A Brief Investigation Of The Gambler's Problem

Holt Spalding

, October 2, 2018

1. Background

1.1. The Problem

The goal of pretty much any reinforcement learning task (as with any control systems task), is to determine what actions an *agent* ought to take in its *environment* at any given time in order to maximize some cumulative reward. It is hard to formally define what an *agent* or its *environment* is, since this really depends on the task. Therefore, rather than describe the metatheory behind reinforcement learning, I think it would be easier to just describe the problem at hand and how RL can be used to solve it.

The Gambler's Problem is a classic reinforcement learning problem commonly attributed to Sutton & Barto[1]. The problem seeks to showcase how dynamic programming techniques can be used to solve a simple Markov decision process (MDP) task (more on this later). The problem goes a little something like this…imagine you are a gambler (our agent), and you are given the opportunity to bet on the outcome of a series of coin flips. You begin with some integer amount of money under 100$, and you are allowed to stake any amount up to the amount you already have on any given coin flip. If the outcome of a coin flip is heads, you receive the exact amount of money you put at stake. Otherwise, if the coin comes up tails, you lose as much money as you staked. You win the game once you have won 100$, and you lose the game if you run out of money. The problem asks us, what is the optimal bet on any given coin flip, given the amount of capital you already have. To solve this problem, we must first formally define an MDP …

[1]

1.2. The Problem As An MDP

A Markov Decision Process (MDP) is just one of many models of how an agent interacts with its environment over discrete time steps ($t = 1, 2, 3, ...$). MDPs provide us the mathematical framework by which we can solve many sequential decision making problems like the one described above. Formally, an MDP is defined in terms of three basic elements: a *state space* $\mathcal{S}$ which represents the set of all possible states of the environment, an *action space* $\mathcal{A}$, which represents the set of all possible actions the agent can take, and the set $\mathcal{R}$ which represents the set of all possible numerical rewards received as a result of performing some action. Given an MDP agent's current state (denoted S_t where t is the current time step), it must decide what action (A_t) to take in order to maximize cumulative rewards over time. Time steps are incremented by one every time an action is performed. Therefore, from here forward we let R_{t+1} and S_{t+1} represent the reward received and state reached at time step $t + 1$ after an action A_t is performed at time step t.

MDP agents decide what action to take in any given state by sampling a probability distribution known as a *policy*. Before the agent has learned optimal behavior, this distribution is completely arbitrary, and an RL algorithm must be simulated on the MDP in order for it to learn the optimal policy. In RL control tasks, this is achieved through *policy evaluation* and *policy improvement*. In *policy evaluation*, we seek to estimate the cumulative reward that would/will be received if we were to follow the trajectory of our current policy. In dynamic programming tasks, we assume a priori knowledge of the amount of reward that will be received given some action, and therefore, since we know what actions our agent will take in any given state under our policy, we can effectively estimate the cumulative reward that will be received given any arbitrary starting state. We let $v_\pi(s)$ denote the expected reward v that will be received if following a policy π starting from state s. In *policy improvement*, we seek to change our policy so we can take more optimal actions and maximize reward. The dynamic programming technique I've used in this experiment combines both policy improvement and policy evaluation (more on this later).

With this in mind, we can very easily reformulate the Gambler's problem in terms of an MDP. The state space consists of all potential amounts of capital our gambler can have at any given moment in time ($\{1, 2, 3, ..99\}$), the action space is set of all possible stakes that can be bet on any given coin flip ($\{0, 1, ..min(s, 100 - s)\}$ where s is the current state of the gambler, ie.

how much capital they have), and reward is only received once the game has been won (winning or losing money does not directly affect the outcome of the game unless the gambler gets 100$ or loses all their money). We need to find the optimal policy, ie. we need to find how much money should be staked given our amount of capital on any given time step in order to win the game. Also, from now on I shall let p_h denote the probability of landing heads on any given coin flip. With this formalization out of the way, we can now begin to discuss how I went about solving the problem.

2. Method

I solved this problem using a common dynamic programming technique called *value iteration*. In value iteration, we estimate the expected reward v of a given state s under our current policy $k+1$ as follows...

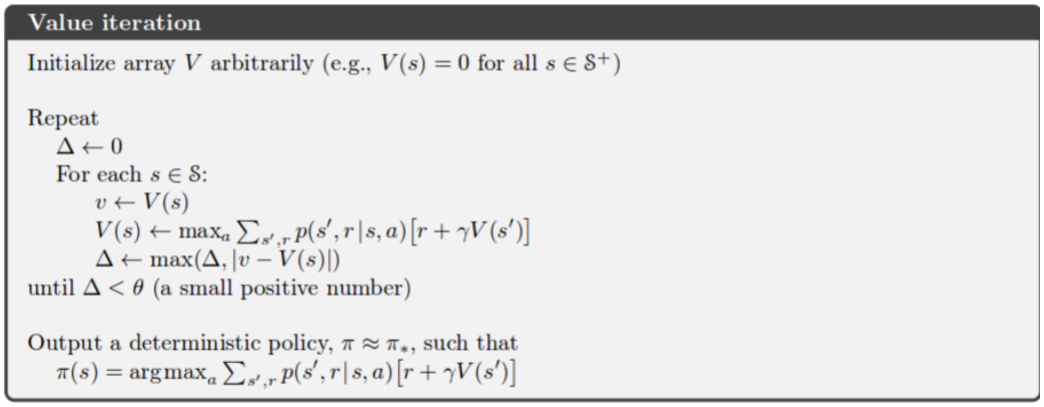
$$\begin{aligned} v_{k+1}(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_a \sum_{s',a} p(s', r | s, a)[r + \gamma v_k(s')] \end{aligned}$$

where s, r, and a represent singular rewards, and actions, respectively...

In the above, we can ignore the second line and focus solely on the first line (I've only featured the first line so the pseudocode below can be understood). You'll notice in the above that we've taken two things taken for granted. One is that we've assumed to know the action (given our state S_t) that maximizes expected reward at each successive time step. In the case of the Gambler's problem, we *do* know that value since we know the reward received if we win a coin flip (our capital + our stake), the reward if we lose a coin flip (our capital - our stake), and the probability our coin lands heads (for the curious, ignoring the gamma term, the expected reward would be denoted thusly: $p_h * (ourcapital + ourstake) + (1 - p_h) * (ourcapital - stake)$). Additionally, notice that when we are estimate the value of a state under our current policy, we are also potentially altering the action we would normally take under our current policy since we are choosing the action which maximizes $\mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a]$. Because of our gamma term in evaluating $v_{k+1}(s)$, we have effectively altered our policy merely by virtue of the fact that we've evaluated it. This update rule to our estimated state-value function makes perfect sense for the Gambler's problem, since we always seek to stake the amount (take the action) that maximizes expected reward, and expected reward is contingent on not only how much money we expect to win after one coin flip, but how much we expect to win in the immediate next state. In the case of the Gambler's problem, γ is simply equal to 1.

3. My Experiment

In my experiment, I solved two different instances of the Gambler's problem, one in which the probability of flipping heads was 0.55, and one in which the probability of flipping heads was 0.25. By the suggestion of Sutton & Barto, I utilized their simple value iteration algorithm described below . . .



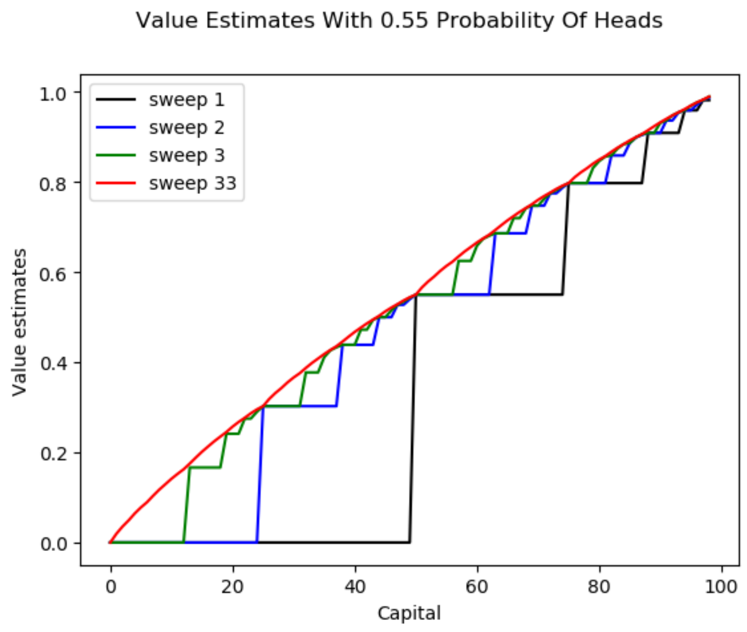
```
Value iteration
Initialize array $V$ arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

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

Output a deterministic policy, $\pi \approx \pi_*$, such that
   $\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
```

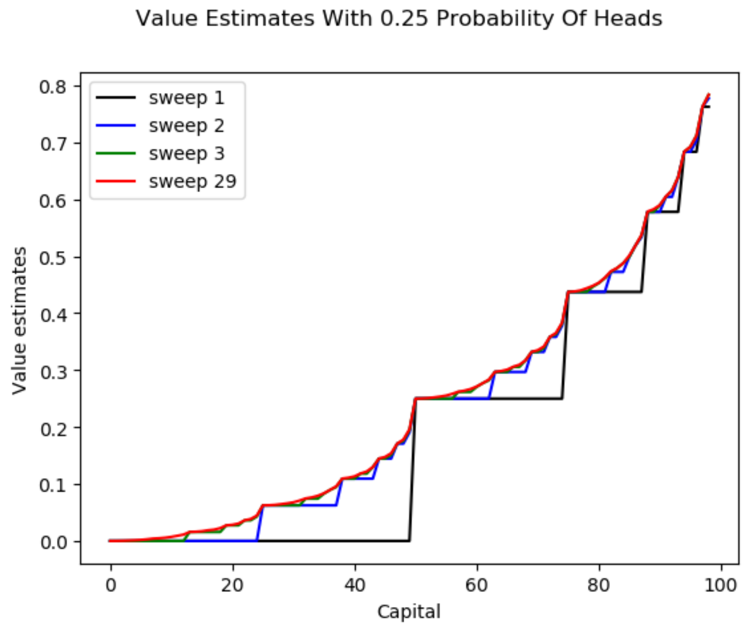
Figure 1: page 67

The notation should be fairly straightforward to understand given what I've already laid out in the other sections. At a high level, this algorithm simply seeks to update our state-value function given that we know the expected reward for each action and can therefore determine which action has maximum expected reward. Then, once we've sufficiently updated our estimated value of each state (and updates to these values fall below θ), our policy π is simply a deterministic policy which chooses which action to take in a state s on the basis of known expected reward r, and our estimated value of the immediate next state $V(s')$

4. Results

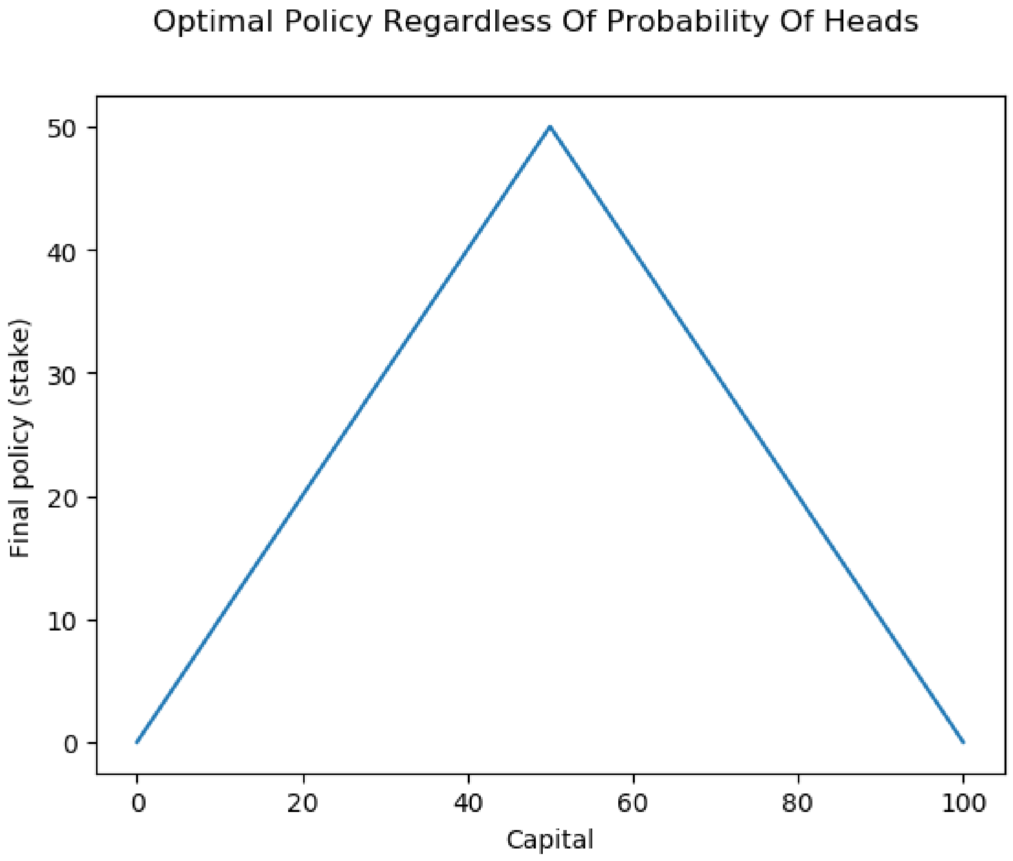
Below are the value functions for both experiments calculated by successive sweeps of value iteration





You'll notice that when there is a 0.55 probability of flipping heads, our value function is much more linear. This makes sense, because when the probability of winning or losing on a coin flip is almost 50-50 (ie. completely random), the value of having 50 dollars worth of capital is about half the value of having 100 dollars (which has value 1). The closer your capital gets to 100 dollars, the more value you are acrewing. However, when the odds are stacked against you, as is the case when your probability of landing heads is around 0.25, the value of any capital drops significantly. In other words, 50 dollars is worth much less if your chance of losing the money is much higher.

Below shows the optimal policy (ie the optimal amount of capital to put at stake) given your current amount of capital. This policy is optimal regardless of the probability of flipping heads. You'll notice the optimal amount to stake decreases once you have over 50$ in capital. That's because we are rewarded only for winning 100$, not necessarily more than that. With this policy, we always want to make the boldest bet possible in order to win the game. It may surprise you, but it's actually the correct result.



5. Brief Discussion

The result for optimal policy may seem rather strange, and I too thought it was strange, until a search through some of the literature on this problem revealed it was actually correct. Dubins and Savage (1960) [2] have mathematically proven that "*bold play*" is actually an optimal strategy for $p_h < 1$, but it's not the only optimal strategy, as is alluded to in chapter 4.5 of Sutton & Barto.

Li & Pyeatt (2005) [3] show a graph of their solution to the Gambler's

[2]

[3]

problem and also distinct from my solution and the solution provided Sutton & Barto. In fact, it's rather noisy due to ties that occur in the evaluation of optimal policy. They claim this noise is due to roundup errors and the floating point limitations of their CPU. Had I had the time, I would have liked to produce a secondary solution that looked more like Sutton & Barto's. However, it appears that it would have been quite difficult to achieve a cross platform solution to this problem (and also it didn't really seem necessary since my solution should also be optimal). This assignment really threw me for a loop, it's gotten me thinking about how my hardware has the potential to limit RL agents. In the future, I'd like to learn more about scientific computing, as I think this would give me some more insight into how functions converge when multiple solutions are availa

6. References

1. Sutton, R. S., & Barto, A. G. (1998). Reinforcement learning: An introduction. MIT press.

2. Dubins, L. E., Savage, L. J. (1960). OPTIMAL GAMBLING SYSTEMS. Proceedings of the National Academy of Sciences of the United States of America, 46(12), 15971598.
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC223086/pdf/pnas00211-0067.pdf

3. Li C., Pyeatt L. (2005) A Short Tutorial on Reinforcement Learning. In: Shi Z., He Q. (eds) Intelligent Information Processing II. IIP 2004. IFIP International Federation for Information Processing, vol 163. Springer, Boston, MA http://dl.ifip.org/db/conf/ifip12/iip2004/LiP04.pdf