Université Catholique de Louvain

Professeur: Marco Saerens
Date: May 20, 2018

Data mining & decision making

LSINF2275

Determining a Blackjack strategy using Q-learning

Auteurs :	*Noma :*
Lucien Ledune	39301400
Axel Struys	19561000

Contents

1 Introduction

In this paper, we will use reinforcement learning to optimize a winning strategy at the blackjack game, and thus maximizing the win rate.

Comparing to the traditional blackjack game, we simplified the rules of the game. First, the player is alone versus the croupier. Second, ace always count as 1 and other figures count as 10. Other cards count as their face values. This enable us to focus more on the learning part than on the creation of the environment itself.

To win at our blackjack game, the player must be as close as possible to 21, without overstepping this value. If the player scores exactly 21, he wins the game. If he overstep, he looses unless the croupier has even more than him. In other cases, he must score greater than the croupier. When the croupier has a score equal or above 17, he stops drawing a card.

The goal of this paper is to determine the optimal strategy, use q-learning and Monte Carlo estimation. We will compare the strategy found with these methods with a semi random strategy.

2 Theory

We will now briefly introduce the theory used for this project. The most important part of our algorithm is the Q-learning function, it enables us to build a policy that (we hope so) will be much better than the semi-random one.

Q-learning is a very popular reinforcement learning technique. The biggest pro to this algorithm is the fact that it is able to learn directly from its environment, we don't need any probability model as for example in the MDP. This is very good for the blackjack game, because even if it is technically possible to calculate those probabilities, it would be a pretty long task, especially as the probabilities to draw a card changes when we draw one. As the Q-learning learns from its environment we will build one, controlled environment where games can be simulated.

Q-learning involves an intelligent agent, a set of states S and a set of actions A that we can perform at each state s_i. We also have a set of rewards R given on certain states under certain conditions. In our case the agent will be the simulation algorithm, transitioning between each states and observing rewards given (or penalities for loosing the game), our states will be the sum of the cards in our hands, and the actions can be to either draw or not. These actions can be performed at every state.

The Q-learning algorithm is based on a function $Q(s, a)$ that gives a certain value to a couple of values (s and a). The goal of the algorithm is to determine those values for each couple of values and then determine the best action at each state, which is essentially a maximisation of $max_a Q(s_i, a)$. The algorithm takes the data gathered by the agent and then use it to iterate the $Q(s, a)$ concerned by that data this way :

$$Q(s_t,a_t) \leftarrow (1-\alpha)\cdot \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\bigg(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}} \bigg)}^{\text{learned value}}$$

There are two major hyperparameters in this function : α and γ. α is the learning rate, which is used in a lot of machine learning algorithms, and γ represents the discount factor. That is because in our Qvalue update function we use the maximum Qvalue of the state we end up in after the chosen action. We in fact take into account the potential future rewards to get the value of a state s_i. This discount factor then reduce the impact of it on the function so it is taken in account for less than the actual reward is, this is pretty intuitive.

At the beginning of the algorithm, we initialize arbitrarily the Qvalues, in our case we used 0 which is pretty neutral.

3 Methodology

In order to get our results, we will go trough multiple steps :

- We will build a semi random strategy, that will be used both to train our model and to check the improvement made by our decision policy. This strategy is to always draw at least two cards, then while the player doesn't obtain 21 or more he has a 50% chance to either draw another card or to break the loop and stop drawing. This enables a lot of different outcomes to the games so our algorithm can learn from a variety of different games.
- We will then take all the necessary data output from these games (S_t, S_{t+1}, a, r) where r is the reward (can be negative in the case of a lost game), and use it to update our Qvalues.
- From the Qvalues we will be able to determine the optimal strategy for each state.

- We will also try to use the monte carlo technique to get better results on the Qvalues. To compare this Monte Carlo with the "standard" method we will use the same numbers of total simulations. This means that for k simulations we will have k/m simulations for each Monte Carlo iteration, m being the number of Monte Carlo iterations.
- We will then simulate a great number of games using different obtained policies and compare the results to see which performs the best, and how much our learned policy is better than our semi random one. In order to make these simulations, we are going to build a controlled environment where the games can be simulated using a certain policy, or the semi random one.

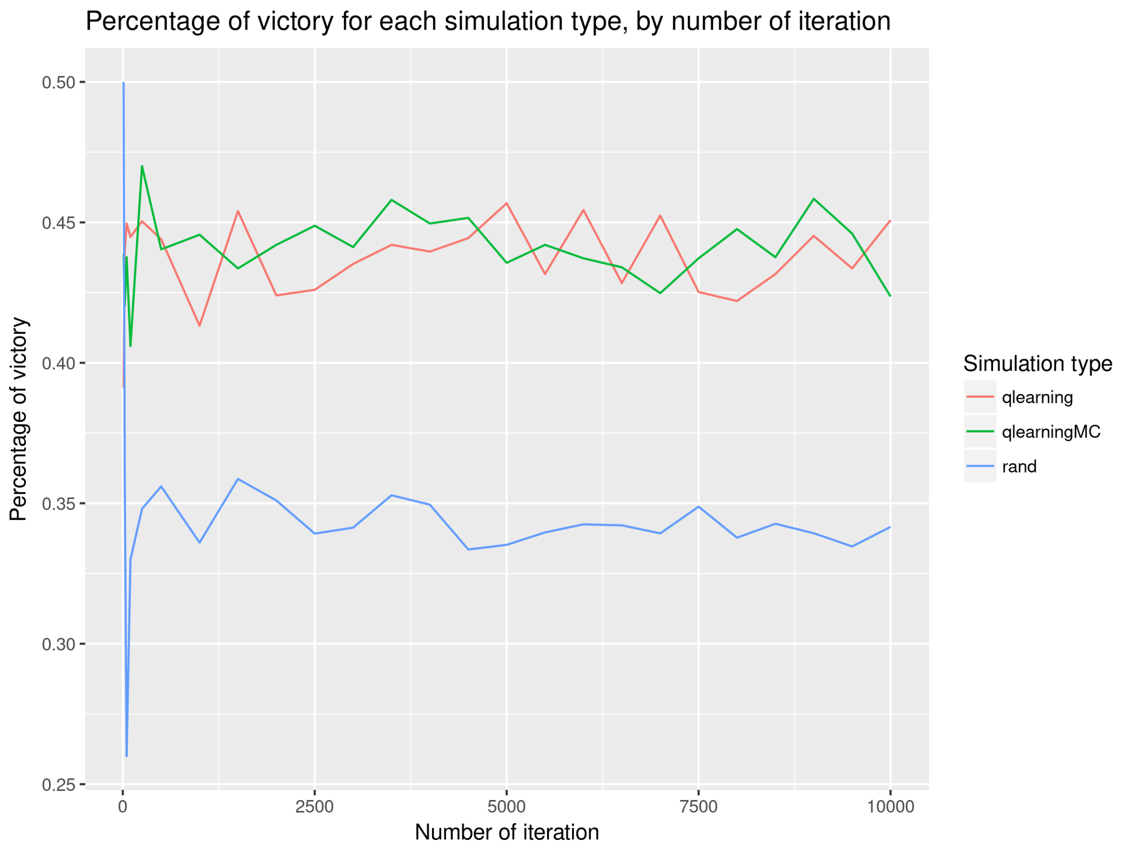
4 Important functions

- $monteCarloSim(m, k, alpha, discount)$: Returns an array of $Q(s, a)$ gathered on the mean of m iterations of k simulations.
- $noMonteCarloSim(k, alpha, discount)$: Same function without monte Carlo.
- $Qpolicy(Qvalues)$: Returns policy associated with an array of Qvalues.
- $policyTestK(k, optP)$: Returns the result of k simulation using a certain policy $optP$.
- $randPolTestK(k)$: Returns the results of k simulations using our semi-random strategy.
- $simulation$: Simulates a game and returns the data necessary for the Q-learning algorithm.
- $simRand(k)$: Simulates k games using $simulation$.

5 Results

Here is a graph showing the convergence of the winrate for our different methods with the number of iterations used to iterate the Qvalues on the X axis. The results shows that our algorithm has a better win rate than the pseudo-random strategy. However, the Monte Carlo technique didn't improve our results compared to using basic Q-values. The results are similar for those two simulation. One possible interpretation is that there is no more room for improvement, showing that the maximum win rate attainable is around 45%. This interpretation is plausible, since a casino game should always favor the casino and not the player. For the random strategy, the win rate was around 35%. Still, the Monte Carlo technique shows less variance and seems more stable.

Simulation type	Variance
Basic Qlearning	0.00022852
Monte Carlo	0.0001735104
Random	0.001364204



6 Discussion

For the choice of our parameters α and γ, we tried several different combinations before sticking to 0.25 and 0.5. This is because the simulations used for the learning algorithm are kind of random (even if it is only semi-random), this way we don't give too many importance to the cases were we would win with an odd strategy purely by luck. The discount was set to 0.5 because our algorithm stocks data for each turn played, this means that the reward only comes after the game is ended, often 2-4 turns. This way we can keep track of the actions performed that enabled us to get that final reward.

7 Conclusion

In this paper, we showed that a Q-Learning algorithm performed better than a pseudo-random strategy. A possible extension of this work is to use the full rules of the black jack game. Also, we could simulate multiple players competing against eachother and the croupier.