Reinforcement Learning Cookbook: A Graduate Student's Guide

Muhammad Kamran Janjua
mjanjua@ualberta.ca

University of Alberta
RLAI Lab
Edmonton, Alberta

Contents

1 Disclaimer

These are personal study notes of Reinforcement Learning (RL) to familiarize myself with the field in hopes of doing academic research on the topic. These notes serve as a summary of a lot of fine academic text available on the topic. These are not referenced properly because I use it only for academic study. Please note that these notes do not serve as an academic textbook, but I hope that someone might find them useful for a comprehensive study on the topic. However, following are some of the major sources these notes are complied from.

1. Rich and Barto's Book
2. Prof. Philip's Lectures @ UMass, Amherst
3. RL Specialization by Dr. White(s)
4. Dr. Silver's Lectures

2 Introduction

Definition 1 *Reinforcement Learning (RL) is an area of machine learning concerned with how an agent can learn from interactions with the environment.*

There are certain main components of an RL framework: ***Agent, Environment, Feedback.*** The Agent is the main entity that takes an action based on some state of the environment and the feedback from the environment tells the agent if the action was good enough in terms of some reward. The agent is also known as the learner and/or the decision maker. Another important thing to note is that RL environments are not like datasets in supervised learning models i.e. the data is fixed. In RL, the description of the environment is provided to the agent since the environment is likely to change based on an action taken by the agent.

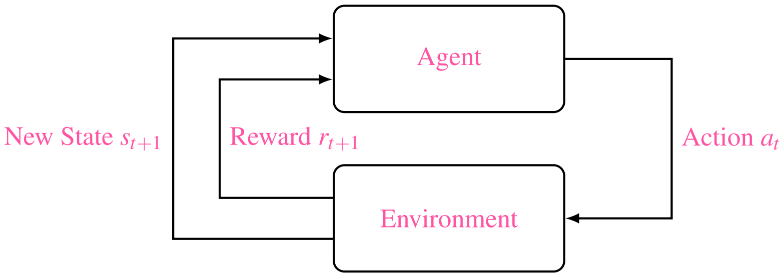

Figure 1: The standard age-old agent-environment interaction diagram. This is re-produced from Rich and Barto's Book (referenced above) and the code is taken from here.

The Figure 1 represents an entire RL system. An agent, the learning, takes a step in the environment by choosing an action a_t and gets the next state s_{t+1} and the reward r_{t+1} which indicates how *good* the action was. This reward signal (or the feedback from the environment) initiates the learning cycle. The agent then samples the states from the environment, takes an action, obtains some reward and adapts to the task at hand. This cycle expresses how an agent can learn from interactions with the environment and the action does not affect the next state.

3 Bandits

In a full RL problem, the actions taken by an agent affect the next state and the reward. This is complicated in a sense that the problem becomes associative i.e. different actions depend on the situations and the agent has to learn to act in more than one of these situations. A non-associative bandit problem is simplified because the agent is repeatedly faced with same choice and the goal is to select the action that yields largest payout.

3.1 k-armed Bandit Problem

In a k-armed Bandit problem, an agent is faced repeatedly with a choice among k different options.

Definition 2 *An agent chooses between k actions and receives a reward based on the action it chooses.*

After each choice the agent makes, it receives an reward based on the action it had taken. The reward is sampled from a stationary probability distribution. The goal of the agent is to maximize the expected reward over a certain time period (time steps). The k-armed bandit problem is analogous to a slot-machine (one-armed bandit). However the only difference is that in k-armed bandit, there are k-levers instead of one. The eventual target is to focus on the *best* levers, generating maximum reward. In the k-armed bandit problem, each of the k actions has an expected or mean reward given that action is selected. This is known as the value of the action. To mathematically formulate this, let's say that the action A_t is selected at timestep t and the reward corresponding to the action is R_t. The value of an arbitrary action a, denoted by $q_*(a)$ is the expected reward when a is selected.

$$\boxed{q_*(a) := E[R_t \mid A_t = a] \forall_a \in \{1,..,k\}} \tag{3.1}$$

It is important to note that if $q_*(a)$ was known, then solving k armed bandit would be simple since the agent would always choose the action with highest associated value. However since this is not the case, we estimate the $q_*(a)$ and denote the estimated value of action a at timestep t as $Q_t(a)$. Ideally we would want $Q_t(a)$ to be as close as possible to $q_*(a)$. Since we maintain the estimated action value, then at any timestep t, there is always an action with highest value. These actions with highest estimated values are known as greedy actions. When an agent selects a greedy action, we can say that the agent is expoloiting the current knowledge it has gathered. However if instead the agent selects one of the other non-greedy actions, then we say that it is exploring since that would help improve the estimate of non-greedy actions. Exploitation is better when the agent wants to maximize the expected reward on one step therefore we can say that exploitation is for short-term benefit. On the other hand, exploration may produce greater total reward in the long run.

Example 1 *Consider that you visit a restaurant to enjoy a good meal. You know that you have always enjoyed meal A. Therefore you are inclined to order the same meal A, but you also see that the restaurant has other meals on the menu which look very tempting. If you go with meal A, that would be exploitation since you would be using your current knowledge. However, if you decide to order meal B, that would be exploration, since in this case you can update your current knowledge.*

3.2 Action-Value Selection

Recall: The value of an action is the expected reward when that action is taken. Since we do not know $q_*(a)$, we use the estimate $Q_t(a)$ instead.
One of the most basic methods of estimating the value of an action is the Sample Average Method (SAM). In this method, the rewards received are averaged. Therefore we can formulate SAM as $Q_t(a) := \frac{\text{sum of rewards when action } a \text{ taken prior to } t}{\text{number of times action } a \text{ taken prior to } t}$ and is mathematically written as follows.

$$\boxed{Q_t(a) := \frac{\sum_{i=1}^{t-1} R_i}{t-1}} \tag{3.2}$$

As the denominator approaches infinity ($t-1 \to \infty$), by the law of large numbers, $Q_t(a)$ converges to $q_*(a)$. The greedy-action, the action with the maximum estimated value can be defined as follows.

$$\boxed{a_g := \text{argmax}_a Q_t(a)} \tag{3.3}$$

where argmax_a denotes the action a for which the expression that follows is maximized (the ties are broken randomly). Alternatively the agent might choose to explore instead by sacrificing the immediate reward (the greedy approach) in order to gain more information on other actions. It is important to note that the agent can not do exploration and exploitation simultaneously. This introduces a major dilemma where the question of choosing between whether to explore or to exploit arises. This is known as the exploration and the exploitation dilemma.

3.3 Estimating Action-Values Incrementally

Given the SAM method, it is also important to estimate the action-values in a computational and memory efficient manner. We can perform the updation in a constant per-time-step memory and computation by incremental updation. Let R_i be the reward received after the i-th selection of an action a, and let Q_n denote the estimate of its action value after it has been selected $n-1$ times. We can simply write this as follows.

$$Q_n := \frac{R_1 + R_2 + \ldots + R_{n-1}}{n-1}$$

We can use this to re-write SAM recursively to derive the updation rule.

$$\begin{aligned}
Q_{n+1} &= \frac{1}{n}\sum_{i=1}^{n} R_i \\
&= \frac{1}{n}\left(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i\right) \\
Q_n &= \frac{1}{n-1}\sum_{i=1}^{n-1} R_i \\
Q_{n+1} &= \frac{1}{n}\left(R_n + (n-1)Q_n\right) \\
&= \frac{1}{n}\left(R_n + nQ_n - Q_n\right)
\end{aligned}$$

Therefore we can write the update rule as follows.

$$\boxed{Q_{n+1} = Q_n + \frac{1}{n}\left[R_n - Q_n\right]} \tag{3.4}$$

The update rule can be generally defined as follows.

$$\boxed{\text{NewEstimate} \leftarrow \text{OldEstimate} + \text{StepSize}\left[\text{Target} - \text{OldEstimate}\right]} \tag{3.5}$$

The expression $[\text{Target} - \text{OldEstimate}]$ is an error in the estimate and it is reduced by taking a step towards the Target. In the Equation (2.4), the StepSize parameter changes from timestep to timestep. In processing the nth reward for action a, the method uses $\frac{1}{n}$ as the step-size parameter. However to simplify, the step-size parameter is denoted by $\alpha_t(a)$.

Algorithm 1: Complete Bandit Pseudocode

```
Initialize, for $a = 1$ to $k$:
$Q(a) \leftarrow 0$
$N(a) \leftarrow 0$
Loop Forever
    $A \leftarrow \begin{cases} \text{argmax}_a Q(a) & \text{with probability } 1-\varepsilon \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$
    $R \leftarrow$ bandit(A)  # takes an action $A$ and returns a reward $R$
    $N(A) \leftarrow N(A) + 1$
    $Q(A) \leftarrow Q(A) + \frac{1}{N(A)}\left[R - Q(A)\right]$
```

3.4 Nonstationary Bandit Problem

In stationary bandit problems, the reward distribution is stationary over time i.e. the reward probability does not change over time. However often RL problems are not stationary and therefore it would make sense to give more weight to recent rewards as compared to long-past rewards. This can be achieved with a constant step-size parameter. Let the constant step-size paramter be denoted by α, then the update equation can be written as follows.

$$\boxed{Q_{n+1} = Q_n + \alpha\left[R_n - Q_n\right]} \tag{3.6}$$

where α is the step-size parameter $\in (0,1]$ and is constant. This results in decaying past rewards over time i.e. Q_{n+1} being a weighted average of past rewards and the initial estimate Q_1.

$$\begin{aligned} Q_{n+1} &= Q_n + \alpha\left[R_n - Q_n\right] \\ &= Q_n + \alpha R_n - \alpha Q_n \\ &= \alpha R_n + (1-\alpha)Q_n \\ &= \alpha R_n + (1-\alpha)\left[\alpha R_{n-1} + (1-\alpha)Q_{n-1}\right] \\ &= \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2 Q_{n-1} \end{aligned}$$

This shows the relationship between R_n and R_{n-1} and we can unroll this further to get to Q_1.

$$\begin{aligned} Q_{n+1} = \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2 Q_{n-1} + \\ \ldots + (1-\alpha)^{n-1}\alpha R_1 + (1-\alpha)^n Q_1 \end{aligned}$$

$$\boxed{Q_{n+1} = (1-\alpha)Q_1 + \textstyle\sum_{i=1}^{n} \alpha(1-\alpha)^{n-1} R_i} \tag{3.7}$$

where $(1-\alpha)Q_1$ is the initial value of Q and $\sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i$ is the weighted sum of rewards over time. The weight $\alpha(1-\alpha)^{n-i}$ assigned to R_i depends on how many rewards ago, $n-i$, it was observed. The quantity $1-\alpha$ is less than 1, thus the weight given to R_i decreases as the number of intervening rewards increases.

3.5 Exploration and Exploitation

As noted earlier, the question of whether to choose exploration or exploitation introduces a dilemma since the agent cannot choose to do both simultaneously. One very simple solution is to randomly choose between exploration or exploitation. However that is not optimal since we can not control the number of times either of them is being chosen. We would ideally want to exploit most of the times with a little chance of exploring.

3.5.1 ε-Greedy Action Selection

ε-Greedy solves the problem by choosing exploitation most of the times with a little chance of choosing to explore. The ε is the probability of choosing and the exploitation probability is $1-\varepsilon$ whereas the exploration probability is ε. The action selection based on ε-Greedy is done as follows.

$$\boxed{A_t \leftarrow \begin{cases} \text{argmax}_a Q_t(a) & \text{with probability } 1-\varepsilon \text{ (exploit)} \\ a \sim \text{Uniform}(a_1,..,a_k) & \text{with probability } \varepsilon \text{ (explore)} \end{cases}} \tag{3.8}$$

where $a \sim \text{Uniform}(a_1,..,a_k)$ is a random action.

Example 2 *Consider the example of rolling the dice* ⚀ ⚁ ⚂ ⚃ ⚄ ⚅
If the it falls on one, we explore otherwise we exploit. In such a case, the ε would be $\frac{1}{6}$ i.e. agent would explore $\frac{1}{6}$% of the times and would exploit $(1-\frac{1}{6})$% of the times.

3.5.2 Optimistic Initial Values

The Optimistic Initial Values method provides a better balance between the exploitation and exploration. The method encourages exploitation early on in the learning. However there are certain limitations to the method. Since the method exploits only early-on, there could be a case in non-stationary problems where the action values change over time. The agent following the optimistic initial value already settled on an option and is now unaware that a different action is better. Also in optimistic initial value method, we have to choose an initial value and this results in a problem since we might not know what the initial value should be because in many cases maximum reward is unknown. The exploration happens if $Q(a) > q^*(a)$, so we set the initial value $Q(a)$ to be optimistically high and the agent explores and adjusts these values accordingly.

Example 3 *Consider the doctor example where the doctor has to prescribe one out of the three medicines (P, Y, B) to the patients. However the doctor does not know which medicine is better, so the doctor decides to solve the problem using RL. Let the reward be* 1 *if the treatment succeeds and otherwise it is* 0*. The update rule is given by* $Q_{n+1} = Q_n + \alpha [R_n - Q_n]$*. Let* α *be* 0.5 *and let the initial values of medicines P, Y, B be* $Q_1(P) = Q_1(B) = Q_1(Y) = 2.0$*. For each of these, we set the* q^* *as well:* $q^*(P) = 0.25$*,* $q^*(Y) = 0.75$*,* $q^*(B) = 0.5$*. The* $q^*(a)$ *in this problem are selected from a normal distribution with mean 0 and variance 1.*

Patient	Medicine P	Medicine Y	Medicine B
1	+1 Q_2(P) = 1.5		
2		+0 Q_3(Y) = 1.0	
3			+1 Q_4(B) = 1.5
4			+1 Q_5(B) = 1.75
5	+0 Q_6(P) = 0.75		
6			+0 Q_7(B) = 0.625
7		+1 Q_8(Y) = 1.0	
8		+1 Q_9(Y) = 1.0	
9		+0 Q_{10}(Y) = 0.5	
10	+0 Q_{11}(P) = 0.375		

Table 1: Running the doctor medicine prescription using the Optimistic Initial Values.

After the first patient comes in, the doctor prescribes the medicine P, the patient feels better, so the $R_1 = 1$ and $Q_1 = 2.0$, then we can apply the update rule as follows.

$$\begin{aligned} Q_2 &= Q_1 + \alpha [R_1 - Q_1] \\ &= 2.0 + 0.5[1.0 - 2.0] \\ &= 2.0 + 0.5[-1.0] \\ &= 2.0 + (-0.5) \\ Q_2 &= 1.5 \end{aligned}$$

The Table 1 shows how the doctor prescribes the medicines to each of the 10 patients. The Q_1 of all the medicines was 2.0 and the values are updated according to the update rule as shown above. Once all the 10 patients are prescribed medicines, the Q values of each of the medicine becomes stabilized: $Q_{11}(P) = 0.375$, $Q_{11}(Y) = 0.5$, and $Q_{11}(B) = 0.625$. As the dry-run indicates that due to the optimistic initial values, the agent explored all three medicines early on and shifted towards exploitation and eventually the estimates converged.

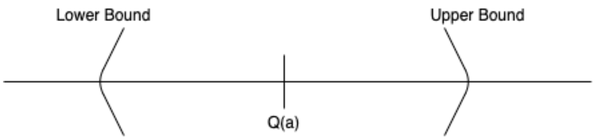

Figure 2: Uncertainity in Estimates. The confidence interval around $q^*(a)$, the uncertainity is represented by $Q(a)$. The upper bracket indicates the upper-bound and the other bracket indicates the lower-bound on the interval. The value of $q^*(a)$ is somewhere in between the two bounds.

3.5.3 Upper-Confidence Bound (UCB) Action Selection

Upper-Confidence Bound (UCB) is another method to balance the exploration and exploitation. Since we are estimating our action values from sampled rewards, there is inherent uncertainity in accuracy of our estimate. Therefore exploration is needed to reduce the uncertainity to be able to take better decisions in the future. The greedy-actions are those that look best at-present, but there may be some actions that may actually be better which are not yet explored. Although ε-greedy action selection forces the non-greedy actions to be tried, but with no preference to those that are nearly greedy or particularly uncertain. UCB uses uncertainity in estimates to drive exploration.

The Figure 2 visually illustrates the confidence interval around $q^*(a)$. If the interval is small, we are certain that $q^*(a)$ is near to our estimate $Q(a)$. However, if the interval is large, we are uncertain that it $q^*(a)$ is near. UCB employs the idea of optimism in the face of uncertainity i.e. in the face of uncertainity, optimistically assume that it is good. For each action and their associated uncertainities, the agent has no idea which to pick, so it optimistically picks the one with the highest upper bound. This is essentially good because it could either be good and the agent gets a good reward or the agent gets to learn about the action and the uncertainity is reduced (improves). UCB selects the non-greedy actions based on their potential for actually being optimal, taking into account both how close their estimates are to being maximal and the uncertainities in those estimates. In this case, the action selection can be done as follows.

$$\boxed{A_t := \operatorname{argmax}_a \left[Q_t(a) + \sqrt[c]{\frac{ln\ t}{N_t(a)}} \right]} \tag{3.9}$$

where $ln\ t$ denotes the natural logrithm of t, $N_t(a)$ denotes the number of times that action a has been selected prior to time t and the number $c > 0$ controls the degree of exploration. The c is a user-defined parameter. The estimated value $Q_t(a)$ is the exploitation factor and the upper confidence bound $\sqrt[c]{\frac{ln\ t}{N_t(a)}}$ is the exploration factor. Each time action a is selected, the uncertainity is reduced since $N_t(a)$ increments, the uncertainity term decreases. On the other hand, if action other than a is selected, t increases, but $N_t(a)$ does not, the uncertainity term increases. The argmax_a allows the maximum the selection of action with maximum combination of both: the estimate and the uncertainity.

3.6 Gradient Bandit Algorithms

We have only considered estimating action-values and then using those to select actions. Another approach is to learn a numerical preference against each action a, denoted by $H(a)$. The larger the preference, the more often that action is chosen. In such an approach, only the relative preference of one action over the other is important. The action preference has no interpretation in terms of the reward. The action probabilities can be determined according to the softmax distribution (Gibbs or Boltzmann distribution).

$$\boxed{Pr\{A_t = a\} := \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}} := \pi(a)} \tag{3.10}$$

here $\pi(a)$ is the probability of choosing an action a at time t. A natural learning algorithm for such a setting is based on the idea of stochastic gradient ascent. On each step, on selecting the action A_t and receiving the reward R_t, the action preferences are updated as follows.

$$\boxed{H_{t+1}(A_t) := H_t(A_t) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(A_t))} \tag{3.11}$$

$$\boxed{H_{t+1}(a) := H_t(a) - \alpha(R_t - \bar{R}_t)\pi_t(a) \ \forall a \neq A} \tag{3.12}$$

where $\alpha > 0$ is the step-size parameter, and $\bar{R}_t \in \mathcal{R}$ is the average of all rewards up through and including time t, which can be computed incrementally. The $\bar{R}_t$ term serves as a baseline with which the reward is compared. If the reward is higher than the baseline, then the probability of taking the action A_t is increased, however if the reward is lower than the baseline, then it is decreased. The non-selected actions move in the opposite directions.

3.6.1 Stochastic Approximation

The gradient bandit algorithm is an stochastic approximation to gradient ascent. Implementing the exact gradient ascent algorithm would mean that each action preference $H_t(a)$ would be updated in proportion to the update's effect on performance.

$$\boxed{H_{t+1}(a) = H_t(a) + \alpha \frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)}} \tag{3.13}$$

where the $\mathbb{E}[R_t]$, the measure of performance, equals $\sum\limits_x \pi_t(x) q_\star(x)$. In this case, we are limited because we do not know the value of $q_\star(x)$. However, we can show that (3.12) is indeed an instance of (3.13) by introducing a baseline B_t in the exact performance gradient since it sums to zero over actions. Therefore, we can write as follows.

$$\boxed{\frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)} = \sum_x (q_\star(x) - B_t) \frac{\partial \pi_t(x)}{\partial H_t(a)}} \tag{3.14}$$

where B_t is a constant scalar not dependent on x. Multiplying (3.14) by $\frac{\pi_t(x)}{\pi_t(x)}$, we obtain $\sum\limits_x \pi_t(x)(q_\star(x) - B_t) \frac{\partial \pi_t(x)}{\partial H_t(a)} / \pi_t(x)$ which expresses the performance in expectations.

$$\boxed{= \mathbb{E}[(q_\star(A_t) - B_t) \frac{\partial \pi_t(A_t)}{\partial H_t(a)} / \pi_t(A_t)]} \tag{3.15}$$

We substitute $B_t = \bar{R}_t$ and $R_t = q_\star(A_t)$ and the equation (3.15) becomes.

$$\boxed{= \mathbb{E}[(R_t - \bar{R}_t)\frac{\partial \pi_t(A_t)}{\partial H_t(a)} / \pi_t(A_t)]} \tag{3.16}$$

Using standard quotient rule for derivatives, it is easy to show that $\frac{\partial \pi_t(x)}{\partial H_t(a)} = \pi_t(x)(\mathbb{1}_{a=x} - \pi_t(a))$, see Rich and Barto's book for detailed proof. Substituting this in (3.16), we obtain as follows.

$$\boxed{= \mathbb{E}[(R_t - \bar{R}_t)\pi_t(A_t)(\mathbb{1}_{a=x} - \pi_t(a))/\pi_t(A_t)] = \mathbb{E}[(R_t - \bar{R}_t)(\mathbb{1}_{a=x} - \pi_t(a))]} \tag{3.17}$$

We can then substitute a sample of the expectation for the performance gradient in (3.17), we obtain the resultant equation that is an instance of the stochastic gradient ascent.

$$\boxed{H_{t+1}(a) = H_t(a) + \alpha(R_t - \bar{R}_t)(\mathbb{1}_{a=x} - \pi_t(a))} \tag{3.18}$$

3.7 Contextual Bandits

The idea of contextual bandits concerns the associative setting. This gently nudges towards the full RL problem where actions effect the next state as well as the reward. In contextual bandits, the actions only effect the immediate reward. Extending the problem definition of k-armed bandits, we also say that the bandit task changes randomly from step to step. In contextual bandits, the goal is to learn a policy that maps situations to the best actions. The inherent idea of contextual bandits is straight-forward. Consider K arms with a reward sequence $r_1, r_2, r_3, ...$, with $r_i \in [0,1]^K$. At each time step t, the bandit algorithm receives some context $s_t \in S$, we call S the context set. Given some context, the bandit algorithm chooses an arm to pull. If the arm A_t at time step t is chosen, then the reward r_t with respect to that arm A_t is obtained. The goal is to learn a policy that can map the context s_t to the best arm A_t (or the best action).

3.8 Implementation

The thought behind writing this guide is to implement as many RL algorithms as humanly possible so that the reader can also relate to the implementation of everything that is written in the text. The language of choice is Python for the ease that it provides in terms of rapid prototyping.

3.8.1 k-Armed Bandits in Action

We discussed how k-armed bandits function and went through a little theory behind them. In this subsection, we implement multi-armed bandits and look at how they perform. First we have to figure out a test environment for this. We choose to opt for the cliched slot-machine environment where we have K slot machines with K arms. Each arm when pulled yields a certain reward and the goal of the agent is to maximize the average cumulative reward. The base environment, which is the SlotMachineEnvironment, is shown in the following code snippet. We first consider the environment where the reward probabilities are fixed i.e. the environment is stationary.

```
class SlotMachineEnvironment:
    def __init__(self, reward_probabilities, rewards):
        """Since this is MAB problem, we first consider the fixed reward
                                              probabilities scenario.
           The reward probabilities and rewards are passed in to the
                                                  environment.
           The agent tries to learn which arms to pull to gain the maximum
                                                  cumulative reward.
        """
        self.reward_probabilities = reward_probabilities
        self.rewards = rewards
        self.k_arms = len(rewards)

    def pull(self, arm):
        if np.random.random() < self.reward_probabilities[arm]:
            return self.rewards[arm]
        else:
            return 0.0
```

Code Listing 1: The SlotMachineEnvironment snippet.

We first consider how a random agent acts in the environment where it pulls the arms randomly and accumulates some reward. This is important because we set this as our baseline agent and see how sophisticated methods such as ε-greedy behave when compared to the random agent.

```
class RandomAgent:
    def __init__(self, environment, steps):
        """Instantiation of a random agent.
        This takes in the environment and the maximum number of steps.
        The agent randomly selects an arm to pull without learning.
        """
        self.environment = environment
        self.steps = steps

    def step(self):
        count_of_arms_pulled = np.zeros(self.environment.k_arms)
        rewards_obtained = []
        cumulative_rewards = []

        for _ in range(self.steps):
            arm_to_pull = np.random.choice(self.environment.k_arms)
            reward = self.environment.pull(arm_to_pull)
            count_of_arms_pulled[arm_to_pull] += 1

            rewards_obtained.append(reward)
            cumulative_rewards.append(sum(rewards)/len(rewards))

        return (count_of_arms_pulled,
                rewards_obtained,
                cumulative_rewards)
```

Code Listing 2: Random Agent snippet.

The random agent implementation is straight-forward since it randomly selects an arm to pull out of the available arms without any idea of which arm is better. Next we move on to implementing the ε-greedy agent.

```
class EpsilonGreedyAgent:
    def __init__(self, environment, steps, epsilon):
        """Instantiation of the epsilon greedy agent.
        This agent estimates the q-values and selects the arm to pull which
                                             has maximum q-value.
        The epsilon decides when to explore and when to exploit.
        """
        self.environment = environment
        self.steps = steps
        self.epsilon = epsilon

    def step(self):
        k_arms = self.environment.k_arms
        q_values = np.zeros(k_arms)
        rewards_obtained_per_arm = np.zeros(k_arms)
        count_of_arms_pulled = np.zeros(k_arms)

        rewards = []
        cumulative_rewards = []

        for _ in range(self.steps):
            if np.random.random() < self.epsilon:
                arm_to_pull = np.random.choice(k_arms)  # explore
            else:
                arm_to_pull = np.argmax(q_values)  # exploit

            reward = self.environment.pull(arm_to_pull)
            rewards_obtained_per_arm[arm_to_pull] += reward
            count_of_arms_pulled[arm_to_pull] += 1
            q_values[arm_to_pull] = rewards_obtained_per_arm[arm_to_pull]/
                                                    count_of_arms_pulled[
                                                    arm_to_pull]

            rewards.append(reward)
            cumulative_rewards.append(sum(rewards)/len(rewards))

        return (count_of_arms_pulled,
                rewards,
                cumulative_rewards)
```

Code Listing 3: ε-Greedy Agent snippet.

The implementation of ε-greedy agent is also relatively straight-forward since the idea is simple. The agent estimates the Q values using the SAM method (discussed above). The problem is simple, we choose to implement it using sample average method, otherwise the incremental estimation is much more efficient since it does not require maintaining all the rewards against time steps. We test the slot environment with different values of ε for the ε-greedy agent. Furthermore, we also implement the Thompson sampling agent which functions by initializing some prior probability, modelled as Beta distribution since we frame our agent as Bernoulli bandit. Once the arm is pulled, and the reward is obtained, then the likelihood belief is updated, which then becomes posterior probability (since some information is now known). This posterior probability is also given by a Beta distribution, but the successes α and failures β are now updated and instead of being initialized with 1 like in the case of prior, the distribution fits better.

```
class ThompsonSamplingAgent:
    def __init__(self, environment, steps):
        """Instantiation of the ThompsonSampling agent.
           The Thompson agent is a Bernoulli bandit (in this case).
        """
        self.environment = environment
        self.steps = steps

    def step(self):
        k_arms = self.environment.k_arms
        count_of_arms_pulled = np.zeros(k_arms)
        rewards_obtained = np.zeros(k_arms)
        failures = np.zeros(k_arms)

        cumulative_rewards = []
        total_rewards = []

        for _ in range(self.steps):
            arm_to_pull = 0
            beta_max = 0
            for j in range(k_arms):
                beta_d = random.betavariate(rewards_obtained[j] + 1,
                                                      failures[j] + 1)
                if beta_d > beta_max:
                    beta_max = beta_d
                    arm_to_pull = j

            count_of_arms_pulled[arm_to_pull] += 1
            reward = self.environment.pull(arm_to_pull)
            if reward > 90: # modification to allow for Bernoulli Bandit
                rewards_obtained[arm_to_pull] += 1 # alpha
            else:
                failures[arm_to_pull] += 1 # beta

            total_rewards.append(reward)
            cumulative_rewards.append(sum(total_rewards)/len(total_rewards))

        return (count_of_arms_pulled,
                rewards_obtained,
                cumulative_rewards)
```

Code Listing 4: ThompsonSampling Agent snippet.

The plot of average cumulative reward over time is given in Figure 3. It is visible from the figure that the ε-greedy agent performs better due to a higher value of $\varepsilon = 0.1$, with the higher value of ε, the agent gets to explore more before settling on one action, or arm, (i.e. exploitation). If the value of ε is 0, then the agent always chooses the greedy action i.e. exploits and it is very likely that this way it will get stuck with a sub-optimal action which might not be fruitful in the long-run. It is important to note that the value of ε is environment dependent and sometimes has to be tuned. It is therefore advised to perform multiple runs and also plot the standard deviation of rewards for better clarity on performance of each

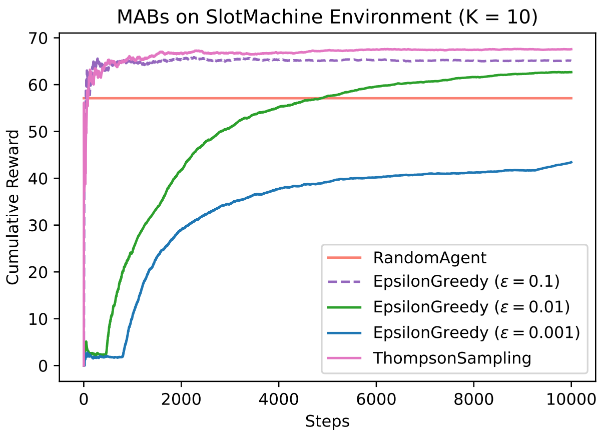

Figure 3: Comparison of different multi-armed bandits with $K = 10$ i.e. the number of arms on the slot machine environment (see Code 1). The best performing agent is shown in pink color.

agent. Another technique looks at decaying the ε over time so that the agent explores early-on and then exploits once sufficient actions have been explored. It is simple to implement with a minor change in the arm selection criteria. However, we can see that the overall best performing agent is the Thompson Sampling agent, colored in pink. One reason is that Thompson sampling functions by estimating probability distributions and employs Bayes rule for updating. Since for each observation, based on the reward a new distribution over the probability of success (defined by higher reward, > 90 for our environment) is computed, therefore the estimates improve as the prior is updated.

4 RL Nuts & Bolts

In this section, we look at the basics of the full reinforcement learning problem and define the terminologies. We also briefly introduce each of the terminology and how a complete RL system functions.