Hands-on Reinforcement Learning

by Premankur Chakraborty

January 29, 2023

Contents

1 Introduction

This project was undertaken as a part of *Winter in Data Science*, an initiative of the Analytics Club, IITB. The project, as the name suggests, deals with an introduction to the basics of Reinforcement Learning.

1.1 What is Reinforcement Learning?

Machine Learning is a very vast topic with many different types and techniques. Reinforcement Learning is one of these types in which a learning agent has to try to take optimal actions in an environment to maximise the total reward over a time period. For each action taken, a function called the *reward function* returns a value called the reward. In RL, we do not explicitly tell the agent the correct behaviour in each situation. Instead, we let the agent figure these things out on its own by *exploring* various different actions, while also *exploiting* actions with the highest rewards. The basic idea is that we reward the agent (usually with some positive reward) for an action in the right direction and we punish it for an unwanted action (usually with a negative reward). In the end, the goal of this agent is to try to reach a point with a maximised reward over a long term.

1.2 Basic Terminology

- **Agent** - Pretty self-explanatory, it's the thing that is doing the actual learning by carrying out actions. It makes decisions based on input from the environment(usually the state and the reward at that time-step).
- **Environment** - Everything outside the agent that it interacts with. It looks at the action made by the agent at each step and returns the current state of the environment along with the reward for that specific action.
- **State** - A set of information that characterises the environment at that time-step.
- **Time-step** - A time step can be considered to be one cycle in which an agent takes an action and then the environment returns the reward of that action back to the agent along with some state information so that the next action can be taken.
- **Action** - The response of an agent to the current state of the environement.
- **Policy** - A mapping from the set of states to a probability distribution of the possible actions to be taken. This is used by an agent to make decisions.
- **Reward** - A numerical value that contributes to the total reward of the agent. This is to be maximised over the long term. Usually positive for a desirable action and negative for an unwanted one.
- **Value Funciton** - An indicator for the long term benefits of a particular state. The value of a state gives us the expected total reward starting from that state.

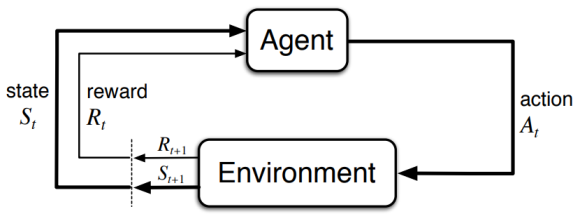

Figure 1: The Agent-Environment interface

2 Bandits

2.1 Definition

Imagine the situation of a slot machine in a casino. It usually consists of a lever which the player pulls, and depending on some probability distribution, the machine returns some reward to the player. This scenario is an example of a **one armed bandit**.

Now, let us expand this scenario with n arms instead of just one. In this scenario, we can choose between n different levers, all of which may have some different, unknown distributions of rewards. Now, the aim of this n-armed bandit problem is to figure out how to maximise the total reward by pulling one of these n levers. This is very close to the problem we aim to solve using Reinforcement Learning. The n different arms of the bandit basically represent the different actions that the agent can take. By taking a certain action (or by pulling a certain lever), the agent gets back a certain reward.

2.2 The Exploration vs. Exploitation Trade-off

Consider an n-armed bandit where the reward returned by any arm of the bandit depends on some probability distribution. Note that this probability distribution of the reward is hidden to the agent. If the agent did know these distributions, it would trivially choose the arm with the best distribution (highest expected reward). Now, the agent only has a limited number of pulls, say T, which we call the horizon.

One strategy is to use the first n pulls to pull each arm once. Thus, we will get some reward from each arm of the bandit. Based on this, we can spend our remaning pulls on the arm that has the highest reward based off of that one pull. However, we can easily see that this is not a very good strategy. It would work quite well if the reward of each arm was deterministic (i.e. each arm had exactly one reward value that it always returns). However, we know that the rewards are actually stochastic. They are chosen from some distribution we do not know anything about.

For example, take $n = 2$. There could be a case where arm 1 returns a reward of 0.1 with probability 0.1 and a reward of 0.8 with probability 0.9. Let us also say that the other arm returns a reward of 0.3 half the time and 0.4 half the time. Now, we start by pulling lever 1 and then lever 2. Let us suppose we get 0.8 for the first lever and 0.4 for the second. In this case, we would continue by pulling lever one for the rest of our pulls. This strategy works great here. However, take the case where lever one gives us the reward of 0.1 and lever two gives us 0.4. In this case, the strategy tells us to pull lever two for the rest of our pulls. However, it is blatantly obvious that the better strategy is to pull lever one as it has a higher expected reward (0.73 versus the 0.35 of the other lever). This clearly shows what is wrong with the strategy used.

In other words, the above strategy simply **exploits** its current knowledge and continuously picks the arm with the highest mean reward so far. However, this is not a good idea because depending on the complexity of the distribution, we have not gained enough information about that arm to conclude that it is, in fact, the best arm to pull. Hence, we need to spend more of our pulls on **exploration**, a process in which we pull arms that don't correspond to the highest mean reward just to see if we can learn some useful information from it.

Now, a great way to explore would be just to choose each arm of the bandit successively. We would spend our pulls equally among all the arms and figure out a lot about the mean rewards those arms give us. However, this too is not a good strategy because it ignores our most fundamental task - to maximise total reward. Hence, a good algorithm must provide a balance between exploration and exploitation.

2.3 Better Algorithms

Suppose that $Q_t(a)$ refers to the mean reward obtained due to action a at timestep t and $q(a)$ refers to the actual or true value of the expected reward upon carrying out action a. Let $N_t(a)$ refer to the number of times action a has been performed till timestep t and R_i refer to the i^{th} reward. Now, we see that:

$$Q_t(a) = \frac{\sum_{i=1}^{N_t(a)} R_i}{N_t(a)}$$

In the limit of $N_t(a) = \infty$, we see that by the law of large numbers, $Q_t(a)$ tends to q(a), or the true value.

In our implementation of this n-armed bandit problem, we must keep track of $Q_t(a)$ and $N_t(a)$ for each a (or for each arm). This is not really that efficient however, and instead we can just store it as the expected reward for the k^{th} reward. We see that:

$$Q_{k+1} = \frac{1}{k}(\sum_{i=1}^{k} R_i)$$

$$Q_{k+1} = Q_k + (\frac{R_k - Q_k}{k})$$

Now, we see that the expected reward is undefined when the arm has never been pulled. We can fix this by defining it to be some initial value which is very large. With this, we can formulate a couple of algorithms. Now, how do we evaluate the performance of these algorithms? One obvious way is of course to keep track of the total reward. Another way is to find the regret of the algorithm. Consider a bandit where the maximum expected reward that can be obtained from any arm is p^* and the minimum is p_{min}. For a total of T steps, the highest possible total reward is Tp^*. Now, the total expected reward is:

$$\sum_0^{T-1} E[r^t]$$

The regret for the algorithm is then:

$$R_T = Tp^* - \sum_0^{T-1} E[r^t]$$

For an algorithm, minimising the regret is desirable. An algorithm that achieves sub-linear regret is a good algorithm. Sub-linear regret means:

$$\lim_{T \to \infty} \frac{R_T}{T} = 0$$

To satisfy this, there are 2 important conditions that must be met:

- **Infinite Exploration** - In the limit of T tending to ∞, each arm must be pulled an infinite number of times.

- **Greed in the Limit** - Let n(T) be the number of greedy pulls out of T total pulls. Then:

$$\lim_{T\to\infty} \frac{n(T)}{T} = 1$$

2.4 Greedy Action Selection

The first is the greedy algorithm that we discussed above. In this algorithm, we greedily choose the action with the highest expected reward $Q(a)$. Here, our action at time t is decided by:

$$A_t = \underset{x}{\text{argmax}}\ Q_t(a)$$

As we've already seen, this strategy is not the best one. Instead, we could try to add a bit of exploration.

2.5 ϵ-Greedy Action Selection

This process of selecting an action is very similar to the greedy method, but with just one small but valuable difference. In this method, we choose the action with the highest expected reward with probability $1 - \epsilon$ and randomly choose an action with probability ϵ.

We see that this algorithm explores more than the greedy one (in which ϵ is 0). Hence, it generally performs better. This algorithm also achieves linear regret (regret is linearly dependant on T).

2.6 UCB Action Selection

The Upper-Confidence-Bound method of action selection is a way to select an action that explores in a better way. In the ϵ-greedy method, we explore by randomly selecting an action from all the possible choices. In this UCB algorithm, we explore by giving a bias to those arms that have higher expected rewards, effectively giving higher weight to those actions that have a better potential for being an optimal action. We do this using the following formula:

$$A_t = \underset{x}{\text{argmax}}\ (Q_t(a) + c\sqrt{\tfrac{ln(t)}{N_t(a)}})$$

Note that here, $N_t(a) = 0$ will give us a as the desirable action. This ensures that every arm is pulled at least once in the beginning.

This algorithm does better than the ones we've already seen. It achieves a favourable sub-linear regret which is of the order O(logT).

2.7 Thompson Sampling

This is another algorithm which achieves sub-linear regret. Consider our actions to give rewards of either 0 or 1 with some bernoulli parameter. Now, for each action, we can keep track of the number of 1s (successes s) and 0s (failures f). Thompson sampling builds a Beta distribution for each action with parameters $(s_a^t + 1, f_a^t + 1)$. It samples a random

draw, say x_a for each action. Now, it chooses the action which gives us the maximum value of x_a.

2.8 Comparison

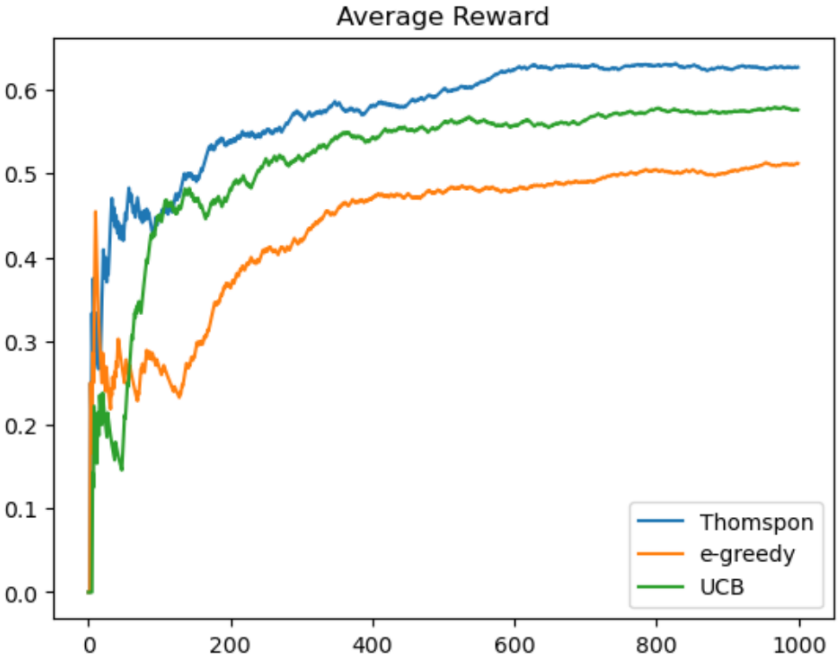

Figure 2: The average reward over time for 3 algorithms

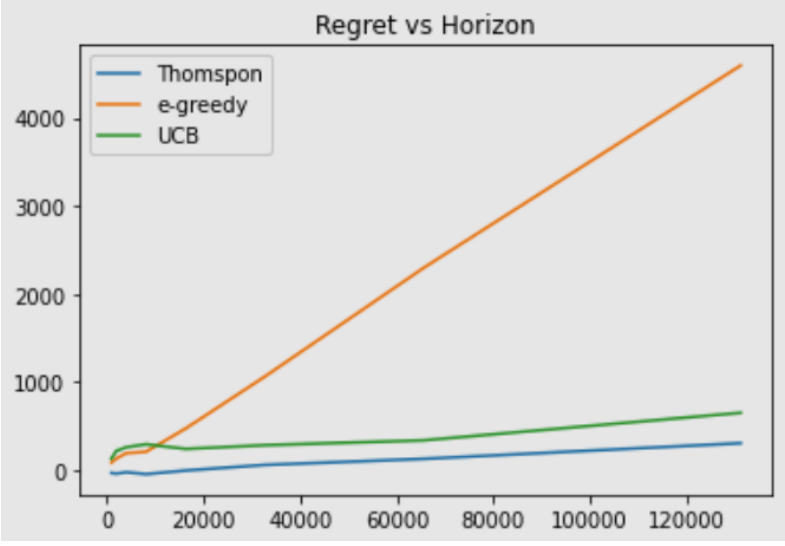

Figure 3: The regret of the algorithm plotted against T(horizon)

As expected, the Thompson algorithm performs much better than the UCB and ϵ-greedy ones. Further, we see that ϵ-greedy achieves a linear regret while the other 2 algorithms clearly have sub-linear regrets.

3 Markov Decision Processes

3.1 Definition

As shown above, the agent and environment interact in discrete time steps. In each time step, the agent receives the state of the environment and the reward of the previous action from the environment. After this, it decides on an action based on a policy function π where $\pi(a|s) = P(A_t = a|S_t = s)$. This action is sent to the environment and this produces the next state and reward. G_t is defined as the return at a given time-step. The expected return or $E[G_t]$ is what an agent tries to maximise at each step.

$$G_t = \sum_{k=0}^{T-k-1} R_{t+k+1}$$

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

In the first equation, the process terminates after a certain number of time-steps (T) and so, G will remain finite. However, in an infinite situation, we need a discount rate γ which lies between 0 and 1.

3.2 Markov Property

A state must contain all the relevant information that an agent might need to make a decision. At the same time, the state should also be compact. A state signal that successfully retains all relevant information is said to be a Markov state:

$$Pr(R_{t+1} = r, St + 1 = s|S_0, A_0, R_1, ...R_t, S_t, A_t) = Pr(R_{t+1} = r, S_{t+1} = s|S_t, A_t)$$

In simple terms, we don't need to remember the previous states. A Reinforcement learning task that satisfies the Markov property is called a markov decision process.

For a policy π, we can define 2 functions called the state value function $v_\pi(s)$ and the action value function $q_\pi(s, a)$.

The Bellman Equations give us:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r +_\pi (s')] \tag{1}$$

$$q_\pi(s, a) = \sum_{s'} p(r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a')) \tag{2}$$

3.3 Bellman Optimality Equations

We can show the existence of a policy $\pi*$ such that the state-value function and action-value functions are maximum for this policy for all states. The optimal value functions satisfy the following equations, called the Bellman Optimality Equations:

$$v_*(s) = \max_{a \in A(s)} \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \tag{3}$$

$$q_*(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')] \tag{4}$$

For finite MDPs, the solutions to the Bellman optimality equations are unique and independent of the policy. Once the state value and action value functions are found, it is possible to determine the policy π. For example, if we have $q_*(s, a)$, all we have to do is find the action a which maximises this function. That is the desirable action.

4 Dynamic Programming

4.1 Meaning

Dynamic Programming refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov Decision Process (MDP). These algorithms can work really well on a perfect model of the environment but require a lot of computational power and so, have limitations. The key idea behind the use of dynamic programming is to use value functions to organise the search for optimal policies.

As seen before, Bellman's optimality equations give us relations between value functions of past and current states. We use this to find relations that can be used to iteratively update these value functions. This forms the basis of our dynamic programming algorithms.

4.2 Policy Evaluation and Improvement

Policy evaluation involves evaluating the state-value function v_π given an arbitrary policy π. Even if the environment's dynamics are completely known, the computations are very hard. Hence, we use an iterative method to find the value function. Note that we can easily do that using equation 1 by converting the bellman equation into an updation rule for the value function.

Now, why are we doing this? We need the value function so that we can improve upon it. As we have seen in the section on MDPs, we can always find a value function with a policy that works better than a previous one until we reach the optimal policy. We do this by using:

$$\pi * (s) = \underset{a}{\text{argmax}} \; q_\pi(s, a) \tag{5}$$

In this way, we will obtain a new policy that is strictly better than the old one. Only if the starting policy is already optimal will we get a policy of the same strength.

4.3 Policy Iteration

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*$$

Policy Iteration is the first dynamic programming algorithm that we can use to try to find an optimal policy to solve the MDP. Policy iteration works with a series of successive runs of policy evaluation and then policy improvement. In each of these cycles, we get a policy that is better than the previous one. In the end, we will obtain a policy that is optimal. Policy iteration can also be made a little bit faster by using the previously

computed value function of the previous policy while computing the value function of the new policy.

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
      $\Delta \leftarrow 0$
      For each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\big[r + \gamma V(s')\big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      $a \leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
      If $a \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return V and π; else go to 2

Figure 4: An algorithm for Policy Iteration

The shown algorithm can fail when 2 policies which are equally good are successively picked. This can be worked around, however, by adding extra checks.

4.4 Value Iteration

Policy Iteration has a few drawbacks. Firstly, each step of this algorithm will involve a policy evaluation, which has an iterative implementation. Further, the convergence to v_* happens only in the limit when we carry out policy iteration. We can show that there is no need to carry out iterations until exact convergence. We can, in fact truncate out policy iteration.

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathcal{S}$:
      $v \leftarrow V(s)$
      $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, π, such that
   $\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$

Figure 5: An algorithm for Value Iteration

Value iteration is a type of policy iteration that is stopped after just one run, that is, one full backup of all states. After this, we move to policy improvement directly. For all

$s \in S$, we carry out:

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')]$$

Observe that this is simply equation 3, the bellman optimality equation converted into an update rule. In this way, we avoid some of the extra computation that needs to be carried out in policy iteration.