Implementation of Reinforcement Learning Algorithms

Keya Desai
Rutgers University
Piscataway, NJ, USA
Email: keya.desai@rutgers.edu

Prakruti Joshi
Rutgers University
Piscataway, NJ, USA
Email: prakruti.joshi@rutgers.edu

***Abstract*—**
Reinforcement learning algorithms constitutes one of the three main machine learning paradigms. It has a wide range of applications as it forms a framework for learning sequential decision making tasks. In this project, we aim to understand and implement the major reinforcement learning algorithms: Dynamic Programming and Monte Carlo methods. The algorithms are trained and tested on pre-defined environments from the open source library Gym by Open AI. We will analyse the performance of the agent in different environments as the agent learns the appropriate actions at each state and maximizes the cumulative reward.
Index terms - reinforcement learning, Monte Carlo method, Gym, Policy iteration, value iteration

I. Project Description

This project falls under the category of Algorithmic Explorations (Type 3). We are proposing to study and implement reinforcement learning algorithms which constitutes a major sub-type of machine learning algorithms. Reinforcement learning is learning what to do given an environment and some kind of feedback mechanism from the environment. An RL algorithm learns how to map situations to actions, so as to maximize some kind of numerical reward signal in that environment. Basic reinforcement learning problem is modeled by defined states, possible actions from a state, transition probabilities from a particular state to another and the immediate reward on transitioning from one state to another. To solve a reinforcement learning problem, we need to find an optimal policy that determines an agent's action in a given state and results in a maximum expected reward value in the long run. Reinforcement learning algorithms use an underlying recursive algorithm of Bellman's equation to determine such policy and state-value functions over time.

In the project, we focus on understanding and implementing the algorithms of Dynamic Programming (Policy iteration and value iteration) and the Monte Carlo estimation methods. To thoroughly understand the concepts of these algorithms, we follow chapters 3-5 of the book- Introduction to Reinforcement learning by Sutton and Barto. We implement value iteration to solve Gambler's problem and Monte Carlo method to solve the game of Blackjack, as defined in the book.

Is it a novel idea? Reinforcement Learning has been studied extensively and there is substantial research being carried out in this field. Our aim is to gain an in-depth understanding of the basics of reinforcement learning first and then extend to a novel application or solve an appropriate real world problem.

What are the main stumbling blocks? Selecting an appropriate environment, one which is not too easy or not too difficult is one of the first stumbling blocks we encountered. One parameter in choosing the environment is that the environment should terminate since the Monte Carlo method will not work efficiently in an environment that may or may not terminate. Another potential challenge is to represent a real-world problem in the form that reinforcement learning algorithms understand. To define states, actions, and rewards for any problem. As a solution to this, we use predefined environments from the OpenAI Gym library [2]. It provides a vast collection of environments on which we can implement and analyze reinforcement learning algorithms. Lastly, understanding the nuances of reinforcement learning and analyzing its algorithms in a short span might turn out to be a challenging task.

Why is it useful? Although reinforcement learning is recent and tested mainly on toy domains, in the near future, it can be used to solve challenging real-world problems. Following are three applications where reinforcement learning is being applied:

1) Robotics - A robot can be trained to learn policies to map raw video images to a robotâs actions. The RGB images were fed to a CNN and outputs were the motor torques. The reinforcement learning component is the guided policy search to generate training data that came from its own state distribution.
2) Games - reinforcement learning has been used to solve many games such as Chess, Pac-Man, Asteroids and achieve superhuman performance in a few. The most famous one must be AlphaGo and AlphaGo Zero. AlphaGo, trained with countless human games, already achieved super-human performance by using value network and Monte Carlo tree search (MCTS) in its policy network
3) Health and Medicine - many reinforcement learning applications in health care mostly pertain to finding

optimal treatment policies. Recent papers cited applications of reinforcement learning to the usage of medical equipment, medication dosing, and two-stage clinical trials [3].

What is the timeline for your project progress?
Week of 4th Nov:

- Read and Research about reinforcement learning, Dynamic programming in RL and Monte Carlo.

Week of 11th Nov:

- Implement the policy iteration or value iteration algorithm of Dynamic programming of RL in Python. Implement this policy iteration algorithm for Gambler's problem to find the optimal policy to reach the goal.
- Analyse the results by varying the parameters.

Week of 18th Nov:

- Explore and understand the environments defined by Open AI gym.
- Implement the algorithm of Monte Carlo estimation method in Python, on the environment of BlackJack game to understand it better.
- Improve the monte carlo estimation by including epsilon-greedy policy approach.
- Analyse the results by plotting the output

Week of 25th Nov:

- Documentation and presentation.

How are you planning to reach the major milestones?

1) Refer to the book 'Introduction to Reinforcement Learning' by Richard Sutton and Andrew Barto.
2) Use the environments provided by Open Gym AI.
3) Divide tasks between team members efficiently.

Existing software libraries useful to exercise the chosen typical algorithm. Gym is a toolkit for developing and comparing reinforcement learning algorithms. The gym library is a collection of test problems i.e. environments that can be used to work out reinforcement learning algorithms. These environments have a shared interface, allowing to write general algorithms. For each environment, gym formulates the problem in a structure which reinforcement learning algorithms understand and use.

A. Stage1 - The Requirement Gathering Stage.

The general system description: The deliverable of the project is to implement the major reinforcement learning algorithms - Policy Iteration and value iteration, Monte Carlo estimation on the suitable environments and analysing the importance and drawbacks of these reinforcement learning algorithms. We will make the agent learn the optimal strategy in the game of Blackjack to maximize the score using Monte Carlo estimation and solve gambler's problem, i.e make the agent reach the goal of maximum money using value iteration.

In any reinforcement learning problem, the main goal is to learn the optimal value of being in a particular state and learning the optimal action to proceed while being in that state. The actions to follow from any state of the environment, in-order to maximize the total future rewards or to reach the final goal, is formally termed as a *policy*. Thus, finding the optimal state-value function and the optimal policy in an environment whose states and actions are defined is a typical reinforcement learning problem.

The user's interaction modes:

1) Selection of environment: The user selects the environment from Gym library on which reinforcement learning is applied. Features of the environment can be defined by the user.
2) Hyper parameters as inputs: Once an environment is selected, the user cannot change the environment. However, it can manipulate the hyper parameters of the algorithm such as the number of episodes, discounted rate factor ($beta$), alpha-factor in Q-learning, epsilon in Monte Carlo on-control method etc. Different observations can be determined by changing the hyper parameters.
3) Data access: The users can view parameters at any point in the functioning of the algorithm. It has access right to the parameters but cannot update them.

Output

1) **Demo:** We demonstrate the agent learning to solve the Gambler's problem using the Dynamic programming method of value iteration. And, the state - value function values and policies the agent takes for different probabilities of the coin turning up heads. We show the results for the optimal state-value functions and the policy that is learnt by the iterative algorithm. These essentially depict the stakes the Gambler should play given his current capital to reach his goal of a fixed capital.

   For the Monte Carlo estimation, we first demonstrate the simulated episodes on the environment of blackjack. An episode comprises of all the actions that the agent takes between the first state and the last or terminal state within the environment. We reinforce the agent to learn to perform the best actions in each state by experience by determining an optimal policy. We demonstrate the two variations of Monte Carlo - 1. state-value estimation and 2. on-control (epsilon greedy) estimation and show the estimate values of the state for different number of episodes. The Blackjack environment is taken from the Gym. [2]

2) **Analysis:** Analysis of the dynamic programming methods and Monte Carlo estimation methods in terms of time efficiency, complexity, constraints, convergence rate, problem-dependent parameters etc.

B. Stage2 - The Design Stage.

Elements of Reinforcement Learning

- *Value function:* Value functions are an estimate of how good it is for an agent to be in a given state. The goodness of being in a state is measured in terms of expected future rewards, which depends on the actions taken from that state. Hence, value functions are defined with respect to a policy that an agent follows in an environment.
- *Policy:* A policy π determines the action that an agent takes in an state of the environment. Formally, "a policy, π, is a mapping from each state, $s \in S$, and actions, $a \in A(s)$, to the probability $\pi(a|s)$ of taking action a when in state s. Thus, the value of a state s under a policy π, denoted by $v_\pi(s)$, is the expected return when starting in s and following π thereafter." [1]

For any policy π and any state s, the following consistency condition hold between the value of s and the value of its possible successor states:

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

γ is the discount rate. The discount rate determines the present value of the future rewards: a reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately. $p(s'|s,a)$ is the transitional probability and $r(s,a,s')$ is the expected value of the next reward, given any current state and action s and a, together with any next state s'. Equation-1 is the *Bellman equation* for v_π. It expresses a relationship between the value of a state and the values of its successor states [1].

Next, we describe two fundamental classes of methods for solving finite Markov decision problems in reinforcement learning - dynamic programming and Monte Carlo methods. A Markov decision process (MDP) is essentially defined by four attributes: states in an environment, the possible actions in a given state, the transition probabilities of moving from one state to another by taking a particular action, an immediate reward gained by transitioning from a particular state to another by taking action a. These encapsulated together form the MDP environment. The main property of the MDP is that it satisfies the Markov property i.e. given a state and the action, the next state determined is independent of all the previous states and actions. Thus, the core problem in a MDP problem boils down to determining the optimal policy which indirectly depends on the state value function. We discuss the different approaches to find this optimal policy.

1. Dynamic Programming

Dynamic Programming is method for solving complex problems by breaking the problem into smaller sub-problems. Dynamic Programming can be used to solve problems which have two properties, *optimal substructure* (Bellman equation) and overlapping sub problems (value function - store of knowledge that can be re-used). Markov Decision Processes satisfy both these properties. Hence, if we have a perfect model of the environment as a Markov Decision Process, it can be used to compute optimal policies. If we have the knowledge about the transition probabilities, then we can use the policy iteration or value iteration to determine the optimal policy by using the Bellman equation in 1. Following is one of the dynamic programming algorithms used to determine the optimal policy evaluated from the optimal value function. [1]

Algorithm 1: Policy Iteration

```
1. Initialization
  $v(S) \in \mathbb{R}$ and $\pi(S) \in$ A(s) arbitrarily for all s $\in S^+$

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

3. Policy Improvement
  policy-stable $\leftarrow true$
  For each s $\in S^+$:
    $temp \leftarrow \pi$(s)
    $\pi$(s) $\leftarrow$ arg $\underset{a}{max} \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma v(s')]$
    If $temp \neq \pi(s)$, then policy-stable $\leftarrow false$
  If policy-stable, then stop and return $v$ and $\pi$;
  else go to 2
```

Suppose we have determined the value function v_π for an arbitrary deterministic policy π. For some state s we would like to know whether or not we should change the policy to deterministically choose an action $a \neq \pi(s)$. One way to answer this is to evaluate the value of taking action a in state s under a policy π, denoted by $q_\pi(s,a)$ as the expected return starting s, taking the action a, and thereafter following π.

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

To choose the best policy, we need to consider changes at all states and to all possible actions, selecting at each state the action that appears best according to $q_\pi(s,a)$. In other words, to consider the new greedy policy, π', given by

$$\begin{aligned} \pi'(s) &= \underset{a}{argmax}\, q_\pi(s,a) & (3) \\ &= \underset{a}{argmax} \sum_{s'} p(s'|s,a)[r(s,a,s' + \gamma v(s')] & (4) \end{aligned}$$

where arg $\underset{a}{max}$ denotes the value of a at which the expression that follows is maximised.The process of making a new policy that improves on an original policy, by making it greedy with respect to the value function of the original policy, is called policy improvement.

Once a policy π, has been improved using v_π to yield a better policy, π', we can then compute v'_π and improve it again to yield an even better policy π''. We can thus obtain a sequence of monotonically improving policies and value functions:

$$\pi_0 \overset{E}{\rightarrow} v_{\pi 0} \overset{I}{\rightarrow} \pi_1 \overset{E}{\rightarrow} v_{\pi 1} \overset{I}{\rightarrow} \pi_2 \overset{E}{\rightarrow} ... \overset{I}{\rightarrow} \pi_* \overset{E}{\rightarrow} v_*,$$

where $\xrightarrow{E}$ denotes a policy evaluation and $\xrightarrow{I}$ denotes a policy *improvement*. "Each policy is guaranteed to be a strict improvement over the previous one(unless it is already optimal). Because a finite Markov Decision Process has only a finite number of policies, this process must converge to an optimal policy and optimal value function in a finite number of iterations. This way of finding an optimal policy is called *policy iteration*"[1].

Value Iteration:

Value iteration is another dynamic programming algorithm which provides an improvement over the drawback of policy iteration of iterative computation required for multiple passes of the entire state set. The value iteration is simply an algorithm which uses the Bellman equation-1 as an update rule. This is now similar to the *policy evaluation* except that it takes the maximum over all possible actions. The termination condition for this iterative update is similar to that of policy iteration by setting a critical value. Once, these optimal state-values are determined by using value iteration, a policy can be set by selecting the action which maximizes the expected reward. Effectively, value iteration often gives faster convergence than policy iteration. We have trained the agent to solve Gambler's problem using value iteration.

Algorithm 2: Value Iteration

```
Initialize array $v$ arbitrarily( e.g.,$v(s) = 0$ for all
  s $\in S^+$)

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

Output a deterministic policy, $\pi$, such that,
    $\pi$(s) $\leftarrow$ arg$\underset{a}{max} \sum_s p(s'|s,a)[r(s,a,s') + \gamma v(s')]$
```

Complexity Analysis: In iterative policy evaluation, for each state, the policy evaluation iterates through all the states and the policy improvement iterates through all the actions. The step of policy evaluation will continue till convergence and hence its complexity will be O(|Episodes till convergence| $\times$ |States|^2). Value iteration on the other hand, for each state, calculates the expected reward for each action and selected the maximum one. This will go on till the state values converge. Hence, its complexity will be O(|Episodes till convergence| $\times$ |States| $\times$ |Actions|).

2. Monte Carlo Methods

While solving a problem using dynamic programming, we assume that we have complete knowledge about the environment - the transition probabilities, reward values. Generally, this complete knowledge about the environments is not available. The other subclass of reinforcement learning algorithm- Monte Carlo methods require only experience. "Experience here means the sample sequences of states, actions, and rewards from simulated interaction with an environment". [1] Thus, this method does not require any prior knowledge about the environmentâs dynamics i.e. probability information and the gains. This provides a method of solving reinforcement learning algorithm based on average returns. As more returns are observed, the average should converge to the expected value by the law of large numbers. This is the fundamental notion behind all the Monte Carlo methods.

"Monte Carlo methods are defined for episodic tasks - experience which can be divided into episodes and each episode eventually terminates no matter the sequence of actions. This ensures that in each episode, well defined returns are available. It is only upon the completion of an episode that value estimates and policies are changed. Monte Carlo methods are thus incremental in an episode-by-episode sense, but not in a step-by-step sense." [1]

The Monte Carlo methods for learning the state-value function for a given policy is estimated from experience as explained above. The state-value functions are approximated to the average of the returns observed after visits to that state in a particular episode. The value function can be estimated in two ways:

1) Every-visit MC: Each occurrence of state s in an episode is called a visit to s. The every-visit MC method estimates v(s) as the average of the returns following all the visits to s in a set of episodes.
2) First-visit MC: Within a given episode, the first time s is visited is called the first visit to s. The first-visit MC method averages just the returns following first visits to s.

Both first-visit MC and every-visit MC converge to v(s) as the number of visits (or first visits) to s goes to infinity (for practical purposes, we simulate the number of episodes as a large number). We implement the first-visit MC to estimate the state-value function. The pseudo-code for the first-visit MC estimation method is as shown:

Complexity Analysis: In first-visit Monte Carlo method, initialization takes constant time. For each episode, the return

Algorithm 3: First - visit Monte Carlo method for estimating v_π

```
Initialize
    $\pi \leftarrow$ policy to be evaluated
    $V \leftarrow$ an arbitrary state-value function
    $Returns(s) \leftarrow$ an empty list, for all $s \in S^+$

Repeat forever:
    (a) Generate an episode using $\pi$
    (b) For each state $s$ appearing in the episode:
        $G \leftarrow$ return following the first occurrence of $s$
        Append $G$ to *Returns(s)*
        *V(s)* $\leftarrow$average(*Returns(s)*)
```

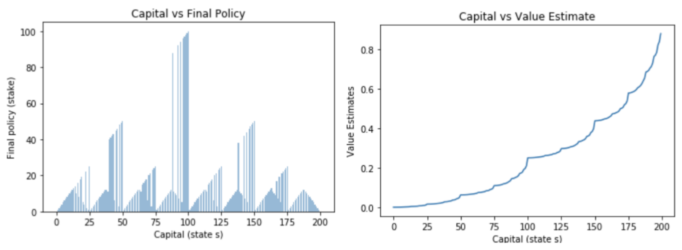
is calculated for every state. Hence, its complexity will be O(|Episodes| $\times$ |States|)

In Monte Carlo estimation described above, there exists a problem of exploring starts. Maintaining exploration v/s exploitation is an essential feature of reinforcement learning algorithm. To ensure that all the actions are selected infinitely often during monte carlo estimation, we have used the epsilon-greedy approach. The only difference in this method is that we tweak the policy such that with ϵ probability we select an action at random to explore and with 1-ϵ probability we select the action that the policy greedily determines to produce the maximum expected rewards. The convergence in this case takes slightly more time than the previous method. However, the estimates for the state-value functions for all the states and actions improve over time.

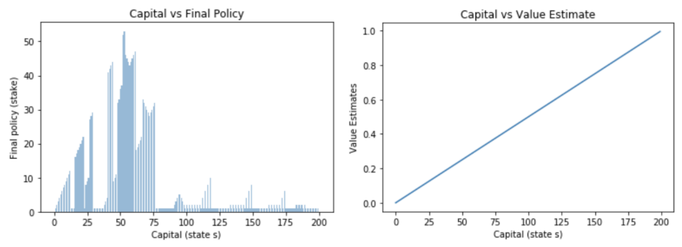
C. Stage3 - The Implementation Stage.

1. Implementation of Value Iteration

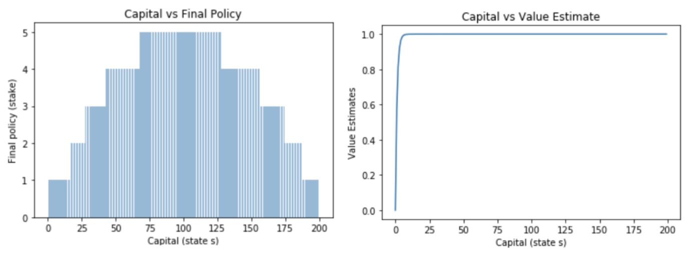
Gambler's Problem: **(As defined in Section 4.4 of [1])**
A gambler has the opportunity to make bets on the outcomes of a sequence of coin flips. If the coin comes up heads, he wins as many dollars as he has staked on that flip; if it is tails, he loses his stake. The game ends when the gambler wins 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, $s \in \{0, 1, ..., min(s, 100 - s)\}$. 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 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 p_h is known, then the entire problem is known and it can be solved, for instance, by value iteration.



((a)) p_h = 0.25



((b)) p_h = 0.50



((c)) p_h = 0.70

Figure 3: Capital(State) vs Final Policy(stake) and Capital(State) vs Value estimate for different p_h values

Implementation: [1] We have implemented value iteration algorithm to determine the state-value of each state(Gambler's capital). The user-control parameters are: g - goal capital (\$200 here), *gamma* - discount factor, *theta* - convergence threshold and p_h - the probability of coin flip to turn heads.

The value should increase as the capital increases and reaches closer to the goal (g parameter in the implementation). The algorithm converges to give the optimal stake (action determined by policy) for a particular capital (state).

Results:
The plots in the Figure 1 show the optimal policy and state-function value for different probabilities of the coin flip to turn head. In all cases, the value function of being in a state closer to our goal capital should increase which is depicted in the "Captial vs Value estimate" in the plot. Since the reward of the goal capital (here \$ 200) is 1 and 0 otherwise, the value function should converge to 1 eventually. The smoothness of the increase in the value function is dependent on the probability of heads. When the probability of heads is high (here 0.75), which in turn implies that the probability of Gambler winning the stake is high, the value function rapidly converges

[1]https://tinyurl.com/whj5fcz

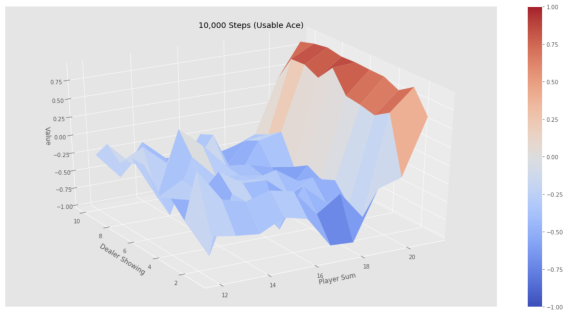
to 1 reflecting the same essence that all the states(Gambler's capital) have high value of the Gambler winning the stake. When the probability of the heads is low (here 0.25), the value function is a bit wavy and gradually converges to 1.

The first plot depicts the optimal policy determined from these converged values of state-value function. It shows the optimal stake that a Gambler should play given that state of his capital. The optimal policy is most random in the case of the probability of heads being 0.5. This is natural as the odds are neither in the favor of the Gambler nor against him. Thus, it introduces maximum entropy in the result at each stake. On the other hand, whenever the probability is biased towards or against the Gambler (in case of p_h being 0.25 or 0.75), the optimal policy of strategy to stake is quite uniform. Particularly, when the probability is 0.75, the Gambler safely stakes in the range of (0-5) and steadily increases his capital to reach the final goal.

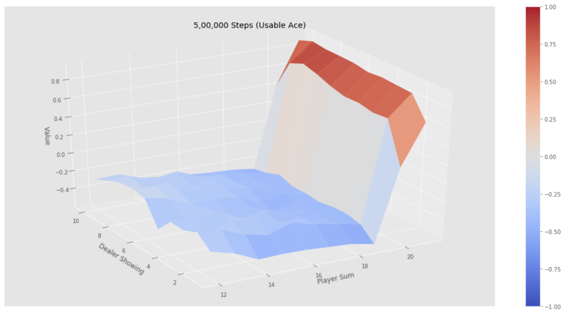
2. Implementation of Monte Carlo method

***Blackjack* (As defined in Section 5.1 of [1])**

Blackjack is a popular casino card game. The object is to obtain cards the sum of whose numerical values is as great as possible without exceeding 21. All face cards count as 10, and the ace can count either as 1 or as 11. We consider the version in which each player competes independently against the dealer. The game begins with two cards dealt to both dealer and player. One of the dealerâs cards is faceup and the other is facedown. If the player has 21 immediately (an ace and a 10-card), it is called a natural. He then wins unless the dealer also has a natural, in which case the game is a draw. If the player does not have a natural, then he can request additional cards, one by one (hits), until he either stops (sticks) or exceeds 21 (goes bust). If he goes bust, he loses; if he sticks, then it becomes the dealerâs turn. The dealer hits or sticks according to a fixed strategy without choice: he sticks on any sum of 17 or greater, and hits otherwise. If the dealer goes bust, then the player wins; otherwise, the outcomeâwin, lose, or drawâis determined by whose final sum is closer to 21. Playing blackjack is naturally formulated as an episodic finite MDP. Each game of blackjack is an episode. Rewards of +1, -1, and 0 are given for winning, losing, and drawing, respectively. All rewards within a game are zero, and we do not discount (γ = 1); therefore these terminal rewards are also the returns. The player's actions are to hit or to stick. The states depend on the player's cards and the dealerâs showing card. We assume that cards are dealt from an infinite deck (i.e., with replacement) so that there is no advantage to keeping track of the cards already dealt. If the player holds an ace that he could count as 11 without going bust, then the ace is said to be usable. In this case it is always counted as 11 because counting it as 1 would make the sum 11 or less, in which case there is no decision to be made because, obviously, the player should always hit. Thus, the player makes decisions on the basis of three variables: his current sum (12-21), the dealerâs one showing card (ace-10), and whether or not he holds a usable ace. This makes for a total of 200 states.



((a)) 10,000 steps



((b)) 500,000 steps

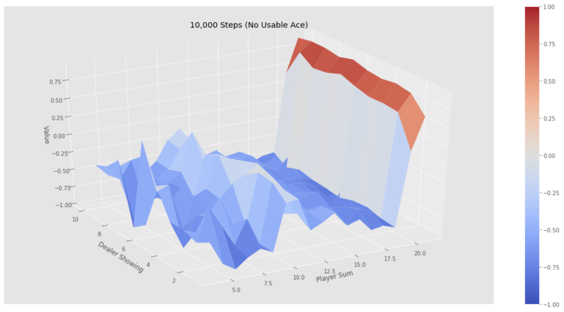
Figure 2: Approximate state-value functions for the blackjack policy that sticks only on 20 or 21, computed by Monte Carlo policy evaluation.

Implementation: [2] We have set the policy as follows: stick if the sum exceeds 20 and hit otherwise. We estimate the state-value function by running the simulation for 10k and 50k episodes.

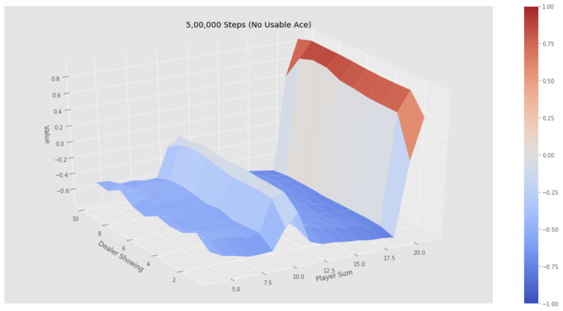
Implementation of $\epsilon - greedy$ [3]: We also implemented the epsilon-greedy policy approach to overcome the problem of exploring starts. ϵ value is set as 0.1.

Results: We obtained the estimates of the state-value function shown in Figure 2 for 10k steps and 50k steps. In any event, after 500,000 episodes the value function is better approximated than using 100,000 episodes. These follows the intuition behind the Monte Carlo estimation that more the experience, more the agent is learning about the environment dynamics. For plotting the state-value function, we have reused the code from GitHub repository of Dennybritz, which has a MIT licence. [5]. Figure 3 shows the state-value functions when the epsilon-greedy approach is inculcated along with Monte Carlo state-estimation. The overall estimation for the all the state improves.

[2]https://tinyurl.com/w7yqw9b
[3]https://tinyurl.com/qk3jp3m



((a)) 10,000 steps



((b)) 500,000 steps

Figure 3: Approximate state-value functions for the blackjack policy that sticks only on 20 or 21, computed by Monte Carlo on policy method.

3. Conclusion:

We started by understanding the fundamental components of Reinforcement learning. We explored the different environments in which a Reinforcement learning problem can be defined and experimented with the Open AI Gym platform and chose to implement the algorithms based on Markov decision process(MDP) environment. Dynamic programming in reinforcement learning has two algorithms: policy iteration and value iteration which are useful when the entire dynamics of the environment (the transition probabilities and the immediate rewards) are known. Monte Carlo covers up for the shortcoming of Dynamic programming when the complete knowledge about the environment is missing. It simulates large number of episodes and estimates the environment dynamics. The implementation is carried out on the environment of Gambler's problem and the game of Blackjack. In each case, the optimal policy and the state-value functions are determined.

References

[1] Richard S. Sutton, Andrew G. Barto, Reinforcement Learning: An Introduction, Massachusetts: The MIT Press, 2012
[2] GYM Library for OpenAI. gym.openai.com/.
[3] Lorica, Ben. "Practical Applications of Reinforcement Learning in Industry." O'Reilly Media, 14 Dec. 2017, www.oreilly.com/radar/practical-applications-of-reinforcement-learning-in-industry/.
[4] Ashraf, Mohammad. "Reinforcement Learning Demystified: Solving MDPs with Dynamic Programming." Medium, Towards Data Science, 17 Dec. 2018, towardsdatascience.com/reinforcement-learning-demystified-solving-mdps-with-dynamic-programming-b52c8093c919.
[5] Dennybritz. "Dennybritz/Reinforcement-Learning." GitHub, 9 Nov. 2019, github.com/dennybritz/reinforcement-learning.