




SF2957: Statistical Machine Learning

Project 2

Henrik Söderhielm, 

Thibaud 

Tim 

Edgar Bueno, 

12th December, 2018

1 Introduction

The purpose of this project is to use reinforcement learning to train an agent to play blackjack. Two different representations of the state space have been considered. The first one is based on representing the state as the set of cards and the dealer's card sum. This will be referred to as the "hand" state-space, or S_1 . The other one represents the state as the agent's and dealer's card-sum. It is called S_2 . Both "On- policy Monte-Carlo learning" and "Q-learning" will be considered.

2 Assignments: Numerical Experiments

2.1 Part (a)

First, we had to add a few lines of code in order to complete the MC learning method. The lines we added are the following.

Additional code for the On policy Monte Carlo method

```
#Average reward
avg_reward = avg_reward + (episode_reward - avg_reward)/episode

# Update the Q-function for each visited state/action pair.
for sta in episode_state_action_count :
    for act in episode_state_action_count[sta] :
        if episode_state_action_count[sta][act] != 0 :
            Q[sta][act]=Q[sta][act]+ (episode_reward -
            Q[sta][act])/(state_action_count[sta][act])
```

The first additional line is just the computation of the average reward, which is the indicator of the performance of our strategy. We used the following definition of the average reward :

$$m_n = R_{n-1} + \frac{1}{n}(R_n - m_{n-1}) \text{ where } m_n = \frac{1}{n} \sum_{i=1}^n R_i$$

In the second part of the previous code, we update the action value at the episode j $q_j(s, a)$ thanks to the following formula from the notes :

$$\hat{q}_j(s, a) = \hat{q}_{(j1)}(s, a) + \frac{1}{N_j(s, a)}(G_{s,a}^{(j)} - \hat{q}_{(j1)}(s, a))$$

where $N_j(s, a)$ is the number of the first j episodes that visited (s, a) and $G_{s,a}^{(j)}$ is defined by :

$$G_{s,a}^{(j)} = \sum_{k=1}^{\tau^{(j)} - \tau_{s,a}^{(j)}} \gamma^k R_{\tau_{s,a}^{(j)} + k}^{(j)}.$$

In the Python code, $N_j(s, a)$ is given by `state_action_count[s][a]` and we have $G_{s,a}^{(j)} = R^{(j)}$ with $R^{(j)}$ the final reward of the episode j . Indeed, in our setup, all the rewards are zeros except the last one, so we have only one term in $G_{s,a}^{(j)}$. We also took $\gamma = 1$. Thus, we have $G_{s,a}^{(j)} = R^{(j)}$.

The `learn_MC` function works (mathematically) as follows:

- First, we take an initial action-value dictionary, `Q_init`.
- Then, we run `n_sims` simulations of games. In each episode j , in each state s , the player is following an ϵ -soft policy. It means that he is taking the action a such that $a = \operatorname{argmax}(\hat{q}_{j-1}(s, a))$ with probability $1 - \epsilon + \frac{\epsilon}{|A(s)|}$ and any random action in $A(s)$ with probability $\frac{\epsilon}{|A(s)|}$. Once he is in a terminal state, the game ends and the player receives a reward $R^{(j)}$.
- After each episode, the action value is updated, thanks to the previous equations.

2.2 Part (b)

In order to complete the Q-learning method, we had to add a few lines of code. The lines we added are the following.

Additional code for the Q-learning method

```
c=1
alpha = c * (state_action_count[state][action])**(-omega)
Q[state][action] = Q[state][action] + alpha * (action_reward +
gamma*(np.max(Q[state2])) - Q[state][action])
```

We just update the action value at the time t , $q_t(S_t, A_t)$, thanks to the following formula from the notes :

$$q_t(S_t, A_t) = q_{t-1}(S_{t-1}, A_{t-1}) + \alpha_t (R_t + \gamma \sup_{a \in A(s)} q_{t-1}(S_t, a) - q_{t-1}(S_{t-1}, A_{t-1})).$$

In the project, we took α_t such that $\alpha_n = cn^\omega$. The initial value for ω , defined in `RL-run` is 0.77.

The `learn_Q` function works (mathematically) as follows:

- First, we take an initial action-value dictionary `Q_init`.
- Then, we run `n_sims` simulations of games. In a game, in each state s , the player is following an ϵ -soft policy. It means that he is taking the action a such that $a = \sup_{a \in A(s)} q_{t-1}(S_t, a)$ with probability $1 - \epsilon + \frac{\epsilon}{|A(s)|}$ and any random action in $A(s)$ with probability $\frac{\epsilon}{|A(s)|}$.
- After each step, the action value is updated, thanks to the previous equations. Thus, there are usually several updates in each game, contrary to the MC method. For each non-terminal step, the reward is 0.

2.3 Part (c)

The Monte Carlo and Q-Learning method performances on S1 and S1 & S2 respectively was tested using the empirical expected return and compared. The results found that the number of decks have direct influence on the results. Graph (a) in figure 1 shows the expected return of the three methods when playing with one deck, in terms of the number of episodes when using $\omega = 0.77$. It can be seen that Q-learning was superior to On-policy Monte-Carlo, and in particular, the case using the extended state space yielded the highest, although still negative, expected return. Similar results were obtained for the case of infinitely many decks. The right panel shows the results for the case of two decks. In this case sum state space performed best. Similar results were obtained for six and eight decks.

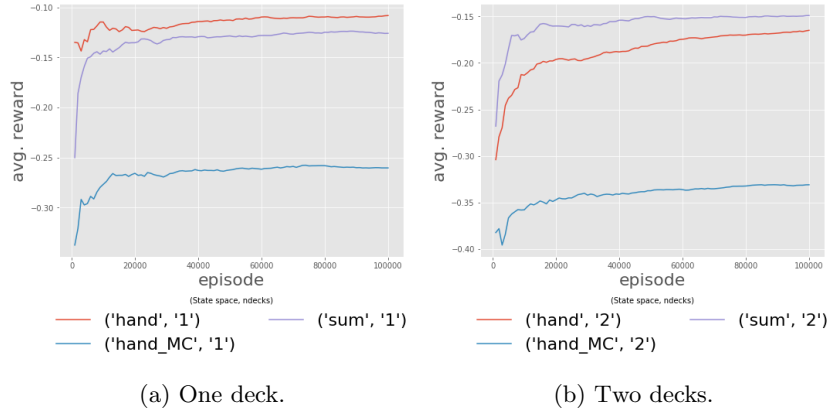


Figure 1: Expected return in terms of the number of episodes for the three methods.

Additionally, we compare the performance of our RL approach for 1,2,4,8 decks w.r.t. the choice of ω , where ω plays the following role in the learning rate α_n ,

$$\alpha_n = cn^{-\omega} \quad (1)$$

In the heatmap in Figure 9 we first see that Q_2 performs significantly better. This might be an effect of quite small choice of epoch size (10^5 epochs), since the state space of Q_1 is much bigger we need more training to visit all the states.

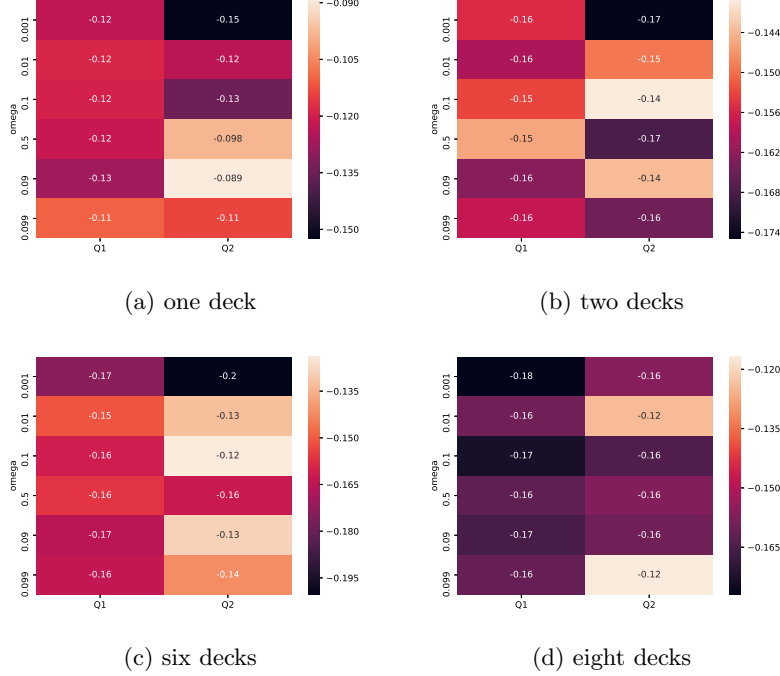


Figure 2: The mean average reward for given ω and Q_1 (full state space), or Q_2 (sum state space).

2.4 Part (d)

In section 2.3, three learning algorithms were compared for five different numbers of decks. In all the cases, Q-learning performed better than On-policy Monte Carlo. However, the decision is not so straightforward between the two state spaces considered under Q-learning. On one hand, the extended state space performed better for one deck and the with-replacement case. On the other hand, the sum state space performed better for the case of two, six and eight decks. In this section we summarize the results by providing tables that indicate which action to take depending on three factors —the presence/absence of a usable ace, the agent card’s sum and the dealer’s visible card— by the number of decks. The python code is found in the Appendix.

For readability we will use the notation Ai for indicating that the card’s sum of the agent is i and Dj for indicating that the dealer’s visible card is a j .

Figure 3 shows the “optimal” strategy to follow in the case of one deck. The left and right panels show, respectively, the case when the agent has zero and one usable ace. The black squares indicate to ask for one more cards (“hit”) and the white squares indicate to “stand”. It is important to note that there are a

couple of squares showing an “unexpected” behavior (D7–A13 and D10–A15 on the left panel and D3–A17, D3–14 and D4–A15 on the right panel). This might be due to the reduced number of iterations. Roughly, the agent’s strategy for the case of no usable ace could be summarized as “ask for more cards if the current sum is smaller than sixteen”. For the case of one usable ace, it could be summarized as “ask for more cards if the current sum is between 13 and 18 and the dealer’s visible card is not an Ace”.

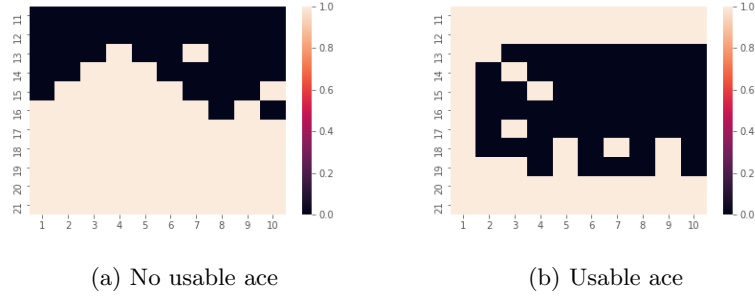


Figure 3: “Optimal” strategy to follow in the case of one deck.

Figure 4 shows the “optimal” strategy to follow in the with-replacement case. The left and right panels show, respectively, the case when the agent has zero and one usable ace. Conventions are as in figure 3. Again, some squares show an unexpected behavior, probably due to a restricted number of iterations. Roughly, the agent’s strategy for the case of no usable ace could be summarized as “ask for more cards if the current sum is smaller than seventeen”. For the case of one usable ace, it could be summarized as “ask for more cards if the current sum is between 12 and 18”.

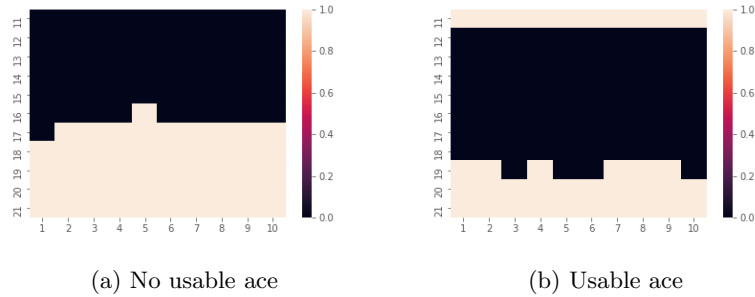


Figure 4: “Optimal” strategy to follow in the with-replacement case.

Analogous plots for the cases of 2, 6 and 8 decks were obtained. They are shown in the Appendix.

In order to know how good our models are, we thought that it would be interesting to compare them with the optimal strategy. Since we are not Blackjack specialists, we searched on the internet for a close to optimal strategy for the state space S_2 . The optimal policy π_* is given in [1], and is represented in Figure 5.

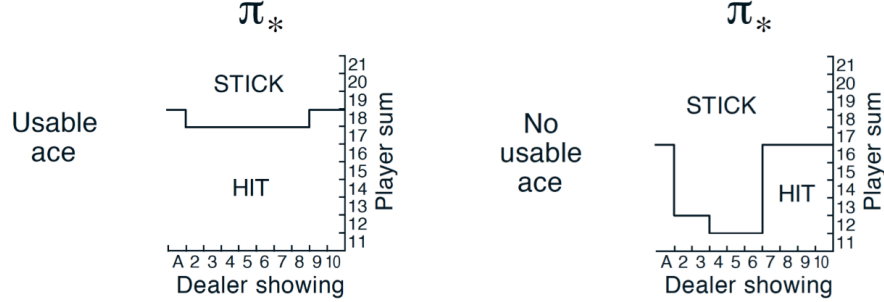


Figure 5: Optimal policy for the state space S_2

Once we have the optimal policy, we can implement a close to optimal Q_{init} based on it. Such an action-value has been implemented in among of the training, at the beginning of `RL.run`

Creation of an optimal Q_{init}

```
def Qinitfun(Sa,Sd,a) :
    if a : #True, usable ace
        if Sa > 18 or ((Sa > 17) and (1<Sd<9)) :
            return(np.array([1,-1]))
        else :
            return(np.array([-1,1]))
    else : #False, no usable ace
        if Sa > 16 :
            return(np.array([1,-1]))
        else :
            if (1<Sd<4 and 12<Sa<17) or (3<Sd<7 and 11<Sa<17) :
                return(np.array([1,-1]))
            else :
                return(np.array([-1,1]))
Q_init = defaultdict(lambda: np.zeros(env.action_space.n))
for a in [False, True] :
    for Sa in range(1,32) :
        for Sd in range(1,11) :
            Q_init[(Sa,Sd,a)] = Qinitfun(Sa,Sd,a)
```

Thanks to this code, we can investigate the influence of the initial strategy. First, we chose to plot the average rewards without updating the action values. To do so, we set $c = 0$ in `RL.py`. The results are shown in figure 6. As expected,

the “optimal” initial Q is doing far better than the default one. With this optimal strategy, for one deck, the average reward is -0.05. This is higher than every average reward that we got in this project.



Figure 6: Average rewards with $c = 0$ (the action values are not updated)

Then, we took back the initial learning rate ($c = 1$ and $\omega = 0.77$) and plot the average reward for one deck. Figure 7 shows that the average reward is decreasing for the optimal initial Q . Indeed, the updates can make q_t going away from q_* , which is based on the optimal strategy.

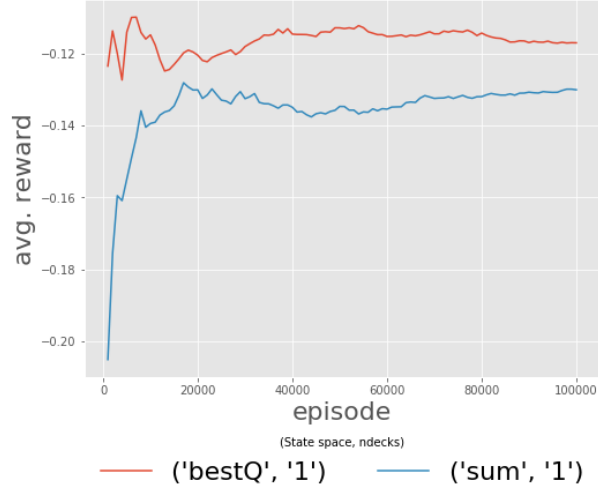


Figure 7: Average rewards with $c = 1$ and $\omega = 0.77$

This problem can be partially dodged by improving the difference between the actions in the initial Q. For example, figure 8 has been obtained by replacing the “[1,-1]” and “[-1,1]” by “[10,-10]” and “[-10,10]”, respectively. Results are then closer to the best ones. To conclude, starting with the best possible Q do not always lead to the best results since the learning can “degrade” the action value q_t .

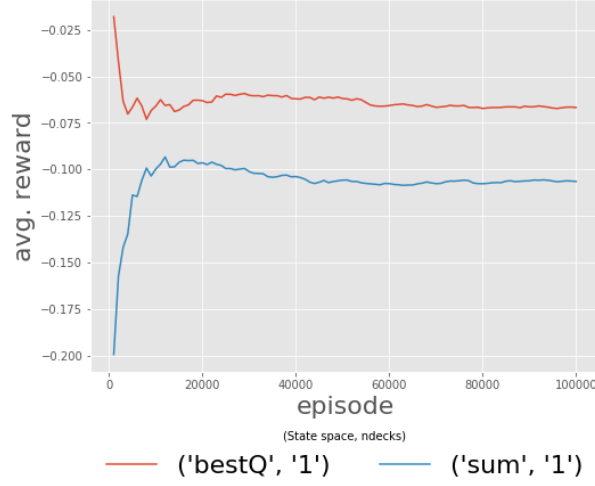


Figure 8: Average rewards with the second version of q_*

3 Conclusion

In this project, we implemented and tested several Reinforcement Learning models. Here are some summarized results that we got:

- The different method are learning quickly: There was nod need to go above one million simulations.
- The On policy Monte Carlo usually gives the worst results for this problem. This might be due to the small amount of non-zero reward in one game.
- The Q learning methods lead to the best results. Those can be better with the hand state space or with the sum based space depending on the number of simulations and the number of decks. Since there are less different states in S_2 , this model learn faster, but S_1 is a more complicated representation of the game which can lead to better overall strategy.
- In the Q learning method, the learning rate can be optimized. This can slightly improve the results.

- A better initial Q can lead faster to higher average rewards. However, a “too good” may sometimes downgrade the results, so this technique should be used with caution.

References

[1] Reinforcement Learning, An introduction, Second edition, Richard S. Sutton and Andrew G. Barto
<https://s3-us-west-1.amazonaws.com/udacity-drlnd/bookdraft2018.pdf>

A Appendix

A.1 “Optimal” strategies for the cases of 2, 6 and 8 decks

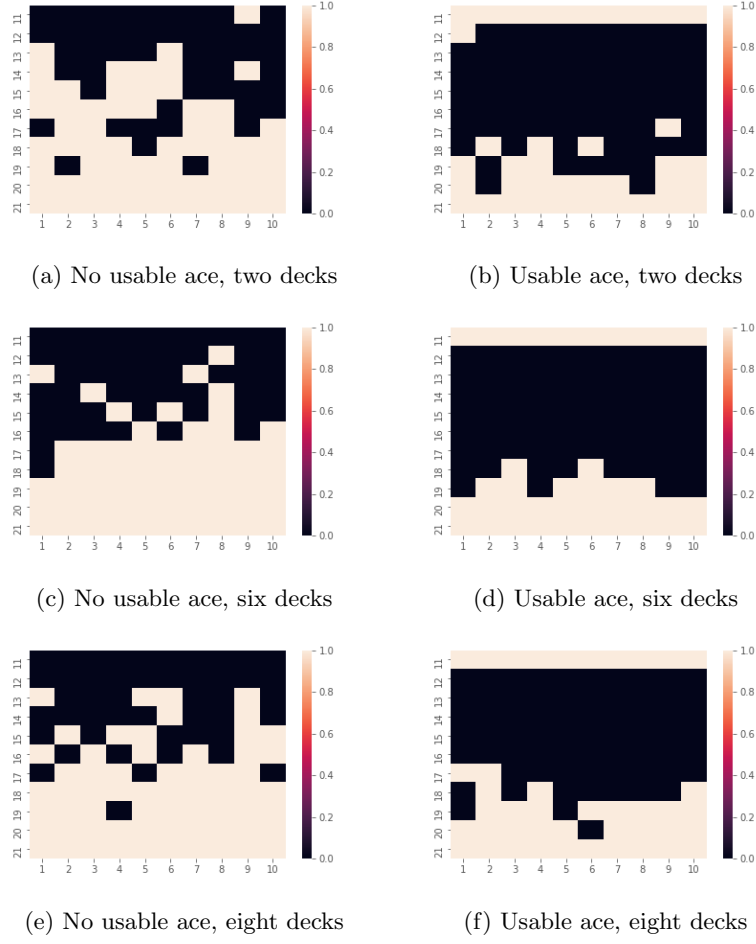


Figure 9: “Optimal” strategy to follow in the case of 2, 6 or 8 decks.