Reinforcement Learning

SoS MidTerm Report

Balaji Karedla

June 2024

Contents

1 Introduction

This is the report of my project on Reinforcement Learning as a part of Summer of Science 2024 by the Mathematics and Physics (MnP) Club.

This is a branch of Machine Learning along with Supervised and Unsupervised Learning. In Supervised Learning, the model is trained on a labeled dataset, while in Unsupervised Learning, the model just goes through the dataset without any labels.

Reinforcement Learning differs from the both as it learns from the environment by interacting with it. The agent learns to achieve a goal by trying out stuff, getting rewards or punishments, and learning from time to time.

Reinforcement Learning is therefore close to how humans learn from their surroundings. But the way human learns from the environment is too complex and not yet understood completely. But, the RL algorithms are designed to mimic the way humans learn.

This is what made me feel interested in Reinforcement Learning. The work I did in this project is present in the repo on GitHub. The graphs I plotted, the code I wrote are all present in the repo. The code is written in Python and the libraries used are `numpy` and `matplotlib`.

2 Multi-Armed Bandits (MAB)

Multi-Armed Bandits are common in casinos and are one of the most basic problems in Reinforcement Learning. The slot machines have levers (arms), which the player pulls to play the game.The player is rewarded according to the arm pulled. The The goal, just like any other game, is to maximize the total reward by selecting the arms accordingly.

Figure 1: A Slot Machine or an Armed Bandit

The Multi-Armed Bandit I studied is a simple version of the Reinforcement Learning problem, where an agent interacts with an environment by selecting one of the available actions (arms) at each time step.

I chose a 10-armed bandit from [1], where the mean of the rewards given by the bandit was randomly selected over a constant distribution over a $[-2, 2)$ range. The bandits give a reward chosen from a normal distribution with a mean equal to the mean of the bandit and a standard deviation of 1. The agent aims to maximize the total reward by selecting the bandit with the highest expected reward.

2.1 Agent Implementation

I implemented the agent using the different algorithms given in [1]. The implementation in all of these algorithms included storing an array of action values, which are estimates of the expected rewards of each bandit. The agent selects the action with the highest action value to *exploit*, the best-known bandit, or selects a random action to *explore* other bandits. The agent updates the action values based on the rewards received from the environment.

The updation of the action values is different across different algorithms.

2.2 Greedy Algorithm

The Greedy Algorithm is the simplest Algorithm for solving the Multi-Armed Bandit problem. The agent keeps track of the action values for each bandit and selects the bandit with the highest action value at each time step. The agent *exploits* the best-known bandit by choosing the action with the highest action value.

The action values of the bandits, represented by q_*, are estimated by the agent using the sample average method to get an estimate, represented by Q_t. The action value of a bandit a at time step t is updated using the following formula:

$$q_* \doteq \mathbb{E}[R_t | A_t = a]$$

$$Q_t(a) \doteq \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}}$$

where R_t is the reward received by the agent at time step t, A_t is the action selected by the agent at time step t, and $\mathbb{1}_{A_i=a}$ is an indicator function that is 1 if $A_i = a$ and 0 otherwise.

In the greedy algorithm, the agent always selects the bandit with the highest action value. So,

$$A_t \doteq \arg\max_a Q_t(a)$$

This definition of Q_t lets us define it recursively as follows:

$$Q_{t+1} = Q_t + \frac{1}{n}[R_t - Q_t]$$

where n is the number of times the action a has been selected.

2.3 Exploration-Exploitation Dilemma

The Greedy Algorithm has a drawback in that it always selects the bandit with the highest action value, which can lead to suboptimal performance. The agent may not explore other bandits to learn their reward distributions, which can result in a lower total reward.

This can be overcome by choosing a random bandit every time with equal probability to explore the bandits. This ensures the agent explores all the bandits and learns their reward distributions. But the reward is highly distributed, so the agent may be unable to exploit the best bandit.

This is the Exploration-Exploitation Dilemma in the Multi-Armed Bandit problem. This can be overcome by choosing the perfect ratio of exploration and exploitation.

2.4 ϵ-Greedy Algorithm

The ϵ-Greedy Algorithm is a simple solution to the Exploration-Exploitation Dilemma. The agent selects the best-known bandit with probability $1-\epsilon$ and selects a random bandit with probability ϵ. The agent *exploits* the best-known bandit with probability $1-\epsilon$ and *explores* other bandits with probability ϵ[1].

The action value of a bandit a at time step t is updated using the following formula:

$$Q_{t+1}(a) = Q_t(a) + \frac{1}{n}[R_t - Q_t(a)]$$

where n is the number of times the action a has been selected.

The value of ϵ is a hyperparameter that controls the exploration and exploitation trade-off. A higher value of ϵ encourages more exploration, while a lower value of ϵ encourages more exploitation.

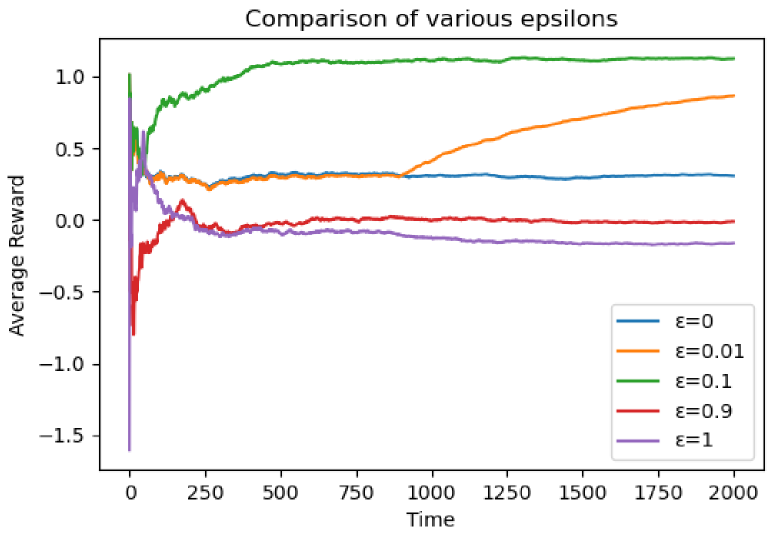

Figure 2: The ϵ-Greedy Algorithm

The 2 shows the performance of the ϵ-Greedy Algorithm with different values of ϵ. The performance of the algorithm is measured by the average reward over time. The algorithm with $\epsilon = 0.1$ performs better than the algorithm with $\epsilon = 0.01$ and $\epsilon = 0$.

[1]Assuming there are k bandits, the probability of selecting a bandit a at time step t is given by:

$$\pi(a_t = a|S_t) = \begin{cases} 1-\epsilon+\frac{\epsilon}{k} & \text{if } a = \arg\max_a Q_t(a) \\ \frac{\epsilon}{k} & \text{otherwise} \end{cases}$$

It is obvious that the greedy algorithm grows faster than the other algorithm at the beginning but the ϵ-greedy algorithm with $\epsilon = 0.1$ performs better than the greedy algorithm in the long run. The $\epsilon = 0.01$ algorithm is closer to the greedy algorithm, but it performs better than the greedy algorithm in the long run, it even performs better than the $\epsilon = 0.1$ algorithm in the long run, but it grows slower than the $\epsilon = 0.1$ algorithm.

Completely random selection of bandits ($\epsilon = 0$) performs the worst among all the algorithms. The agent does not exploit the best-known bandit and spends most of the time exploring other bandits, resulting in a lower total reward.

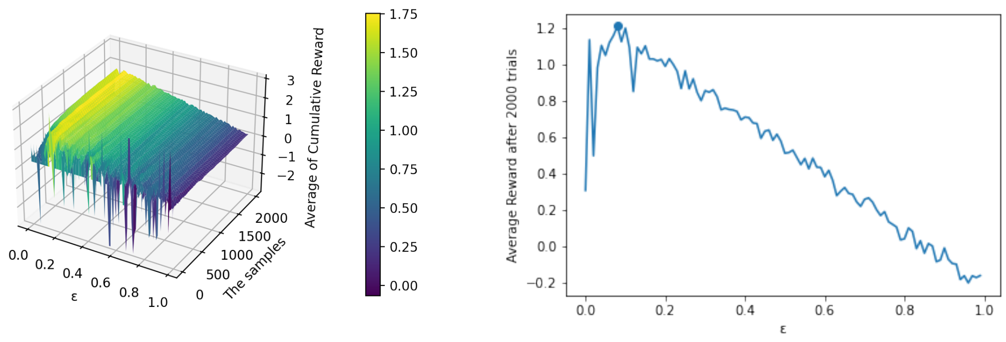

Figure 3: Comparison of epsilons in the ϵ-Greedy Algorithm

The maximum of the reward is at $\epsilon = 0.08$ and then decreases.

2.5 Tracking a Non-Stationary Problem

The Multi-Armed Bandit problem can be made more challenging by introducing non-stationary rewards. In the non-stationary problem, the reward distributions of the bandits change over time. The mean of the rewards given by the bandits is updated at each time step by adding a small random value to the mean.

The non-stationary problem is more challenging because the agent has to adapt to the changing reward distributions to maximize the total reward. The agent has to balance exploration and exploitation to learn the new reward distributions and exploit the best-known bandit.

To solve the non-stationary problem, the agent has to use a different approach to update the action values. The agent has to give more weight to recent rewards to adapt to the changing reward distributions.

One of the most popular ways to do it is to use a constant step-size parameter α to update the action values. The action value of a bandit a at time step t is updated using the following formula:

$$Q_{n+1} \doteq Q_n + \alpha[R_n - Q_n] \tag{1}$$

The Q_{n+1} can be written in terms of all the previous rewards and the initial estimate of the action value with:

$$\begin{aligned}
Q_{n+1} &= Q_n + \alpha[R_n - Q_n] \\
&= \alpha R_n + (1-\alpha)Q_n \\
&= \alpha R_n + (1-\alpha)[\alpha R_{n-1} + (1-\alpha)Q_{n-1}] \\
&= \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2 Q_{n-1} \\
&= \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2\alpha R_{n-2} + \cdots + (1-\alpha)^{n-1}\alpha R_1 + (1-\alpha)^n Q_1 \\
&= (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i
\end{aligned}$$

This is called a weighted average because the sum of the weights is $(1-\alpha)^n + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} = 1$. The quantity $(1-\alpha)$ is less than 1, and thus the weight given to Ri decreases as the number of intervening rewards increases. In fact, the weight decays exponentially according to the exponent on $1-\alpha$. Accordingly, this is sometimes called an *exponential recency-weighted average.*

Sometimes it is convenient to vary the step-size parameter from step to step. Let $\alpha_n(a)$ denote the step-size parameter used to process the reward received after the nth selection of action a. As we have noted, the choice $\alpha_n(a) = \frac{1}{n}$ results in the sample-average method, which is guaranteed to converge to the true action values by the law of large numbers. But of course convergence is not guaranteed for all choices of the sequence $\{\alpha_n(a)\}$. A well-known result in stochastic approximation theory gives us the conditions required to assure convergence with probability 1:

$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty \qquad \text{and} \qquad \sum_{n=1}^{\infty} \alpha_n^2(a) < \infty \tag{2}$$

Or more formally, $\sum_{n=1}^{\infty}$ diverges and $\sum_{n=1}^{\infty}$ converges. The first condition is required to guarantee that the steps are large enough to eventually overcome any initial conditions or random fluctuations. The second condition guarantees that eventually the steps become small enough to assure convergence.

2.6 Optimistic Initial Values

The Greedy Algorithm and the ϵ-Greedy Algorithm have a drawback in that they may get stuck in a suboptimal bandit. The agent may not explore other bandits to learn their reward distributions, which can result in a lower total reward.

This can be overcome by using optimistic initial values for the action values. The agent initializes the action values with a high value to encourage exploration. The agent explores other bandits to learn their reward distributions and exploits the best-known bandit to maximize the total reward.

The action values of the bandits are initialized with a high value, such as 10, to encourage exploration. The agent selects the bandit with the highest action value at each time step to *exploit* the best-known bandit or selects a random bandit to *explore* other bandits.

As can be seen in the Figure 4, the optimistic initial values algorithm performs better than the greedy algorithm in the long run. The agent explores other bandits to learn their reward distributions and exploits the best-known bandit to maximize the total reward.

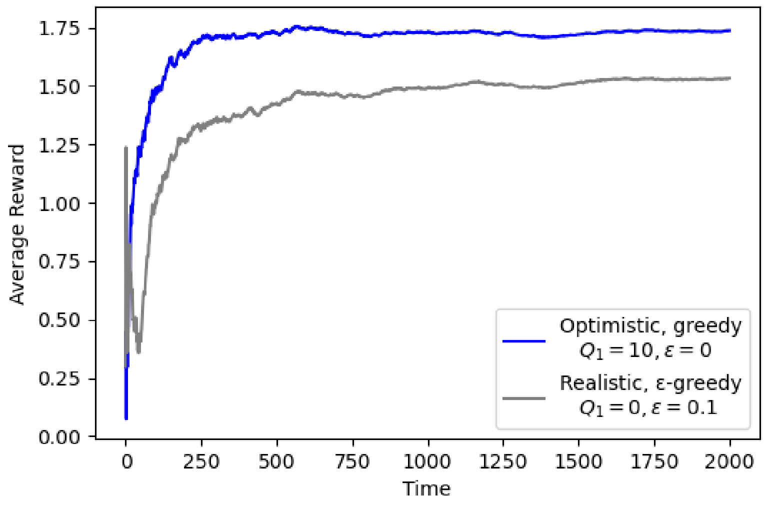

Figure 4: Comparison of Optimistic Initial Values

2.7 Upper-Confidence-Bound (UCB) Action Selection

Exploration is needed because the agent is uncertain about the reward distributions of the bandits. The agent has to explore other bandits to learn their reward distributions and exploit the best-known bandit to maximize the total reward. The agent has to balance exploration and exploitation to maximize the total reward. The agent has to choose the perfect ratio of exploration and exploitation.

The Upper-Confidence-Bound (UCB) Action Selection Algorithm is one of the many solutions to the Exploration-Exploitation Dilemma. The agent selects the bandit with the highest upper confidence bound at each time step. The agent *exploits* the best-known bandit by choosing the action with the highest upper confidence bound.

The action value of a bandit a at time step t is updated using the following formula:

$$A_t \doteq \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right] \tag{3}$$

where $N_t(a)$ is the number of times the action a has been selected prior to time t, and c is a hyperparameter that controls the exploration and exploitation trade-off. A higher value of c encourages more exploration, while a lower value of c encourages more exploitation. plt.savefig('report/images/ucb.png')

As can be seen in the Figure 5, the UCB Algorithm performs better than the Greedy Algorithm and the ϵ-Greedy Algorithm. The agent explores other bandits to learn their reward distributions and exploits the best-known bandit to maximize the total reward.

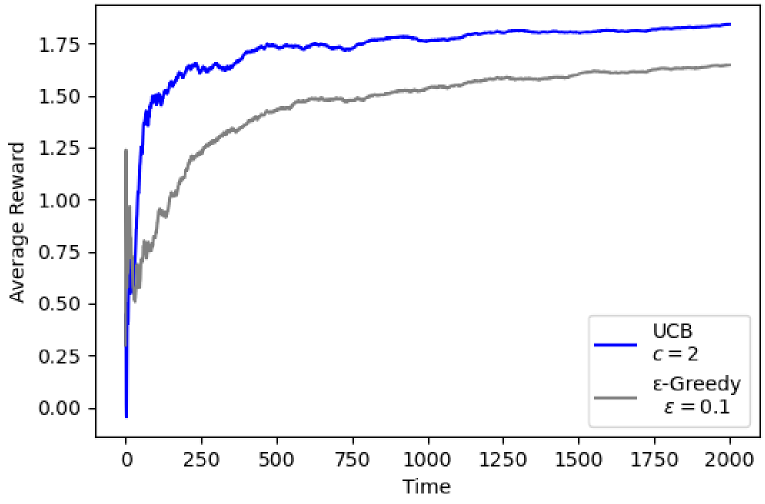

Figure 5: The UCB Algorithm

2.8 Gradient Bandit Algorithm

2.9 Associative Search (Contextual Bandits)

I skipped the above topics… :P

3 Finite Markov Decision Processes

This chapter introduces the formal definition of Markov Decision Processes and the optimal policies.

3.1 The Agent-Environment Interface

The MDP is a mathematical framework for modeling decision-making problems. It formulates the problem as an agent interacting with an environment. The agent and the environment interact at discrete time steps, and the agent makes decisions based on the environment's state. The environment then transitions to a new state, and the agent receives a reward based on the transition.

The interactions between the agent and the environment are taken to be at a sequence of discrete time steps $t = 0, 1, 2, 3, \ldots$. At each time step t, the agent receives some representation of the environment's state $S_t \in \mathcal{S}$, where $\mathcal{S}$ is the set of all possible states. The agent then selects an action $A_t \in \mathcal{A}(S_t)$, where $\mathcal{A}(S_t)$ is the set of all possible actions in state S_t. The environment then transitions to a new state S_{t+1} and the agent receives a reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$. The agent's goal is to maximize the cumulative reward it receives over time. The MDP and agent together give rise to a sequence or *trajectory* that is a sequence of state-action-reward triples.

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, S_3, A_3, R_4, \ldots \tag{4}$$

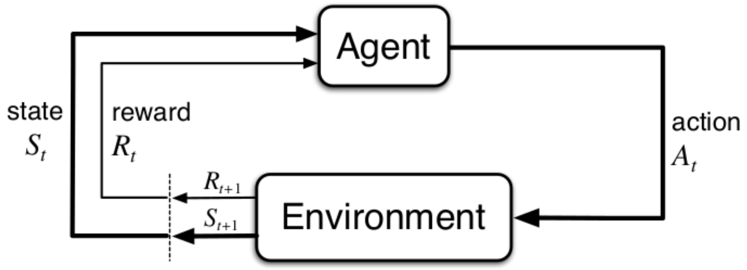

Figure 6: The agent-environment interface in a Markov Decision Process.

In a finite MDP, the sets of states, actions, and rewards are all finite. The agent-environment interface in a finite MDP is shown in Figure 6. The random variables R_t and S_t, have a well-defined probability distribution that depends only on the preceding state and action. That is for particular values of these random variables $s' \in \mathcal{S}$ and $r \in \mathcal{R}$, there is a probability of these values occurring given the preceding state and action:

$$p(s', r \mid s, a) \doteq \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\} \tag{5}$$

for all $s', s \in \mathcal{S}, r \in \mathcal{R}$ and $a \in \mathcal{A}(s)$

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) = 1 \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}(s) \tag{6}$$

If a state s' and reward r depend only on the preceding state and action, then the environment is said to be a *Markov Decision Process.* And, if a state contains all relevant information from the history, then the state is said to be *Markov* and the process is said to have the *Markov Property.*

$$p(s' \mid s, a) \doteq \Pr\{S_t = s' \mid S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in \mathcal{R}} p(s', r \mid s, a) \tag{7}$$

$$r(s, a) \doteq \mathbb{E}[R_t \mid S_{t-1} = s, A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} rp(s', r \mid s, a) \tag{8}$$

The goal of any agent is to maximize its rewards.

3.2 Returns and Episodes

The agent run can be taken as *continuing* or *episodic*, to consider the iterations of the game.

For example, consider a robot that is trying to navigate a maze. In the continuing case, the robot continues to navigate the maze indefinitely, and the agent-environment interaction does not terminate. In the episodic case, the robot navigation is considered as episodes of trying to reach the goal and an episode ends when the robot reaches the goal or when it gets stuck in the maze.

Continuing method is used when the life of the bot is long and needs to survive, while *Episodic* is used to define short games.

The agent tries to maximize the cumulative reward it receives over time. The *expected return*

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T \tag{9}$$

where the T is the final time step of the episode. The return depends on the rewards received by the agent at each time step. The agent's goal is to maximize the expected return over time.

When the life-span is not finite, the G_t doesn't converge as $T \to \infty$. Hence, a factor of γ is used to define G_t as

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \dots) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

This even works even when termination occurs at $t = T$, provided we define $G_T = 0$.

3.3 Policies and Value Functions

A *policy* is a mapping from states to probabilities of selecting each possible action. If an agent is following a policy π at a time, the probability of choosing an action a, when you're in the state s is denoted by $\pi(a \mid s)$.

The *value function* of a state s under a policy π, denoted by $v_\pi(s)$, is the expected return when starting in s, following the policy π.

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s\right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right] \tag{10}$$

The expected return when we choose an action a from s, following the policy π, is defined as $q_\pi(s, a)$ and is defined as

$$q_\pi(s,a) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a\right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right] \tag{11}$$

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi \left[G_t \mid S_t = s\right] && (12) \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \mid S_t = s\right] && (13) \\ &= \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma \mathbb{E}_\pi \left[G_{t+1} \mid S_{t+1} = s'\right]\right] && (14) \\ &= \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma v_\pi(s')\right], \quad \text{for all } s \in \mathcal{S} && (15) \end{aligned}$$

The equation 15 is called the *Bellman Equation* for v_π. It expresses a relationship between the value of a state and the values of its successor states.

3.4 Optimal Policies and Optimal Value Functions

The goal of Reinforcement Learning is to find the optimal policy π, which maximizes the expected return. A policy π is defined to be better than or equal to another policy π' if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$. In other words, $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$. The optimal policy is denoted by π_*. The optimal value function is denoted by v_* and is defined as

$$v_*(s) \doteq \max_\pi v_\pi(s) \tag{16}$$

for all $s \in \mathcal{S}$ and the optimal action-value function is defined as

$$q_*(s,a) \doteq \max_{\pi} q_\pi(s,a) \tag{17}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$.

The optimal value function satisfies the Bellman Optimality Equation, which is given by

$$v_*(s) = \max_a \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma v_*(s')\right] \tag{18}$$

for all $s \in \mathcal{S}$. The optimal action-value function satisfies the Bellman Optimality Equation, which is given by

$$q_*(s,a) = \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma \max_{a'} q_*(s', a')\right] \tag{19}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$.

The Bellman Optimality Equation is a recursive equation and applying this equation gives simultanious nonlinear equations, which when solved gives the best optimal policy. But solving the Bellman Optimality Equation is computationally expensive and infeasible for large MDPs. Hence, we use iterative methods to solve the Bellman Optimality Equation.

4 Dynamic Programming

The key idea of Dynamic Programming in Reinforcement Learning is to use an iteratice process to change the values of the value functions to improve the policy. The process is called *Generalized Policy Iteration.*

4.1 Policy Iteration

4.2 Policy Evaluation (Prediction)

Policy Evaluation uses the equation of the Bellman Expectation Equation to estimate the value function of a policy. The Bellman Expectation Equation is given by

$$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma v_\pi(s')\right] \tag{20}$$

where $v_\pi(s)$ is the probability of taking the action a in the state s, following the policy π.

The policy evaluation finds the optimal value function of a policy by solving the Bellman Expectation Equation iteratively. The algorithm is given by

Algorithm 1 Policy Evaluation

```
1: Input $\pi$, the policy to be evaluated
2: Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
3: Initialize $V(s)$ arbitrarily, for $s \in \mathcal{S}$, and $V(terminal)$ to 0
4: repeat
5:     $\Delta \leftarrow 0$
6:     for each $s \in \mathcal{S}$ do
7:         $v \leftarrow V(s)$
8:         $V(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) [r + \gamma V(s')]$
9:         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
10:    end for
11: until $\Delta < \theta$
```

The policy evaluation algorithm, when used on a simple 4×4 grid world, converges to the optimal value function of the policy.

4.2.1 Policy Improvement

Policy Improvement is choosing the best action in each state greedily by choosing the action that maximizes the value function.

It follows from the idea that, if you select an action a from s and then follow the policy π and the return turned out to be better than all other actions, then it is better to choose the action a from the state s always.

Let the improved policy is π', if for all $s \in \mathcal{S}$, $q_\pi(s, \pi'(s)) \geq v_\pi(s)$. Then the policy π' is better than or equal to the policy π.

$$\pi'(s) \doteq \arg\max_a q_\pi(s, a) \tag{21}$$

$$v_{\pi'}(s) = \max_a \sum_{s',r} p(s', r \mid s, a) [r + \gamma v_{\pi'}(s')] \tag{22}$$

4.2.2 Policy Iteration

Once a policy has been evaluated and improved, the process is repeated until the policy cenverges to the optimal policy. This process is called Policy Iteration. We can show it as:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} v_* \tag{23}$$

The policy iteration algorithm is given by

Algorithm 2 Policy Iteration

```
Initialization:
$V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$; $V(terminal) \doteq 0$

Policy Evaluation:
repeat
    $\Delta \leftarrow 0$
    for each $s \in \mathcal{S}$ do
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s)) \left[r + \gamma V(s')\right]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    end for
until $\Delta < \theta$

Policy Improvement:
$policy\_stable \leftarrow true$
for each $s \in \mathcal{S}$ do
    $old\_action \leftarrow \pi(s)$
    $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma V(s')\right]$
    if $old\_action \neq \pi(s)$ then
        $policy\_stable \leftarrow false$
    end if
end for
if $policy\_stable$ then return $\pi$ and $V$
    break
else
    go to Policy Evaluation
end if
```

4.3 Jack's Car Rental Problem

Jack manages two locations for a nationwide car rental company. Each day, some number of customers arrive at each location to rent cars. If Jack has a car available, he rents it out and is credited \$10 by the national company. If he is out of cars at that location, then the business is lost. Cars become available for renting the day after they are returned. To help ensure that cars are available where they are needed, Jack can move them between the two locations overnight, at a cost of \$2 per car moved. We assume that the number of cars requested and returned at each location are Poisson random variables, meaning that the probability that the number is $\frac{\lambda^n}{n!}e^{-\lambda}$, where is the expected number. Suppose is 3 and 4 for rental requests at the first and second locations and 3 and 2 for returns. To simplify the problem slightly, we assume that there can be no more than 20 cars at each location (any additional cars are returned to the nationwide company, and thus disappear from the problem) and a maximum of five cars can be moved from one location to the other in one night. We take the discount rate to be = 0.9 and formulate this as a continuing finite MDP, where the time steps are days, the state is the number of cars at each location at the end of the day, and the actions are the net numbers of cars moved between the two locations overnight.

Policy Iteration is used to solve the Jack's Car Rental Problem. The policy evaluation is done using the Bellman Expectation.

The results obtained from the Policy Iteration are shown in the figure 7.

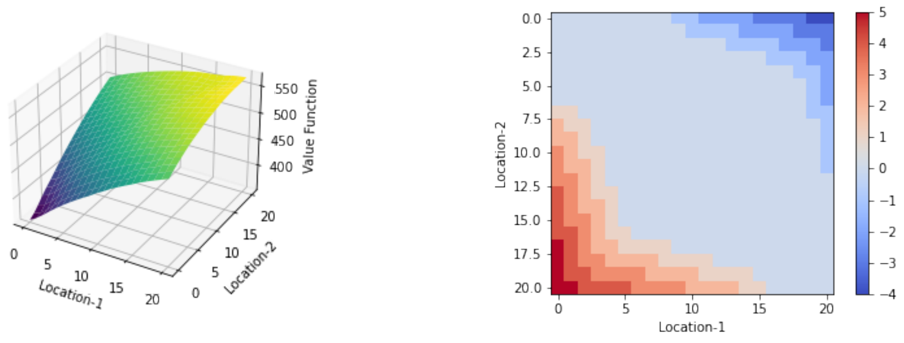

Figure 7: The results obtained from the Policy Iteration of the Jack's Car Rental Problem.

If the following modifications are made to the Jack's Car Rental Problem:

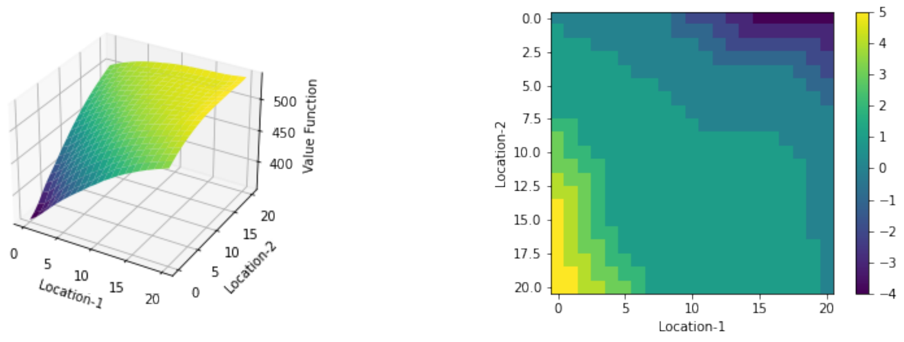

Figure 8: The results obtained from the Policy Iteration of the Jack's Car Rental Problem with the modifications.

The results obtained from the Policy Iteration of the Jack's Car Rental Problem with the modifications are shown in the figure 8.

The values represented in the right side of the figures in the number of cars to be moved from the first locaation to the second location to get the maximum returns.

4.4 Gambler's Problem

A gambler has the opportunity to make bets on the outcomes of a sequence of coin flips. If the coin comes up heads, he wins as many dollars as he has staked on that flip; if it is tails, he loses his stake. The game ends when the gambler wins by reaching his goal of \$100, or loses by running out of money.

On each flip, the gambler must decide what portion of his capital to stake, in integer numbers of dollars. This problem can be formulated as an undiscounted, episodic, finite MDP. The state is the gambler's capital, $s \in \{1, 2, ..., 99\}$ and the actions are stakes, $a \in 0, 1, ..., \min(s, 100 - s)$. The reward is zero on all transitions except those on which the gambler reaches his goal, when it is $+1$. The state-value function then gives the probability of winning from each state. A policy is a mapping from estimates levels of capital to stakes. The optimal policy maximizes the probability of reaching the goal. Let p_h denote the probability of the coin coming up heads. If ph is known, then the entire problem is known and it can be solved, for instance, by value iteration.

I tried the Policy Iteration to solve the Gambler's Problem for $p_h = 0.40$, $p_h = 0.25$, $p_h = 0.55$, and $p_h = 0.50$.

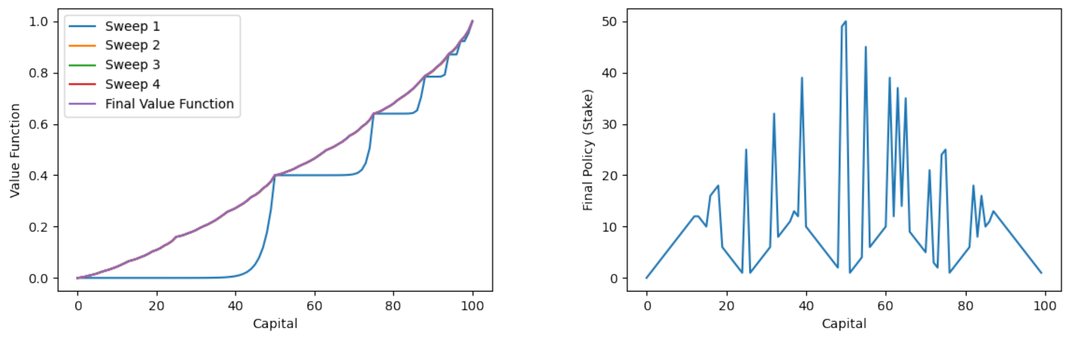

Figure 9: The results obtained from the Policy Iteration of the Gambler's Problem for $p_h = 0.40$.

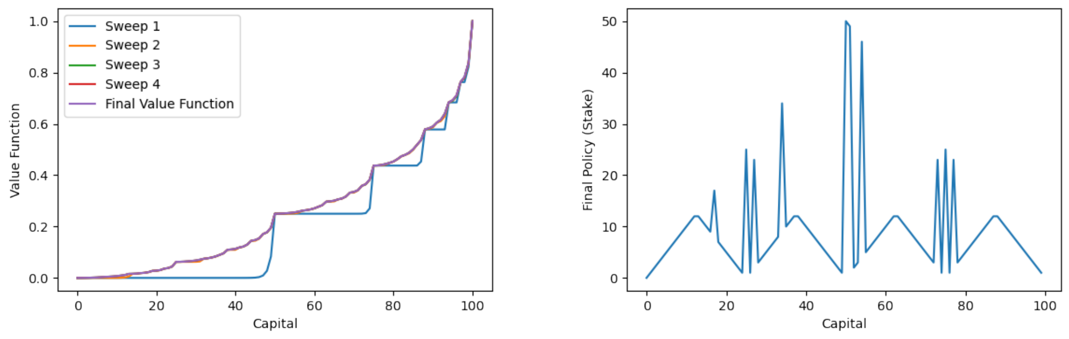

Figure 10: The results obtained from the Policy Iteration of the Gambler's Problem for $p_h = 0.25$.

Of course, the most beautiful graph is the one with $p_h = 0.55$. As expected, when $p_h \geq 0.50$, the engine tries to set the stake as just 1, because the expected value is positive and the maximum loss is minimised. This was something I learnt, because I would never use such a strategy myself. The regularities in the graphs for $p_h = 0.40$ and $p_h = 0.25$ are also interesting.

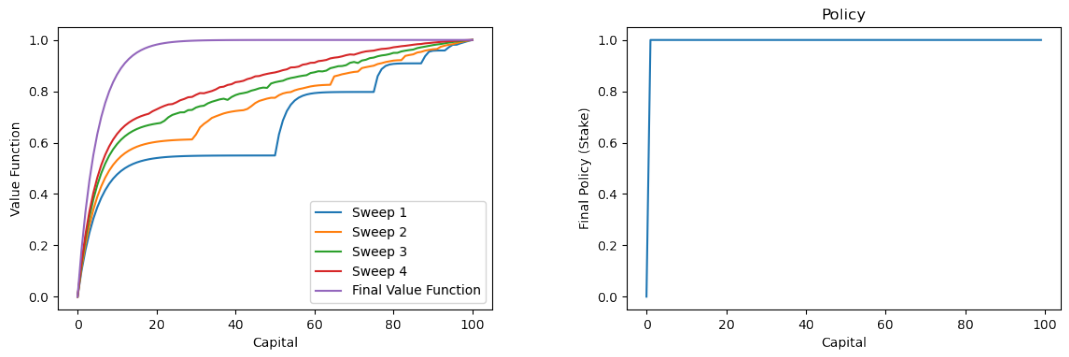

Figure 11: The results obtained from the Policy Iteration of the Gambler's Problem for $p_h = 0.55$.

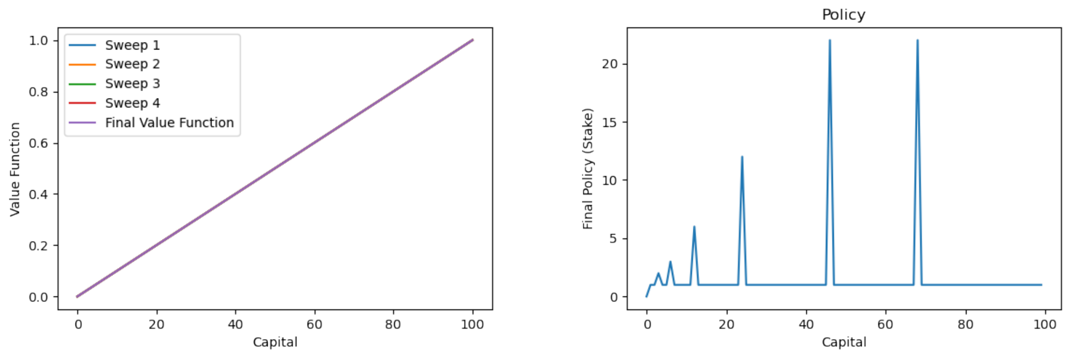

Figure 12: The results obtained from the Policy Iteration of the Gambler's Problem for $p_h = 0.50$.

5 Monte Carlo Methods

Monte Carlo methods are a class of algorithms that rely on random sampling to estimate the value function. These methods assume that the sample is huge enough to provide a good estimate on the value of the state.

This method is better than the Dynamic Programming methods when the model of the environment is not known. And, the Monte Carlo techniques update the value function after the episode ends, unlike the Dynamic Programming methods which update the value function after each step.

5.1 Monte Carlo Prediction

The Monte Carlo Prediction is the process of estimating the value function of a policy by averaging the returns observed after visits to the stare.

The Monte Carlo Predictions can be done in two ways:

- **First-visit Monte Carlo**: The value of a state is the average of the returns following the first time the state is visited in an episode.

- **Every-visit Monte Carlo:** The value of a state is the average of the returns following every visit to the state in an episode.

The algorithm for First-visit Monte Carlo is as follows:

Algorithm 3 First-visit Monte Carlo Prediction, for estimating V $\approx v_\pi$

```
Input: a policy $\pi$ to be evaluated
Initialize:
$V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$
while True (for each episode) do
    Generate an episode following $\pi$: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    For each step of the episode $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    if $S_t$ not in $S_0, S_1, \ldots, S_{t-1}$ then
        Append $G$ to $Returns(S_t)$
        $V(S_t) \leftarrow$ average($Returns(S_t)$)
    end if
end while
```

Every-visit Monte Carlo is similar to the First-visit Monte Carlo, but G is appended to $Returns(S_t)$ every time S_t is visited in an episode, instead of just the first-visit.

References

[1] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* 2nd. Cambridge, MA: MIT Press, 2018. ISBN: 978-0262039246.