SOC 2024: RetroRL
Week 1

Uday Singh, 22B1262

June 18, 2024

1 Multi Armed Bandit Problem

1.1 Introduction

The *k-armed bandit problem*, so named by analogy to a slot machine, or "one-armed bandit," except that it has k levers instead of one. Each action selection is like a play of one of the slot machine's levers, and the rewards are the payoffs for hitting the jackpot. Through repeated action selections you are to maximize your winnings by concentrating your actions on the best levers.

1.2 Explaining the Problem

Imagine you're staying in an old town for 300 days due to work, and the town has only three restaurants where you'll be having dinner each night. You want to get the best experience for the price you pay (assume all restaurants charge the same and you rate the experience on a scale of 10).

Here's the twist: each restaurant has a probability distribution of the experience they provide on any given day. For example, one restaurant might have a mean experience of 8 with a variance of 1.

You want to maximize your total experience points over the 300 days. Note that you don't know these distributions beforehand.

One approach would be to visit restaurants 1, 2, and 3 for 100 days each. This is called the exploration method.

Another approach would be to try each restaurant for the first 3 days, then choose the best one based on these initial experiences and go there for the remaining days. This is called the exploitation method. Note that the experience points from the same restaurant vary according to this probability distribution.

A third approach is to split your time between exploring and exploiting. You keep some days for exploring all the restaurants and other days for exploiting the best-known restaurant based on the exploration results. This is called the epsilon-greedy strategy (epsilon is the fraction of exploring days).

1.3 Mathematical Terminologies

In our k-armed bandit problem, each of the k actions has an expected or mean reward given that the action is selected; let us call this the *value* of that action. We denote the action selected on time step t as A_t, and the corresponding reward as R_t. The value then of an arbitrary action a, denoted $q^*(a)$, is the expected reward given that a is selected:

$$q^*(a) \doteq E[R_t \mid A_t = a].$$

We denote the estimated value of action a at time step t as $Q_t(a)$. We would like $Q_t(a)$ to be close to $q^*(a)$.

1.4 Useful Formulas

- $$Q_t(a) \doteq \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t}$$
- Greedy action selection method
$$A_t \doteq \arg\max_a Q_t(a)$$
- The update rule

  New Estimate = Old Estimate + Step Size × (Target − Old Estimate)

$$Q_{n+1} = Q_n + \frac{1}{n}(R_n - Q_n)$$

1.5 Coding the Problem

Code and Results at `https://github.com/Uday1Singh/RetroRL-SOC-2024`

2 Markov Decision Process

2.1 Introduction

A Markov Decision Process (MDP) is a mathematical framework for modeling decision-making in situations where outcomes are partly under the control of a decision maker and partly random.

2.2 Components of MDP

An MDP is defined by the following components:

- **States (S)**: A finite set of states that represent all possible situations.
- **Actions (A)**: A finite set of actions available to the decision maker.
- **Transition Model (P)**: The probability $P(s'|s, a)$ of transitioning from state s to state s' when action a is taken.
- **Reward Function (R)**: The immediate reward received after transitioning from state s to state s' by taking action a, denoted as $R(s, a, s')$.
- **Discount Factor (γ)**: A factor $\gamma \in [0, 1)$ that represents the difference in importance between future rewards and present rewards.

2.3 Policy

A policy π is a strategy used by the decision maker to choose actions based on the current state. Formally, a policy π is a mapping from states to a probability distribution over actions, $\pi : S \to \mathcal{P}(A)$.

2.4 Value Function

The value function is used to estimate the expected cumulative reward from each state under a particular policy.

2.4.1 State-Value Function

The state-value function $V^\pi(s)$ under policy π is defined as:

$$V^\pi(s) = E^\pi \left[G_t \mid S_t = s \right] \tag{1}$$

2.4.2 Action-Value Function

The action-value function $Q^\pi(s, a)$ under policy π is defined as:

$$Q^\pi(s, a) = E^\pi \left[G_t \mid S_t = s, A_t = a \right] \tag{2}$$

2.5 Bellman Equations

The Bellman equations provide a recursive decomposition of the value functions.

2.5.1 Bellman Expectation Equation for V^π

$$V^\pi(s) = E^\pi \left[R_{t+1} + \gamma V^\pi(S_{t+1}) \mid S_t = s \right] \tag{3}$$

2.5.2 Bellman Expectation Equation for Q^π

$$Q^\pi(s, a) = E^\pi \left[R_{t+1} + \gamma Q^\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a \right] \tag{4}$$

2.6 Optimality

The goal in an MDP is to find an optimal policy π^* that maximizes the expected cumulative reward.

2.6.1 Optimal State-Value Function

The optimal state-value function $V^*(s)$ satisfies the Bellman optimality equation:

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} P(s'|s, a)[R(s, a, s') + \gamma V^*(s')] \tag{5}$$

2.6.2 Optimal Action-Value Function

The optimal action-value function $Q^*(s, a)$ satisfies the Bellman optimality equation:

$$Q^*(s, a) = \sum_{s' \in S} P(s'|s, a) \left[R(s, a, s') + \gamma \max_{a' \in A} Q^*(s', a') \right] \tag{6}$$

The optimal policy π^* can be derived from $Q^*(s, a)$ by choosing the action that maximizes $Q^*(s, a)$ for each state s.

2.7 Pole Balancing Problem

The objective in this task is to apply forces to a cart moving along a track so as to keep a pole hinged to the cart from falling over. The control objective is to keep the pole in the vertical position by applying horizontal actions (forces) to the cart.

The action space consists of two actions:

- Push the cart left – denoted by 0
- Push the cart right – denoted by 1

2.7.1 The observation space

The observation space or the states are:

- **Cart position**, denoted by x. The minimal and maximal values are -4.8 and 4.8, respectively.
- **Cart velocity**, denoted by $\dot{x}$. The minimal and maximal values are $-\infty$ and ∞, respectively.
- **Pole angle of rotation** (measured in radians), denoted by θ. The minimal and maximal values are -0.418 radians ($-24°$) and 0.418 radians ($24°$).
- **Pole angular velocity**, denoted by $\dot{\theta}$. The minimal and maximal values are $-\infty$ and ∞, respectively.

2.7.2 Code

Code and Results at `https://github.com/Uday1Singh/RetroRL-SOC-2024`

3 Dynamic Programming

Dynamic programming (DP) is a foundational method used in reinforcement learning (RL) to solve Markov Decision Processes (MDPs). It provides a suite of algorithms that can efficiently compute optimal policies and value functions. The core idea of dynamic programming is to break down complex problems into simpler subproblems and solve them recursively.

3.1 Policy Evaluation

This is the process of computing the value function for a given policy. The value function $V^\pi(s)$ for a policy π is the expected return starting from state s and following policy π.

$$V^\pi(s) = E_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \mid S_0 = s \right]$$

3.2 Policy Improvement

This involves improving the current policy by making it greedy with respect to the current value function. The new policy π' can be obtained by:

$$\pi'(s) = \arg\max_a \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma V^\pi(s') \right]$$

3.3 Policy Iteration

This method alternates between policy evaluation and policy improvement until convergence. It is guaranteed to converge to the optimal policy and value function.

1. Initialize a policy π
2. Repeat until convergence:
   a. Policy Evaluation: Compute V^π for the current policy π
   b. Policy Improvement: Update the policy to be greedy with respect to V^π

3.4 Value Iteration

Instead of evaluating the policy to convergence, value iteration simplifies the process by directly updating the value function using the Bellman optimality equation:

$$V_{k+1}(s) = \max_a \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma V_k(s') \right]$$

This method iteratively updates the value function until it converges to the optimal value function V^*, from which the optimal policy π^* can be derived.