MULTI-ARMED BANDIT

BY

Kavyaa Somasundaram (**mrc17ksm**)
Vishnu Hari (**mrc17vhi**)

Introduction:

This lab work deals with improving / implementing a multi-armed bandit using epsilon greedy algorithm. We were provided with a sample code snippet and the **goal is to improve the performance of the bandit such that in 20 simulation runs, the implemented bandit should reliably beat the reference bandit by 5% (atleast 15 simulation runs).** Epsilon greedy is a technique which is used as a balancing approach towards the exploration and exploitation trade-off. In our algorithm, we take the epsilon value to be 0.1.

Implementation:

The code can be divided into three phases, initialization of the states, choosing the optimal arm (run function) and updating the rewards for the agent (get_feedback function). We tried the implementation in three different ways,

- Without learning rate and by updating the payouts ("expected_values") along with implementing the epsilon decay values.
- With including and changing the learning rates and epsilon decay values.
- We tried with implementing uniform and random distribution of time samples from the arms.

The core idea in our implementation of e-greedy algorithm is,

1. **Initialization:** We introduced two variables, "learning rate" and "arm_index". We introduced an *if* condition such that, it initializes the values for the "expected_values" randomly for each trial and returns "expected_values" variable that is been updated from the "get_feedback" function.
2. **Choosing the optimal arm (run function):** We first randomly choose the arms for each trial and then estimate the "expected_values" for each arm. Then we look for the arm with the maximum "expected value" and exploit it. Thus we return the arm with the maximum "expected_value".
3. **Updating the probability of "expected_value" for each arm (get_feedback):** The rewards are generated in the "simulate" file and then passed into this function. The first thing to calculate is the total rewards (sum), which is calculated for each arm with the help of arm_index. Then the frequencies (which is initialized to 0 with the length of the arms) is updated for each arm. Then the "expected_values" is updated by dividing the total reward by the frequencies for each arm. The last step is implementing the epsilon decay (1-e). We tried three different epsilon decay values such as 0.1, 0.95 and 0.5 to analyse the performance variation of the bandits. This epsilon decay is used as a technique for exploration and exploitation trade-off.
4. **Another implementation** we tried was to include the learning rate in the updation of the "expected_values" by adding the difference between the current and previous estimation values multiplied by a learning rate that is fixed.

Results & Discussion:

We were **successfully able to beat the reference bandit reliably for 15 and more simulation runs**. We tried with different learning rates and epsilon decay and with both normal and uniform distribution of time samples for the arms. Out of which, when we used uniform distribution of the time samples and learning rates 0.3 the performance of the bandits was not good enough. When we were continuously rerunning the test for multiple times (20 trials), the pass rate we got for the different scenarios we tried / experimented with are listed below: (all these results are by using random distribution of the time samples)

1. Epsilon decay = 0.5, learning rate=0.1, epsilon = 0.1: pytest- 18/20 passed, obfuscated test-5/20 passed
2. Epsilon decay = 0.1, learning rate =0.1, epsilon = 0.1: pytest-19/20 passed, obfuscated test-5/20 passed
3. Epsilon decay = 0.95, learning rate = 0.1, epsilon = 0.1: pytest- 19/20 passed, obfuscated test- 6/20 passed
4. Epsilon decay = 0.95, no learning rate, epsilon = 0.1: pytest – 17/20 passed, obfuscated test – 10/20 passed

Though we get a good pass rate in the pytest, we are not sure as to why we get a lower pass percentage in the obfuscated test. From results (3) and (4), we observe that without learning rates, the pytest pass percentage is almost similar, but the obfuscated test percentage increases. From the results above, we can observe that epsilon decay of 0.95 gives optimal results as it exploits 95% of the time and explores only 5% of the time. The performance can be improved by increasing

the number of simulation runs, as the outcome can change over time. Another way to improve is to use Thompson sampler or by using the upper and lower confidence bounds in selecting the best arms.
Link to Github pull request: https://github.com/TimKam/multi-armed-bandit-lab/pull/39

References:
[1] https://towardsdatascience.com/reinforcement-learning-multi-arm-bandit-implementation-5399ef67b24b
[2]https://towardsdatascience.com/solving-multiarmed-bandits-a-comparison-of-epsilon-greedy-and-thompson-sampling-d97167ca9a50