
ENSAE ParisTech

Master 1st year Python Machine Learning Project

# On-Policy Model-free 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 last part, I will analyse the results I obtained through simulations.

# 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 [3] [5], shape recognition [4], and it is used for some very complex problems such as autonomous cars [6].

Through this project, 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 [7], 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 $\pi$ 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 $\pi$*

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

**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 then following the policy $\pi$*

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

*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 $\pi$*

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

**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*

$$q_\star(s, a) = \max_\pi q_\pi(s, a)$$

**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 $\epsilon$ greedy policy,then we update our current estimation of the action-value function in the direction of the average of the action value of all visited state-action pairs during the episode.

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

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

## 2.4 SARSA Algorithm

The intuition behind this algorithm is that the agent goes one step away from his state, he gets a reward then he bootstraps the value with his estimate of the value function starting from the state he fell in. Therefore this estimate will have more variance, however, one advantage is that we don't have to go through the whole episode to update the value. His policy is dynamically updated after every action.

**Algorithm 2** SARSA 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**
    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-$\lambda$ 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 $\lambda(1-\lambda)$, then the estimate of n step SARSA has a weight of $\lambda^n(1-\lambda)$. The range of $\lambda$ is [0,1], so if we take a small $\lambda$, we are closer from TD learning with a few steps, and if $\lambda$ 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 $\lambda$ is like a cursor between TD Learning and Monte Carlo Learning.

More formally :

$$q_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} q_t^{(n)}$$

The Algorithm 3 implementation is an backward-view implementation, as opposed to the forward-view I explained above. In this backward view,

we define the eligibility trace E, the agent increment the eligibility trace of the state action pairs with their contribution to his reward. It will be then a weight to the update of the value function for every state action pair.

**Algorithm 3** SARSA-$\lambda$ 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. $\epsilon$greedy)
        $\delta \leftarrow R + \gamma Q(S', A') - Q(S, A)$
        $E(S, A) \leftarrow E(S, A) + 1$
        **for** all $s \in S, a \in \mathbf{A}(s)$ **do**
            $Q(s, a) \Leftarrow Q(s, a) + \alpha\delta E(s, a)$
            $E(s, a) \leftarrow \gamma\lambda E(s, a)$
        $A \leftarrow A', S \leftarrow S',$

## 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 parametrising it. The state action value function becomes $Q(S, A, \mathbf{w})$, where $\mathbf{w}$ is a vector which size is the feature space size. In this case, we update the parameters $\mathbf{w}$ 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, meanwhile for the table lookup case, we have no estimate of the values for unvisited states.

In case of the Monte Carlo 'GLIE' algorithm, the update is :

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

This equation means that we make a step in direction which minimizes 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 which minimizes the difference of our estimate and the sampled value with SARSA-$\lambda$.

# 3 Reinforcement Learning applied to Blackjack

## 3.1 Experimental results

In this section, I will analyze the results of the simulations I performed using 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 = $\{hit, stick\}$

### 3.1.1 Case the score upper limit is 21

The first case is when I set the score upper limit to 21. The state space size is 210. This case has some interesting game related properties. The player will play under the optimal policy on average between two and three games. So the furthest states (high score) are easily reached. As opposed to higher upper bound games. Also the player almost has equal probability of losing or falling in a safe area, when he is between 17 and 21.

I inferred 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

$$V(s, a) = \underset{a \in \mathcal{A}}{max} Q(s, a)$$

In the figure 2, 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 this value is drastically higher, because the dealer has a low probability of reaching these states, therefore the player has high chances of winning.

The figure 2 shows the value function and the RMSE (Root Mean Squared Error) for the algorithms presented in the previous part.

We can see that Monte Carlo is the most Stable algorithm, because we don't see a high variance in the plot of the value function, as opposed to SARSA, whose RMSE is very noisy and never converges. However, the RMSE for SARSA $\lambda$ is stable when the number of episodes grows. And it achieves better performance then the SARSA, which corresponds also to SARSA-$\lambda$ for $\lambda = 0$. It is also what we expected because SARSA-$\lambda$ is a trade-off between Monte Carlo and SARSA.

The linear function approximation achieves lower performance asymptotically than SARSA-$\lambda$, it is normal because it is the same algorithm with less information. It still achieves less biais than SARSA, this depends on the chosen features. However the value function shape is much more different than the other value functions.

In term of convergence speed, SARSA-$\lambda$ is the fastest, followed by the linear approximation,(same base algorithm), then SARSA. It is also natural, because SARSA has much less information than SARSA-$\lambda$ which explores more before bootstrapping. Therefore, it gathers more accurate information per episode, so it gets closer from the optimal solution faster.

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

Quadratic approximation is equivalent to linear approximation in this case because the features are binary.

The Figure 1 shows how the bias decreases asymptotically when $\lambda$ increases. This is due to the more additional weight the agent gives to the further states, this incurs that his estimation gets closer from a Monte Carlo estimation, rather than the TD Learning one.

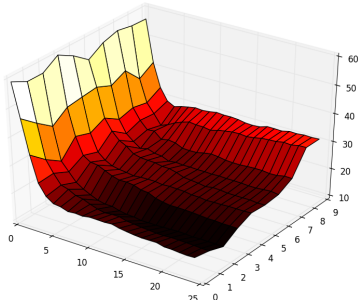
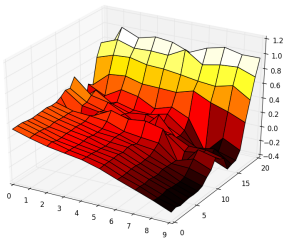
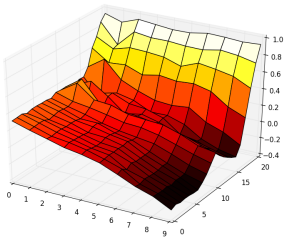


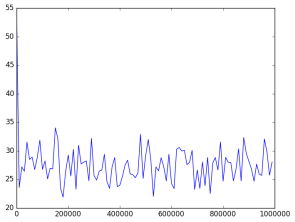

Figure 1: RMSE against episodes for *lambda* in the [0,1] range with a step of 0.1, and for a $10^5$ episodes in total with a measure step of 4000, confronting SARSA-$\lambda$ and Monte Carlo Glie algorithm



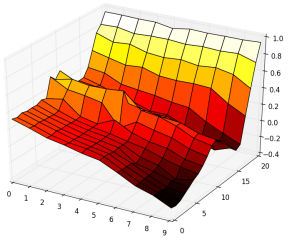
(a) Monte Carlo GLIE



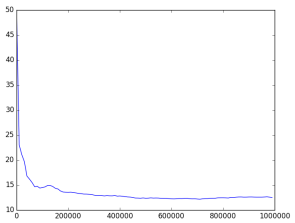
(b) SARSA



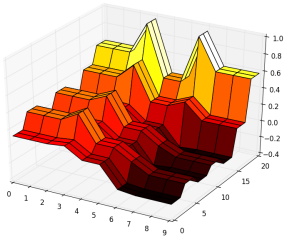
(c) SARSA



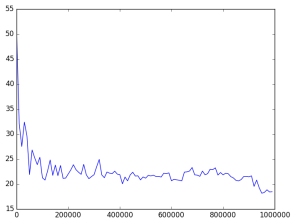
(d) SARSA-$\lambda$ for $\lambda = 0.5$



(e) SARSA-$\lambda$ for $\lambda = 0.5$



(f) Linear Function Approximation and SARSA-$\lambda$ for $\lambda = 0.5$



(g) Linear Function Approximation and SARSA-$\lambda$ for $\lambda = 0.5$

Figure 2: 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

### 3.1.2 Case the upper limit is 51

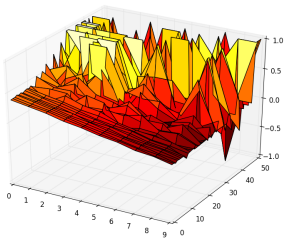
In this case the state space size is 510. The challenge compared to the previous case is reaching some states because there is a low probability of reaching them. Those states are those which the player has a very high score in. Since we sample these states less the variance of the obtained results increases with the player score.

The figure 3 shows the results for an upper bound of 51, for the same number of episodes as the previous case. We notice many abnormal results. The Monte Carlo estimation is very unstable, as opposed to the other estimations. For this reason the RMSE plots, which are computed against the Monte Carlo estimation cannot be interpreted. A possible explanation of this high noise in the Monte Carlo estimation is that, as opposed to TD learning, the agent doesn't change his policy on real time, therefore he loses a lot of time in the states in the middle by oscillating between them, instead of quickly understanding that they are worthless and quickly changing his policy to find more interesting states. We can see that the states close from the beginning have the least variance, which increases the further the state is from the initial ones.

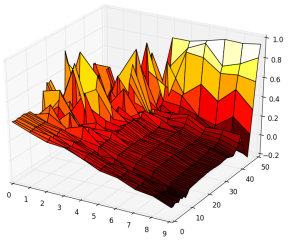
One possible solution is allowing the player to start at t=0 from a state with a higher score than 0 and 10. However, this will have a negative impact on the policy the agent will learn because the policy won't take into account the low probability of reaching the states with very high scores.

Another possible solution is decreasing the $\gamma$ parameter, which was set to 1 so far. This parameters sets the impatience or the value the agent gives to long term rewards. If it is close from 0, it means that the agent wants immediate reward, therefore he will look for the shortest path to this reward. By setting $\gamma$ to 1, we didn't penalize the time the agent takes to get his reward. We see in figure 4 that in fact, there is a slight decrease of the noise when decreasing $\gamma$ to 0.5.

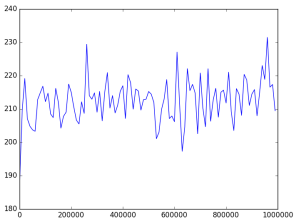
Another simple solution is increasing the number of episodes, however this becomes computationally expensive. In case of multiplying the number of episodes by 10, we obtain the figure, the computation time is also multiplied by 10. Even with $10^7$ episodes, there is still too much variance in the Monte Carlo estimation. This solution is intractable for higher dimension state spaces.



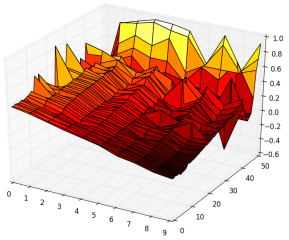
(a) Monte Carlo GLIE



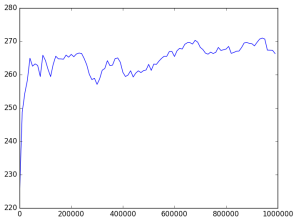
(b) SARSA



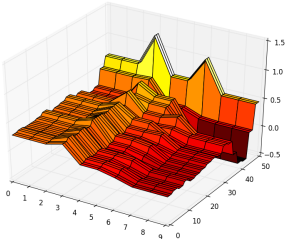
(c) SARSA



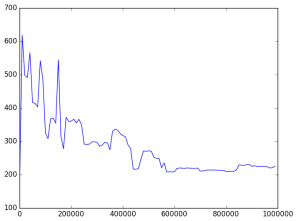
(d) SARSA-$\lambda$ for $\lambda = 0.5$



(e) SARSA-$\lambda$ for $\lambda = 0.5$

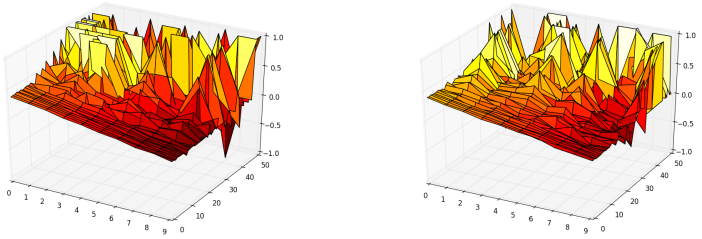


(f) Linear Function Approximation and SARSA-$\lambda$ for $\lambda = 0.5$



(g) Linear Function Approximation and SARSA-$\lambda$ for $\lambda = 0.5$

Figure 3: Optimal Value functions after $10^6$ episodes for the case of the upper bound of 51, and the corresponding RMSE against Monte Carlo Optimal Value functions with a step-size of 1000



(a) $\gamma$=1 (b) $\gamma$=0.5

Figure 4: Optimal Value functions after $10^6$ episodes with Monte Carlo Glie Algorithm for the case of the upper bound of 51

## 3.2 Further discussion

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. The size of the state space is $n \sum_{k=1}^{n} C_n^k$ which is $O(n2^n)$. State Action value function approximation is necessary. It is not possible to use table lookup, therefore the basic Monte Caro GLIE and Temporal difference algorithms are intractable.

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. An explicit relation between the state space and feature space size is unclear when n becomes large; 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.

For example, 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 this sub-state space is $10^6$.

The linear function approximation has theoretical guarantees of convergence, which is not the case of other function approximation such as neural networks, which require other algorithms which were experimentally proved to be stable in some very complex problems, such as Deep Q Networks and Experience Replay [3].

# 4 Conclusion

The algorithms I used achieved reasonable performance in the simple case, however, they needed more tuning in the more general case.

There are several other ways that could increase the quality of the estimation of the optimal policy:

- Changing the exploration policy: I used the epsilon greedy policy in the whole problem, however there are other policies such as optimism in the face of uncertainty, Thomson sampling [1] ...
- Changing the epsilon decay : it has been proven that with the optimal decay, epsilon greedy achieves the theoretical lower regret bound, therefore an appropriate decay could be difficult to outperform with the other algorithms.
- Trying to find the optimal mapping policy directly through policy gradient for example.
- Using a Neural Network action value function approximator with the DQN and Experience Replay algorithm.

# References


[1] Shipra Agrawal and Navin Goyal. Thompson sampling for contextual bandits with linear payoffs. *CoRR*, abs/1209.3352, 2012.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] A. Stafylopatis and K. Blekas. Autonomous vehicle navigation using evolutionary reinforcement learning. *European Journal of Operational Research*, pages 306–318, 1998.

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