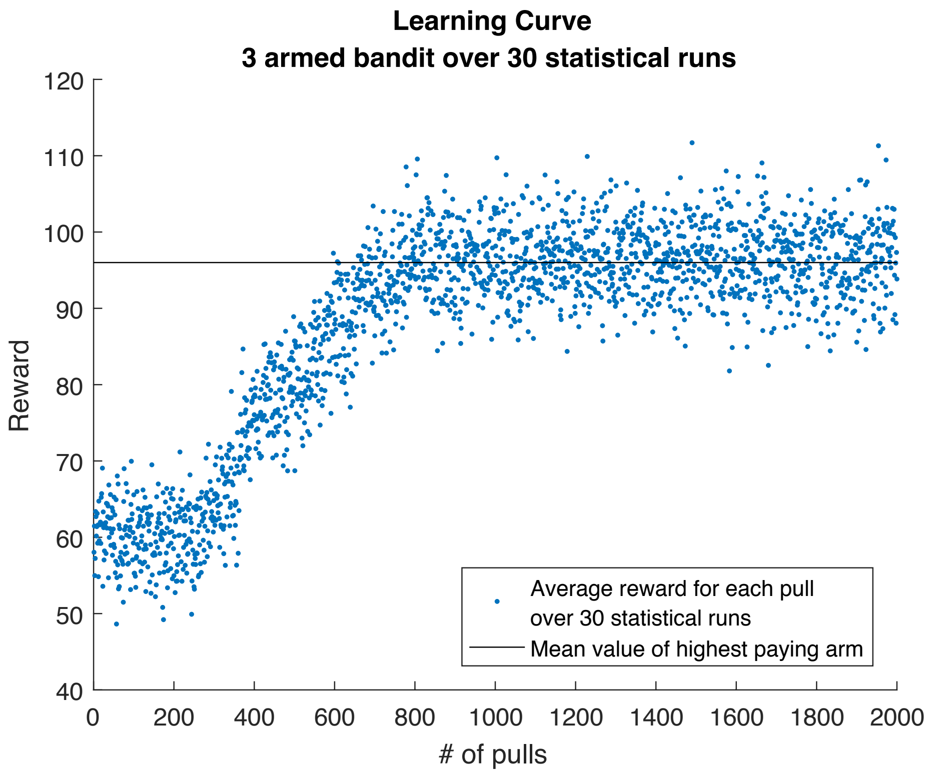

Figure 1: Learning curve of agent with a 3 armed bandit over 30 statistical runs.

factor in and the agent finds the best arm, the bad arm drops to an almost 0% likelihood, while the yellow arm flies up to almost 100%.

Lastly, as an exploration in the roles certain parameters play in the agent's performance, one parameter was chosen and varied over a series of runs. The parameter chosen in this case was ϵ. The three armed bandit scenario was run for 30 iterations for each value of ϵ chosen. The initial ϵ value was 0.01, and was increased by 0.05 every 30 runs, resulting in values of 0.01, 0.06, 0.11, 0.16, and 0.21. The performance of the agent for each of these values can be seen in Fig. 3.

The value of 0.01 very slowly converges to the best arm in over 3000 pulls, where all the others converge at or before 500 pulls. Since the value of ϵ is correlated with how often the agent takes a random action, this behavior makes perfect sense. The blue line is only taking a random action about every 100 pulls, requiring around 30 random pulls to find the best arm. The green line also needs about 30 random pulls, but on the other hand takes a random action about 1 in 5 pulls, meaning it hits the 30 random pulls to find the best arm at only 150 pulls overall. The values other than these two extremes fall in succession in between the two, as expected.

Another interesting aspect about these results is that looking closely, the blue line has a smaller amplitude in its fluctuations than the green line. This also makes sense since it takes a random action so infrequently, that when averaged across 30 trials, its nearly unnoticeable and all the fluctuations result from the normal distribution. The green line has a greater

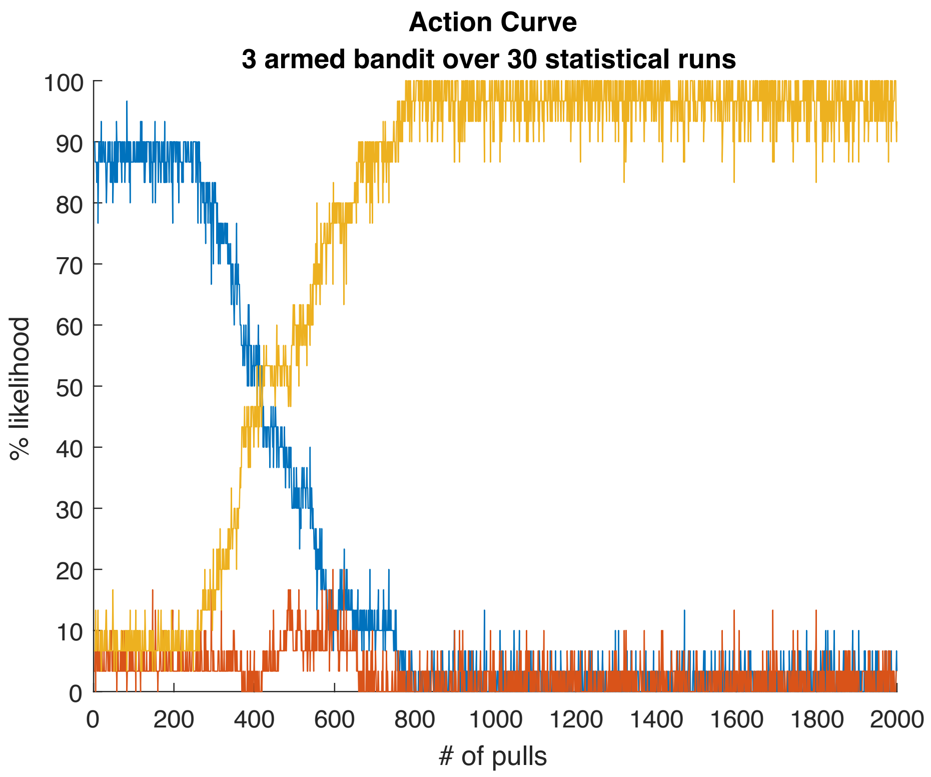

Figure 2: Action curve of agent with a 3 armed bandit over 30 statistical runs.

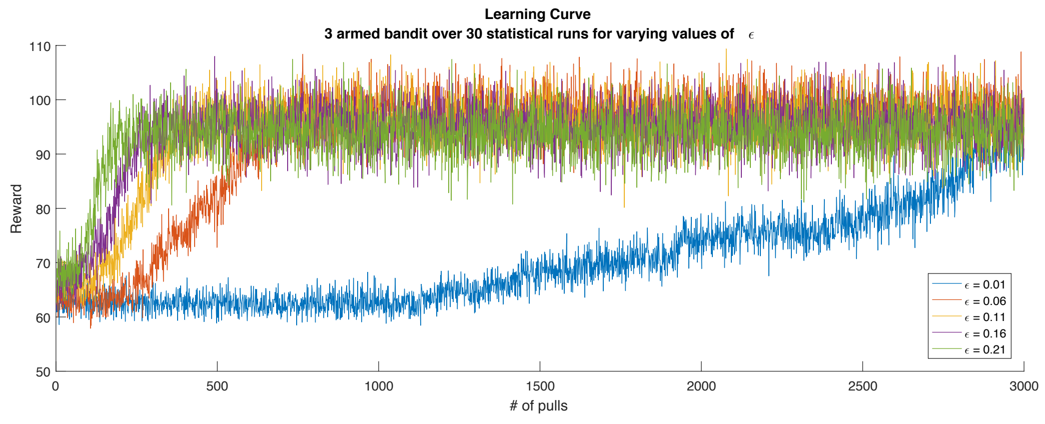

Figure 3: Learning curve of agent with a 3 armed bandit over 30 statistical runs for varying values of ϵ.

amplitude because when it takes a random action every 5 pulls, it is getting a lot of bad pulls which are affecting the overall returns resulting in a much larger distribution of payouts in addition to the normal distribution that the best arm gives.

A better solution for this problem when varying values of ϵ would be to start with a high value, so that it converges quickly ($<$ 300 pulls), then decrease ϵ, so that the best arm's reward is not being contaminated by the bad exploration pulls.

4 Conclusion

This report begins with an explanation of how my code works, namely how I implemented the `testA()` and `textB()` functions. The user is prompted for the number of arms to create for the multi armed bandit when the program is run, allowing for a simple and effortless means of changing the value. As demonstrated in this report, and confirmed by running the program, it is apparent that I successfully completed the entire project creating a multi armed bandit, an action-value learner to learn the best paying arm, and running and testing it over 30 statistical runs.

References

[1] "Box-muller transform." `https://en.wikipedia.org/wiki/Box-Muller_transform`. Accessed: 2017-02-08.