ENSAE ParisTech

Master 1st year Python Machine Learning Project

On-Policy Reinforcement Learning for Blackjack

by Mehdi Abbana Bennani

supervised by
Pr. Xavier Dupré

December 23, 2016

Abstract

Reinforcement Learning is a major Machine Learning class of algorithms. In this report, I will apply some major RL algorithms to a simplified blackjack game, mostly inspired form the Easy21 Assignment by Prof. David Silver at UCL. I will demonstrate through simulations that these algorithms achieve a good performance in this framework, and I will show some limits of these algorithms.

Contents

Introduction

The report is organized as follows:in the first part, I define the problem, in the second part I describe the reinforcement learning terminology and the algorithms I will use. In the third part, I will describe some important coding aspects such as code structure and performance improvement tricks.

Motivation

Reinforcement learning is a one of the major Machine Learning classes of algorithms. It achieved a better performance than many state of the art algorithms in many domains, such as in games [2] [4], shape recognition [3], and it is used for some very complex problems such as driveless cars.

Through this assignment, I aim to:

- explore and understand Reinforcement Learning Algorithms
- apply these algorithms to concrete problems and explore their limits
- test new approaches and personal ideas in these problems

Running the simulations

The source code is available under my GitHub repository :

https://github.com/MehdiAB161/Reinforcement-Learning.git

The running instructions are provided within the Readme file

1 Problem definition

1.1 Simple case

This exercise is similar to the Blackjack example in Sutton and Barto 5.3, however, the rules of the card game are different and non-standard.

- The game is played with an infinite deck of cards (i.e. cards are sampled with replacement).
- Each draw from the deck results in a value between 1 and 10 (uniformly distributed) with a colour of red (probability 1/3) or black (probability 2/3).
- There are no aces or picture (face) cards in this game
- At the start of the game both the player and the dealer draw one black card (fully observed)
- Each turn the player may either stick or hit. If the player hits then she draws another card from the deck. If the player sticks she receives no further cards
- The values of the player's cards are added (black cards) or subtracted (red cards).
- If the player's sum exceeds a value n, or becomes less than 1, then she "goes bust" and loses the game (reward -1)
- If the player sticks then the dealer starts taking turns. The dealer always sticks on any sum of n - 4 or greater, and hits otherwise.
- If the dealer goes bust, then the player wins; otherwise, the outcome – win (reward +1), lose (reward -1), or draw (reward 0) – is the player with the largest sum.

2 Forumlation of Reinforcement Learning Problem

2.1 Definitions

Definition 1 *State Value Function*
The state value function v(s) of a Markov Reward Process is the expected return starting from state s

$$v(s) = \mathbb{E}[G_t|S_t = s]$$

Definition 2 *Policy*
A policy π is a distribution over actions given states

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$$

Definition 3 *Return*
The return G_t is the total discounted reward from time step t.

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Definition 4 *State Value Function*
The state value function $v_\pi(s)$ of a MDP is the expected return starting from state s, and then following the policy π

$$v_\pi(s) = \mathbb{E}[G_t|S_t = s]$$

Definition 5 *State Action Value Function*
The state action value function $q_\pi(s, a)$ of a MDP is the expected return starting from state s, and taking action a, then following the policy π

$$v_\pi(s) = \mathbb{E}[G_t|S_t = s]$$

The state action value function $q_\pi(s, a)$ of a MDP is the expected return starting from state s, and taking action a, then following the policy π

$$q_\pi(s) = \mathbb{E}[G_t|S_t = s, A_t = a]$$

Definition 6 *Optimal Value Function*
The optimal state action value function $q_\star(s, a)$ is the maximum state action value function over all policies

$$q_\star(s, a) = \underset{\pi}{max} q_\pi(s, a)$$

Definition 7 *Optimal Policy*
We define a partial ordering over a policies:
$\pi \geq \pi'$ *if* $v_\pi(s) \geq v_\pi(s)$

2.2 Theorems

Theorem 1 *Optimal Policy For any MDP*

- *There exists an optimal policy $\pi_\star$ that is better than or equal to all other policies $\pi_\star \geq \pi, \forall \pi$*
- *All policies achieve the optimal value function $v_{\pi_\star} = v_\star$*
- *All policies achieve the optimal state action value function $q_{\pi_\star} = q_\star$*

Theorem 2 *The Bellman Expectation Equation*

The state-value function can be decomposed into immediate reward plus discounted value of the successor state.

$$v_\pi(s) = \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s]$$

The action-value function can similarly be decomposed

$$q_\pi(s) = \mathbb{E}[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a]$$

2.3 Monte Carlo Algorithm

The intuition behind this algorithm is that, in order to estimate the value of a state action pair, we run a full episode, with a policy such as ϵ greedy policy,then we update our current estimation of the action-value function in the direction of the average of the action value estimation of all visited state-action pairs during the episode.

Algorithm 1 Greedy in the Limit with Infinite Exploration Algorithm (GLIE)

Sample kth episode using $\pi : \{S_1, A_1, R_1, S_2, ...\}$
for each state S_t and Action A_t in the episode **do**
    $N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{G_t - Q(S_t, A_t)}{N(S_t, A_t)}$
Improve policy based on new action-value function
$\epsilon = 1/k$
$\pi \Leftarrow \epsilon - greedy(Q)$

2.4 SARSA Algorithm

The intuition behind this algorithm is that, in order to estimate the value of a state action pair, we run a full episode, with a policy such as ϵ greedy policy,then we update our current estimation of the action-value function in the direction of the average of the action value estimation of all visited state-action pairs during the episode.

Algorithm 2 SARSA Algorithm for On-Policy Control

Initialize $Q(s, a), \forall s \in \mathbb{S}, a \in \mathbb{A}(s)$, arbitrarlily, and Q(terminal-state,-)=0
for each episode **do**
    Initialize S
    Choose A from S using policy derived from Q (e.g., $epsilon - greedy$)
    for each step in the episode **do**
        Take action A, observe R, S'
        Choose A' from S'
        $Q(S, A) \leftarrow Q(S, A) + \alpha(R + \gamma Q(S', A') - Q(S, A)$
        $S \leftarrow S'$
        $A \leftarrow A'$

2.5 SARSA-λ Algorithm

In the SARSA algorithm, the agent goes just one step away from his states and then he bootstraps using his estimate of the action value function. We can imagine algorithms which go n steps and then bootstrap. For the case of the sarsa lambda algorithm, he averages over all the updates with from 1 step to the maximum number of steps with a geometric weight. In other words, the estimate using one step sarsa has a weight of $\frac{\lambda}{1-\lambda}$, then the estimate of n step sarsa has a weight of $\frac{\lambda^n}{1-\lambda}$. The range of λ is [0,1], so if we take a small λ, we are closer from TD learning with a few steps, and if λ is close to 1, the states which are very far have almost a weight of one, therefore, it is almost Monte Carlo Learning. So the λ is a cursor between TD Learning and Monte Carlo Learning.

2.6 Action Value Function Approximation

Where the state action space size is very big, we cannot use table lookup anymore to get the state action value function values for all states. In this case, we approximate the value function by parametrizing it. The state

Algorithm 3 SARSA-λ Algorithm for On-Policy Control

Initialize $Q(s,a), \forall s \in \mathbb{S}, a \in \mathbb{A}(s)$, arbitrarily, and Q(terminal-state,-)=0
for each episode **do**
    E(s,a) $= 0$ for all $s \in \mathbb{S}, a \in \mathbb{A}(s)$
    Initialize S, A
    for each step in the episode **do**
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q (e.g. ϵgreedy)
        $\delta \leftarrow R + \gamma Q(S', A') - Q(S, A)$
        $E(S, A) \leftarrow E(S, A) + 1$
        for all $s \in S, a \in \mathbf{A}(s)$ **do**
            $Q(s, a) \Leftarrow Q(s, a) + \alpha\delta E(s, a)$
            $E(s, a) \leftarrow \gamma\lambda E(s, a)$
        $A \leftarrow A', S \leftarrow S'$,

action value function becomes $Q(S, A, w)$, where w is a vector which size is the feature space size.

In this case, we update the parameters w instead of updating every single state action pair value function. This approach also has some upsides such as approximating the value of state-actions that may be never visited.

In case of the Monte Carlo 'GLIE' algorithm, we operate the following update:

$$\Delta\mathbf{w} = \alpha(G_t - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(S_t, A_t, \mathbf{w}) \tag{1}$$

This equation means that we make a step in direction of the difference of our estimate and the sampled value with Monte Carlo.

For TD$-\lambda$, the update is as follows:

$$\Delta\mathbf{w} = \alpha(R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w})) - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(S_t, A_t, \mathbf{w})) \tag{2}$$

This equation means that we make a step in direction of the difference of our estimate and the sampled value with Sarsa λ.

3 Reinforcement Learning applied to Blackjack

3.1 Experimental results

I applied the Algorithms above to the simplified blackjack game defined in the first section. For the value function approximation, the features are binary considering their belonging to the following overlapping intervals of the following cuboid: dealer(s) = [1, 4], [4, 7], [7, 10] player(s) = [1, 6], [4, 9] ... action = hit, stick

Case the score upper limit is 21

The first case is when I set the score upper limit to 21. Therefore the state space size is 210. I expect the Monte Carlo estimation to be the most stable algorithm, and is unbiased asymptotically.

I infered the state value function from the state action value function as the the value of the action which brings the most value from state s $V(s, a) = argmaxQ(s, a)$ The results are shown in the figure ... In this figure, we can see that we can identify the mechanisms of the game. The value function is decreasing with the card value of the dealer. Also the value 10 is relatively safe for the player because he is sure not to lose the game, from 17 to 21 ....

The figure ... shows the value function using Sarsa lambda, in this figure we see that our estimate is more noisy than in the previous one. The reason is that Sarsa lambda is based on bootstrapping, therefore our updates are biased. I also expected Sarsa lambda to be slower in term of convergence speed, we can see that on the rmse figure.

For this application, value function approximation is not relevant because the state space is very small the size is equal to scoreLimit * 10, which in our case corresponds to 520 states. The chosen features' space size is ....

First, I used linear function approximation, which I have theoretical convergence guaranties The result is displayed in figure ... We can see that

The RMSE is

The other function approximation I used is quadratic approximation, I don't have theoretical convergence guaranties. The results are displayed in ....

Case the upper limit is 99

In this case the state space size is close from 1000. However, the biggest challenge is reaching some states because there is a low probability of reaching them. Those states are those which the player has a very high score in. Therefore, the variance of the obtained results increases with the player score. The figures ... show these results, for the same number of episodes as the previous case.

One possible solution is allowing the player to start at t=0 from a state with a higher score than 0 and 10. However, this will have a negative impact on the policy the agent will learn because the policy won't take into account the low probability of reaching the states with very high scores.

Another simple solution is increasing the number of episodes, however this becomes computationally expensive. In case of multiplying the number of episodes by 10, we obtain the figure, the computation time is also multiplied by 10. This solution is intractable for higher dimension state spaces.

3.2 Further analysis

In case of a very high dimensional state space, for example a finite card set without re-sampling and by integrating a memory to the agent. It is not possible to use table lookup, therefore the basic Monte Caro GLIE and Temporal difference algorithms are intractable, $n\sum_{k=1}^{n} C_n^k$ which is $O(n2^n)$. State Action value function approximation is necessary.

As said before, the linear function approximator has theoretical garantees of convergence, which is not the case of other function approximation such as neural networks, which require other algorithms which proved to be stable to some very complex problems, such as Deep Q Networks and Experience Replay [2].

In the case of using some simpler function approximation such as linear or quadratic function approximation, the number of necessary episodes depends on the size of the feature space, also on the state space size. A direct relation between the state space and feature space size is unclear, when n becomes very high because there are some states which have low probability of being visited, therefore they don't need the same number of features as the states which are visited often. These states correspond to the memory of picking a few cards.

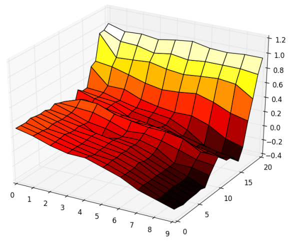
In the case of n=21, the dominant states will be picking from 1 to 4 cards. We can then choose more features for those states, and less features for the other states. The number of possible combinations for the memories from 1 to 4 cards is 7546 * 210 ($\sum_{k=1}^{4} C_2 1^k * 210$) The order of the state space is

10^6.

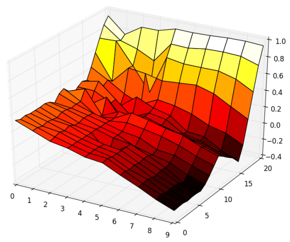
Figures

4 Conclusion

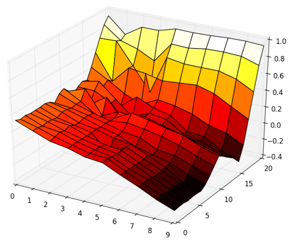
Deep Q Networks and Experience Replay [2]



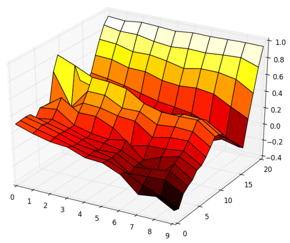
(a) Monte Carlo GLIE



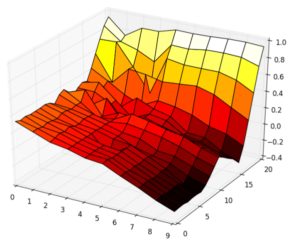
(b) SARSA



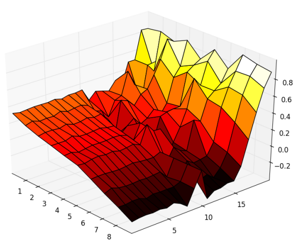
(c) SARSA



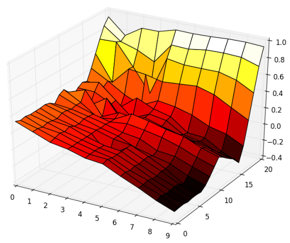
(d) SARSA-λ for $\lambda = 0.8$



(e) SARSA



(f) Linear Function Approximation and SARSA-λ for $\lambda = 0.8$



(g) SARSA

Figure 1: Optimal Value functions after 10^6 episodes for the case of the upper bound of 21, and the corresponding RMSE against Monte Carlo Optimal Value functions with a step-size of 1000

References

[1] Alessandro Lazaric, Marcello Restelli, and Andrea Bonarini. Reinforcement learning in continuous action spaces through sequential monte carlo methods. In *Advances in Neural Information Processing Systems*, 2007.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015.

[3] Danilo Jimenez Rezende, S. M. Ali Eslami, Shakir Mohamed, Peter Battaglia, Max Jaderberg, and Nicolas Heess. Unsupervised learning of 3d structure from images. *CoRR*, abs/1607.00662, 2016.

[4] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[5] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.