RetroRL

Summer of Code

Dattaraj Salunkhe
Mentors: Devavrat Patni, Tanay Bhat

1 Multi Armed Bandits

Multi armed bandit problems are problems in which we are faced with choosing between k options, such that after each choice we get a reward from a probability distribution of the option. Our aim is to maximise the total expected reward after a certain number of choices. This is an example of a k-multi armed bandit problem.
We do not know the exact value of the expected reward, though have an estimate for it. At each time step, we will know which option had yielded the maximum reward. We can then choose that option in the future. This is called *exploiting* our prior knowledge and the action is called a *greedyaction*, and the. We can also try one of the non greedy action, and see if it yields a better reward. This is called *exploring* our options. We can have two basic algorithms based on this, greedy and ϵ-greedy algorithms. We always choose the greedy action in the former one, however in the latter, we explore our options an ϵ fraction of the total choices, and exploit in the remaining $1-\epsilon$.

Consider a multi-armed bandit problem with a set of K actions (slot machines). Let $Q(a)$ denote the expected reward for taking action a. Initially, all Q-values are unknown.

As we play the machines and receive rewards (r_t), we update the Q-value for the chosen action a_t using:

$$Q(a_t) \leftarrow Q(a_t) + \alpha(r_t - Q(a_t)) \tag{1}$$

Here, α (0 ¡ α ¡= 1) is the learning rate that controls how much weight to give to the new reward (r_t) compared to the previous estimate ($Q(a_t)$).

The challenge lies in balancing exploration (trying new machines) and exploitation (playing the seemingly best one). Algorithms like ε-greedy address this by using the Q-values to decide which machine to pull, with a small chance (ε) for exploration.

2 Markov Decision Processes

Here, we imagine an agent interacting with its environment in discrete steps. At each step, the environment presents the agent with a "state," which could be anything from a robot's battery level to the current game board in chess. Based on this state, the agent chooses an "action" (play a move).

The key concept here is the Markov property. This basically states that the future of the system only depends on the current state, not the entire history leading up to it. Imagine a robot deciding to explore a maze. As long as it knows its current location (state), it doesn't matter how it got there - it can make the best decision based on that location alone.

Coming to the concept of rewards, they are numerical values assigned to the outcomes of the agent's actions. The goal, of course, is to maximize the total reward the agent receives over time. MDPs provide a framework for the agent to learn optimal policies, which are essentially maps that tell the agent the best action to take in each state to achieve the highest long-term rewards.

3 Dynamic Programming

Imagine you're planning a long hike with rest stops along the way. DP helps you find the most efficient route by breaking down the problem into smaller, more manageable sub-problems. It considers each rest stop as a state and calculates the optimal value (future reward) of being at that state. This value considers not just the immediate reward of resting but also the future rewards (effort saved) gained by being more rested on later parts of the hike.
DP algorithms work backward, starting from the end goal (your final destination) and iteratively calculating the optimal value for each state. This way, when you're at a specific rest stop (state), you know the best action to take (continue or rest) based on the pre-calculated value of reaching the next stop (considering both immediate and future rewards). This approach ensures you make optimal decisions at each step, leading to the most efficient route (highest overall reward) for the entire hike (MDP).

4 Implementation of multi-armed bandits and pole balancing

In the multi-armed bandit implementation using different values of ϵ, it was found that higher values of ϵ reach faster to their maximum rewards, while the lower ϵ takes longer time to reach its maximum, but it gives slightly higher rewards than the values of ϵ higher than it. Also, the $\epsilon = 0$ gives very low rewards, as it does not explore at all.

In the pole balancing problem, the goal is to design a controller or train an agent to keep the pole upright for as long as possible by applying a force (left or right) to the cart. The challenge arises from the unstable nature of the system - any small tilt in the pole or movement of the cart can cause it to fall. This problem requires the controller or agent to constantly monitor the state of the system (cart position, pole angle, etc.) and make quick decisions (push left or right) based on that information to keep the pole balanced.