Report II - SF2957 Statistical Machine Learning

Anna Barysheva 960603-7423 Karin Larsson 980522-4384
Ruxue Zeng 981015-T549

16 December, 2021

1 Reinforcement learning for Blackjack

1.1 Project purpose

The purpose of this project is to use reinforcement learning to train an agent to play blackjack. Two different representations of the state space will be considered. Both On-policy Monte-Carlo learning and Q-learning will be considered.

1.2 Problem formulation

In this project the game of Blackjack is setup in the following manner:

- One agent plays against the dealer, with the agent staking one unit on each hand;
- The cards 2-10 counts as their numerical value, suites counts as 10, and ace counts as either 1 or 11 depending on whichever is best;
- If an ace can be counted as 11 without the agent going bust (card sum exceeds 21) it is known as a *usable ace*, the same goes for the dealer;
- The dealer always plays according to the same strategy such that drawing cards until it has a card sum greater than or equal to 17.

State space: The state has the information of the current player and the dealer positions for a given time step. Since in Blackjack, the color and cards suite do not matter, only their numerical values will be considered. Two space states $\mathcal{S}_1$ and $\mathcal{S}_2$ are introduced below.

The numerical values will be used as the cards identifier, with aces equal to 1 . The state space may be represented by the agent's hand $(s_1^a, \dots, s_{10}^a)$ and the dealer's hand $\left(s_1^d, \dots, s_{10}^d\right)$. Here s_i^a and s_i^d is the number of cards of value i for the agent and the dealer respectively. Card sums for both players are restricted as $\sum_{i=1}^{10} is_i^a \leq 31$ and $\sum_{i=1}^{10} is_i^d \leq 26$.

By the rules of the game, the agent's policy takes into account only the dealer's first card and thus the space state can be reduced to

$$\mathcal{S}_1 = \left\{ (s_1, \ldots, s_{10}, S_d) : \sum_{i=1}^{10} i s_i \leq 31, S_i \geq 1 \text{ for } i = 1, \ldots, 10, \text{ and } S_d \leq 26 \right\},$$

where S_d is the dealer's card sum. With this state representation it is always possible to determine if a state is terminal, it is sufficient to check if the agent or dealer is bust, or if dealer's card sum exceeds 17.

An alternative state space is to further reduce the previous state space to

$$\mathcal{S}_2 = \{(S_a, u, S_d) : 1 \leq S_a \leq 31, u \in \{0, 1\}, 1 \leq S_d \leq 26\},$$

where S_a and S_d are the card sums of the agent and the dealer respectively, and u indicates if the player is holding a usable ace (1) or not (0).

Worth to notice, even though the space state $\mathcal{S}_2$ is smaller than $\mathcal{S}_1$, it has some flaws. For instance, the state $(21, 1, S_d)$ may or may not be a Blackjack (depending on how many cards have been drawn). Moreover, in the finite deck setting, this representation does not keep track of the remaining number of cards in the deck.

Action space: The player is allowed to choose to either *stay* or *hit*. Therefore, the actions available for any state are:

$$\mathcal{A} = \{\text{stay}, \text{hit}\}.$$

Rewards: The objective of the agent is to beat the dealer in one of the following ways:

- $\mathcal{R}(\#\text{ the player's cards} = 2; S_a = 21, S_d < 21) = 1.5$
- $\mathcal{R}(\text{Final score check}; S_d < S_a \leq 21) = 1$
- $\mathcal{R}(S_d > 21, S_a \leq 21) = 1$
- $\mathcal{R}(\text{Draw}) = 0$
- For other cases, the agent looses with the reward $\mathcal{R} = -1$

1.3 On-policy evaluation using Monte-Carlo methods

Consider a discounted MDP with a terminal state:

- **MDP:** $(\lambda, S, A_s, p(\cdot \mid s, a), r(s, a), a \in A_s, s \in S)$.
- **Terminal state:** There is a state s_{end} after which no reward is collected.

- **Episode:** It starts at time $t = 1$ in state s_1 and finishes after a random time when s_{end} is reached.

- **Assumption:** under any policy, episodes finish in finite time almost surely.

Let π be a deterministic stationary policy. If one knows the MDP, evaluating the value function $V^\pi(s) = \mathbb{E}\left[\sum_{t=1}^{\infty} \lambda^t r_t\left(s_t^\pi, a_t^\pi\right) \mid s_1^\pi = s\right]$ of π is done by solving: $V^\pi\left(s_{\text{end}}\right) = 0$ and

$$\forall s \neq s_{\text{end}}, V^\pi(s) = r(s, \pi(s)) + \lambda \sum_{j \in S} p(j \mid s, \pi(s)) V^\pi(j)$$

If we do not know the MDP or the transition probabilities are too complex (i.e. when playing Blackjack), model-free RL is used and π can be evaluated through sampling using Monte-Carlo methods (objective: for any state s, evaluate $V^\pi(s)$). Policy π is simulated for n episodes and for i-th episode, it is obtained:

$$\tau_i = \left(s_{1,i}, a_{1,i}, r_{1,i}, \ldots, s_{T_i,i}, a_{T_i,i}, r_{T_i,i}\right),$$

where $r_{t,i}$ is the reward received in step t in episode i and T_i is the length of the episode, i.e., $s_{T_i} = s_{\text{end}}$. If s is visited at time t in episode i, compute the return $G_i = \sum_{u > t} \lambda^{u-t} r_{u,i}$ (First-visit MC). By the law of large numbers,

$$\frac{1}{n} \sum_{i=1}^{n} G_i \rightarrow V^\pi(s), \text{ as } n \rightarrow \infty.$$

The implemented code is structured below. Here, the average reward is updated as the running arithmetic mean following the equation:

$$m_n = m_{n-1} + \frac{1}{n}\left(x_n - m_{n-1}\right).$$

We also update the Q-matrix for each visited pair of state & action.

```
avg_reward += (episode_reward - avg_reward)/episode

for state in episode_state_action_count :
    for action in episode_state_action_count[state] :
        if episode_state_action_count[state][action] > 0 :
            Q[state][action] += (episode_reward - Q[state][action])/
                                (state_action_count[state][action])
```

Listing 1: On-policy Monte-Carlo learning

We define an ϵ-soft policy, which selects a greedy action (the best optimal one) with probability $1 - \epsilon + \epsilon/|\mathcal{A}(s)|$ and any random action in $\mathcal{A}(s)$ with probability $\epsilon/|\mathcal{A}(s)|$. It is initialized with an arbitrary q_0 action value and an ϵ-soft policy π_0. In each iteration i, for each $s \in \mathcal{S}$, let

$$a_*(s) = \text{argmax}_{a \in \mathcal{A}(s)} \hat{Q}_{i-1}(s, a),$$

and, for all $a \in \mathcal{A}(s)$

$$\pi_i(a \mid s) = \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}(s)|, & \text{if } a = a_*(s) \\ \epsilon/|\mathcal{A}(s)|, & \text{if } a \neq a_*(s) \end{cases}$$

For each (s, a) that appears in the i-th episode and that is saved in *episode_state_action_count*, the Q-table is updated as follows:

$$\hat{Q}_\pi^{(i)}(s, a) = \hat{Q}_\pi^{(i-1)}(s, a) + \frac{1}{N_i(s, a)} \left(G_{s,a}^{(i)} - \hat{Q}_\pi^{(i-1)}(s, a) \right)$$

In Blackjack, all rewards are set up to be zeroes until we reach the end of the game and thus $G_{s,a}^{(i)} = r_{T_i,i}$ and the discounted factor is zero.

A potential problem here is that an episodic reward cannot be evaluated until a corresponding episode terminates. This implies that the policy estimate can only be updated after each episode. Thus Q-learning is under consideration that provides estimates' updates after each time step.

1.4 Q-learning

On-policy methods attempt to evaluate or improve the policy that is used to make decisions. An on-policy learner learns the value of the policy being carried out by the agent. The policy used by the agent is computed from the previously collected data. It is an active learning method as the gathered data is controlled.

In opposite, *off-policy* methods evaluate or improve a policy different from that used to generate the data. An off-policy learner learns the value of the optimal policy independently of the agent's actions. The policy used by the agent is often referred to as the behavior policy. In this section, we consider an off-policy algorithm called Q-learning to solve Blackjack problem. More precisely, Q-learning is an off-policy algorithm since it does not need the action a_t when updating $Q^{(t-1)}$.

We apply the R-M algorithm to the estimate the Q-function. Recall that $Q(s, a)$ is the maximum expected reward starting from state s and taking action a :

$$Q(s, a) = r(s, a) + \lambda \sum_j p(j \mid s, a) V^\star(j)$$

Note that $V^\star(s) = \max_{a \in A_s} Q(s, a)$, and hence

$$Q(s,a) = r(s,a) + \lambda \sum_j p(j \mid s,a) \max_b Q(j,b)$$

Q is the fixed point of an operator H (defined on $\mathbb{R}^{S\times A}$)

$$(HQ)(s,a) = r(s,a) + \lambda \sum_j p(j \mid s,a) \max_b Q(j,b)$$

Given the action value $Q^{(t-1)}$, an action a_{t-1} is selected following the ϵ-greedy algorithm (exploration vs. exploitation). Let a_* be the greedy action with respect to $Q(s,a)$, so that:

$$a_*(s) = \mathrm{argmax}_{a\in\mathcal{A}(s)} Q^{(t-1)}(s,a),$$

and, for all $a \in \mathcal{A}(s)$,

$$\pi_{t-1}(a \mid s) = \begin{cases} 1-\epsilon+\epsilon/|\mathcal{A}(s)|, & \text{if } a = a_*(s), \\ \epsilon/|\mathcal{A}(s)|, & \text{if } a \neq a_*(s). \end{cases}$$

We draw an action a_{t-1} from $\pi_{t-1}(\cdot \mid s_{t-1})$ and generate r_t, s_t. Then the action value is updated following the scheme:

$$Q^{(t)}(s_{t-1}, a_{t-1}) = Q^{(t-1)}(s_{t-1}, a_{t-1}) + \alpha \left[r_{t-1} + \lambda \max_{a\in\mathcal{A}(s_t)} Q^{(t-1)}(s_t, a) - Q^{(t-1)}(s_{t-1}, a_{t-1}) \right]$$

***Q*-learning convergence.** Assume that the step sizes (α_t) satisfy $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 < \infty$. Next, assume that the behaviour policy π_b visits every (state, action) pairs infinitely often. For any discount factor $\lambda \in (0,1)$, under the Q-learning algorithm,

$$\lim_{n\to\infty} Q^{(t)} = Q, \qquad \text{almost surely.}$$

In the implementation, we computed the learning rate α and updated the Q-table as presented below:

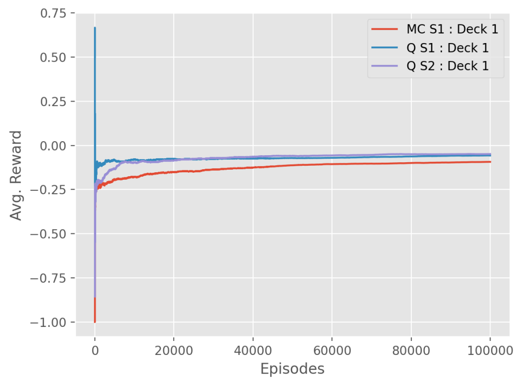
```
alpha = (state_action_count[state][action])**(-omega)
Q[state][action] += alpha*(action_reward +
                    gamma*(np.max(Q[state2]))-Q[state][action])
```

Listing 2: Q-learning learning

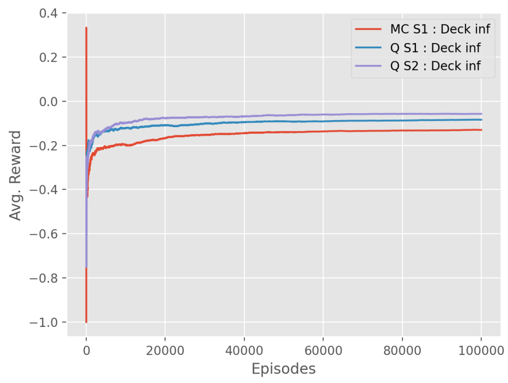
Each time step, we update the learning rate as it is recommended in lecture notes: $\alpha_t = ct^\omega$, where $\omega = 0.77$ by default.

1.5 Performance evaluation

The performance was evaluated of two learning algorithms : Monte-Carlo on S_1 and Q-Learning on both $\mathcal{S}_1$ and $\mathcal{S}_2$. From Figure 1, when $\omega = 0.77$ is fixed and the average reward over episodes is plotted, Q-Learning shows a slightly better results in comparison to on-policy Monte-Carlo.



(a) One deck



(b) Inf decks

Figure 1: Performance of three different learning algorithms

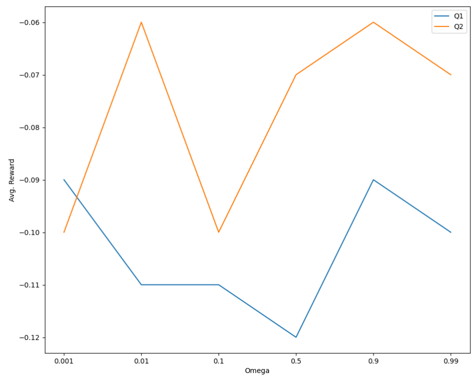
It also can be seen that Q-learning performance on the different state spaces $\mathcal{S}_1$ and $\mathcal{S}_2$ depends on the number of decks. When there is one deck to play, both result similarly; however, when the number of decks increases, the reduced state space $\mathcal{S}_2$ performed better than the extended state space $\mathcal{S}_1$. This could happened due to the fact that when only one deck is played, Q-learning on both state spaces learns quickly and in addition, the average rewards seem to coverage. However, when the infinite number of decks is played, Q-learning trained on $\mathcal{S}_1$ might need much more time to learn compared to $\mathcal{S}_2$. Since it is impossible to keep track of the remaining number of cards for the infinite deck, the sum state space $\mathcal{S}_2$ is more preferable for this case. For the finite large number of decks, before obtaining the convergence, the sum state space can potentially results better than the extended one.

1.5.1 Decay rate ω

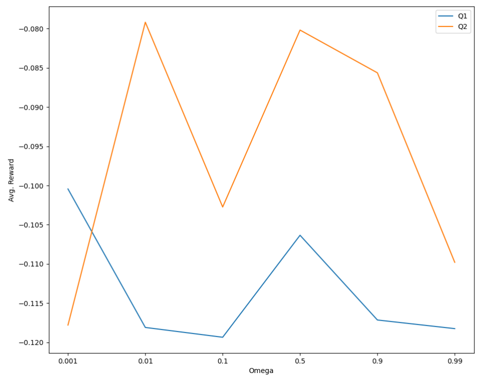
Additionally, the performance of Q-Learning trained on both state spaces for $1, 2, 6, 8$ and inf decks w.r.t. the choice of ω was considered. The number of episodes is fixed to be 10^4 for all following simulations. Recall that ω plays the following role in the learning rate α_n:

$$\alpha_n = cn^{-\omega}$$

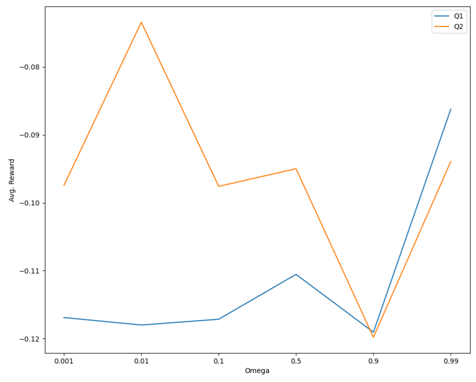
Starting from here, if Q-learning is trained on the extended space state $\mathcal{S}_1$, the algorithm is called Q_1. If the sum space $\mathcal{S}_2$ is used, Q_2 then is trained.



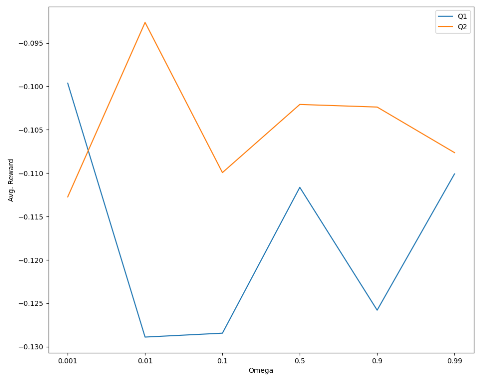
(a) One deck



(b) Two decks



(c) Eight decks



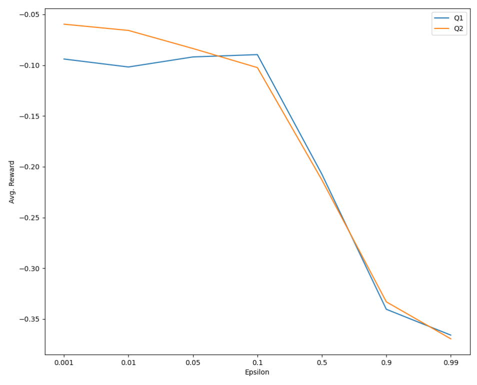
(d) Inf decks

Figure 2: The average reward of Q_1 and Q_2 for given ω

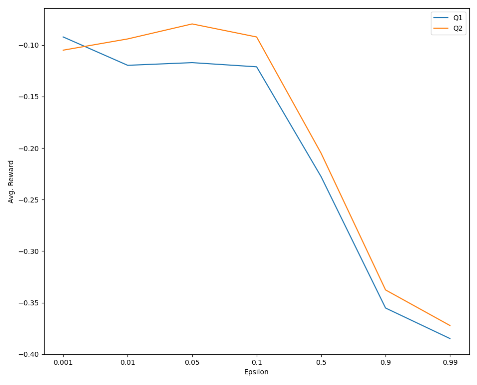
In general, we observe in Figure 2 that Q_2 performs significantly better than Q_1. This might be an effect of quite small choice of epoch size. More specific, the state space $\mathcal{S}_1$ is much bigger than $\mathcal{S}_2$, so it needs much more iterations to visit all the states. We remark also that the value of omega doesn't affect a lot the performance of both algorithms, maybe with number of episodes sufficient, it doesn't matter that much.

1.5.2 Decaying exploration rate ϵ

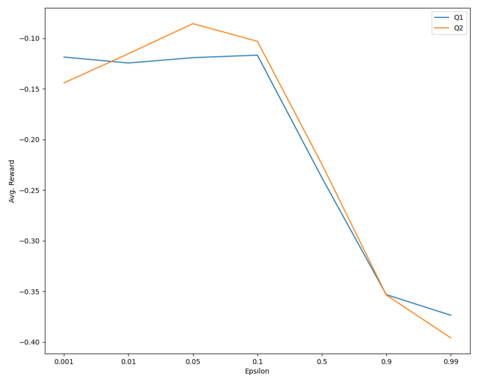
We also want to compare the performance of Q_1 and Q_2 w.r.t. the choice of decaying exploration rate ϵ. In Figure 3, we observe that with same $\omega = 0.77$ and number of episodes $= 10^4$, Q_1 and Q_2 had same order of performance, like what we obtained before. We remark also that the performance decreases when ϵ increasing, this can be explained by the fact the exploration rate is the probability of not choosing the optimal policy and taking a random action. When choosing large ϵ, taking random actions often when the environment is already explored enough does not improve the performance.



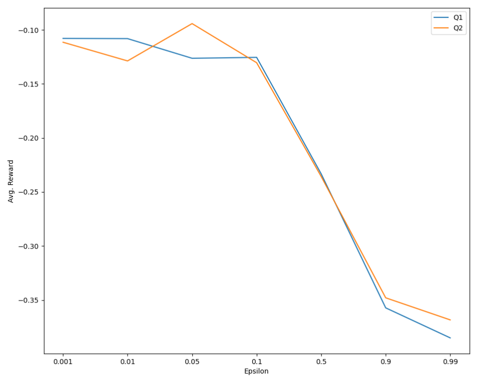
(a) One deck



(b) Two decks



(c) Six decks



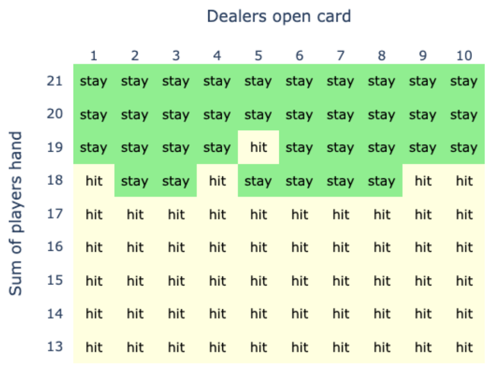
(d) Eight decks

Figure 3: The average reward for given ϵ of Q_1 and Q_2

1.6 Optimal Policy

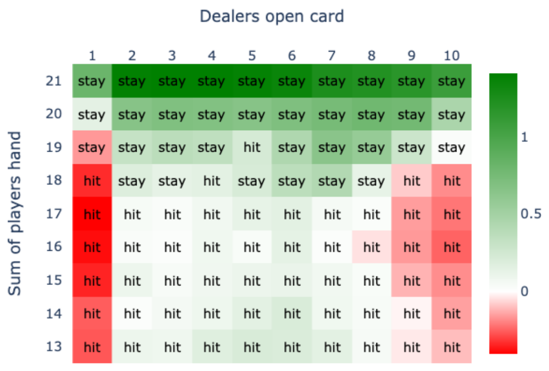
The optimal policy when using the Q-learning algorithm was obtained. The Q-learning algorithm was run for 1 000 000 episodes using six decks. The optimal policies are highlighted in Figures 4a and 5a. In Figures 4b and 5b the corresponding Q-values can be observed.

Note that the minimal sum that the player can have is 4 (2+2). The ace is counted as 11 if possible. Therefore, in order to view all possible situation where an action can be taken it is only necessary to view the sum of the players cards larger than 3 and smaller than 22. Note that the ace is represented as 1 among the dealers cards. In the case when the player has a usable ace, the sum of the players hand can not be less than 13 (ace + 2).



	1	2	3	4	5	6	7	8	9	10
21	stay	stay	stay	stay	stay	stay	stay	stay	stay	stay
20	stay	stay	stay	stay	stay	stay	stay	stay	stay	stay
19	stay	stay	stay	stay	hit	stay	stay	stay	stay	stay
18	hit	stay	stay	hit	stay	stay	stay	stay	hit	hit
17	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
16	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
15	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
14	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
13	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit

(a) Optimal policy

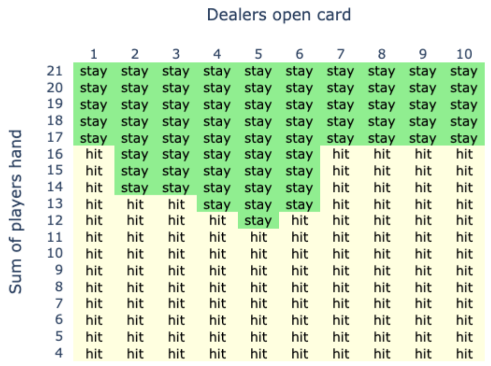


	1	2	3	4	5	6	7	8	9	10
21	stay	stay	stay	stay	stay	stay	stay	stay	stay	stay
20	stay	stay	stay	stay	stay	stay	stay	stay	stay	stay
19	stay	stay	stay	stay	hit	stay	stay	stay	stay	stay
18	hit	stay	stay	hit	stay	stay	stay	stay	hit	hit
17	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
16	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
15	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
14	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
13	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit

(b) Q-values for the optimal policy

Figure 4: Optimal policies when player has a usable ace. Values obtained from Q-learning

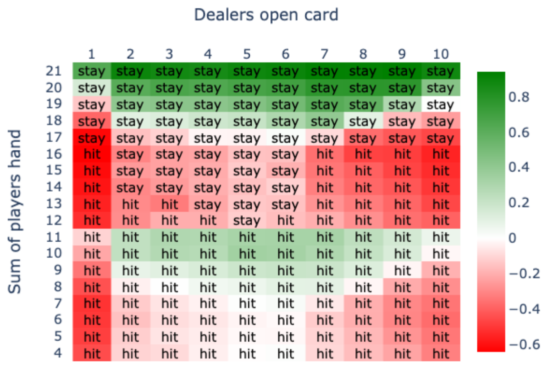
On the Internet, one can find suggested optimal policies, i.e. an optimal policy for 6 decks in Figure 6. There *Hard* and *Soft* correspond to case of no usable and usable aces respectively. Thus, comparing recommended strategies from the Internet and the optimal policies calculated in this project, can be concluded that the obtain results are reasonable.



Dealers open card

Sum of players hand	1	2	3	4	5	6	7	8	9	10
21	stay	stay	stay	stay	stay	stay	stay	stay	stay	stay
20	stay	stay	stay	stay	stay	stay	stay	stay	stay	stay
19	stay	stay	stay	stay	stay	stay	stay	stay	stay	stay
18	stay	stay	stay	stay	stay	stay	stay	stay	stay	stay
17	stay	stay	stay	stay	stay	stay	stay	stay	stay	stay
16	hit	stay	stay	stay	stay	stay	hit	hit	hit	hit
15	hit	stay	stay	stay	stay	stay	hit	hit	hit	hit
14	hit	stay	stay	stay	stay	stay	hit	hit	hit	hit
13	hit	hit	hit	stay	stay	stay	hit	hit	hit	hit
12	hit	hit	hit	hit	stay	hit	hit	hit	hit	hit
11	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
10	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
9	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
8	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
7	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
6	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
5	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
4	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit

(a) Optimal policy

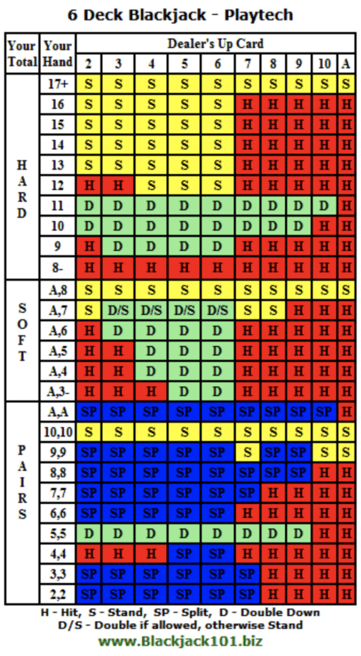


Dealers open card

Sum of players hand	1	2	3	4	5	6	7	8	9	10
21	stay	stay	stay	stay	stay	stay	stay	stay	stay	stay
20	stay	stay	stay	stay	stay	stay	stay	stay	stay	stay
19	stay	stay	stay	stay	stay	stay	stay	stay	stay	stay
18	stay	stay	stay	stay	stay	stay	stay	stay	stay	stay
17	stay	stay	stay	stay	stay	stay	stay	stay	stay	stay
16	hit	stay	stay	stay	stay	stay	hit	hit	hit	hit
15	hit	stay	stay	stay	stay	stay	hit	hit	hit	hit
14	hit	stay	stay	stay	stay	stay	hit	hit	hit	hit
13	hit	hit	hit	stay	stay	stay	hit	hit	hit	hit
12	hit	hit	hit	hit	stay	hit	hit	hit	hit	hit
11	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
10	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
9	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
8	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
7	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
6	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
5	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
4	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit

(b) Q-values for the optimal policy

Figure 5: Optimal policies when player does not have a usable ace. Values obtained from Q-learning



6 Deck Blackjack - Playtech

Your Total	Your Hand	Dealer's Up Card									
		2	3	4	5	6	7	8	9	10	A
HARD	17+	S	S	S	S	S	S	S	S	S	S
	16	S	S	S	S	S	H	H	H	H	H
	15	S	S	S	S	S	H	H	H	H	H
	14	S	S	S	S	S	H	H	H	H	H
	13	S	S	S	S	S	H	H	H	H	H
	12	H	H	S	S	S	H	H	H	H	H
	11	D	D	D	D	D	D	D	D	D	H
	10	D	D	D	D	D	D	D	D	H	H
	9	H	D	D	D	D	H	H	H	H	H
	8-	H	H	H	H	H	H	H	H	H	H
SOFT	A,8	S	S	S	S	S	S	S	S	S	S
	A,7	S	D/S	D/S	D/S	D/S	S	S	H	H	H
	A,6	H	D	D	D	D	H	H	H	H	H
	A,5	H	H	D	D	D	H	H	H	H	H
	A,4	H	H	D	D	D	H	H	H	H	H
	A,3-	H	H	H	D	D	H	H	H	H	H
PAIRS	A,A	SP	SP	SP	SP	SP	SP	SP	SP	SP	H
	10,10	S	S	S	S	S	S	S	S	S	S
	9,9	SP	SP	SP	SP	SP	S	SP	SP	S	S
	8,8	SP	SP	SP	SP	SP	SP	SP	SP	H	H
	7,7	SP	SP	SP	SP	SP	SP	H	H	H	H
	6,6	SP	SP	SP	SP	SP	H	H	H	H	H
	5,5	D	D	D	D	D	D	D	D	H	H
	4,4	H	H	H	SP	SP	H	H	H	H	H
	3,3	SP	SP	SP	SP	SP	SP	H	H	H	H
	2,2	SP	SP	SP	SP	SP	SP	H	H	H	H

H - Hit, S - Stand, SP - Split, D - Double Down
D/S - Double if allowed, otherwise Stand
www.Blackjack101.biz

Figure 6: Optimal policy for 6 decks taken from the Internet

1.7 Summary

Choosing between Q-learning and on-policy Monte-Carlo is bias-variance trade-off one should decide on. Q-learning updates are biased since they depend on the parameter initialization. This was illustrated when the average reward for different values of ω were calculated. Although on-policy Monte-Carlo does not have this problem, it suffers from high variance and thus, to achieve the same performance as for Q-learning, more episodes are needed.

Based on the obtained results, Q-Learning had a better performance than on-policy Monte-Carlo learning algorithm when playing Blackjack. In particular, the sum state space performs better than extended state space when we have 10^5 episodes.

Also, regarding the exploration rate, it is reasonable to have a decaying exploration rate with the number of iterations rather than a fixed one. This means that the environment is explored less and less over time and the algorithm is more focused on the optimal solution.

2 Appendix

2.1 Optimal policy

```python
import blackjack_extended as bjk #The extended Black-Jack environment
import blackjack_base as bjk_base #The standard sum-based Black-Jack environment
import RL as rl

import sys
import os
import time
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import plotly.express as px
import plotly.graph_objects as go
import plotly.figure_factory as ff

def train_model():
    """Train the model, return Q. Use fixed values """
    directory = "{}/data".format(sys.path[0])
    if not os.path.exists(directory):
        os.makedirs(directory)
    path_fun = lambda x: "{}/{}_{}.txt".format(directory,x, decks)

    sum_env = bjk_base.BlackjackEnvBase(decks = 8, seed = 31233)

    n_sims = int(1e6)
    # Q-learning with player sum state representation
    sumQ, sum_avg_reward, sum_state_action_count, sum_avg_rewards = rl.learn_Q(
        sum_env, n_sims, omega = 0.77, epsilon = 0.05, init_val = 0.0,
        episode_file=path_fun("sum_state"), warmup = n_sims//10)

    print("Model trained")
    print("Number of explored states (sum states): " + str(len(sumQ)))
    print("Cumulative avg. reward = " + str(sum_avg_reward))

    return sumQ

def calculate_best_actions(sumQ, savetype = 'numpy', save = True):
    """ Calculate the best actions given each state.
    savetype denotes how the values should be saved. Options: all in one file('df'), separate files('numpy') or 'both'
    if save == False, the values will not be saved, simply printed"""
    #state has form (12, 1, False): (env.sum_hand(self.player), env.dealer_show_cards(), env.usable_ace(self.player))
    #print("State ", state, type(state))
    if save:#check that the correct folder exists, otherwise create folder
        directory = "{}/saved_optimal".format(sys.path[0])
        if not os.path.exists(directory):
            os.makedirs(directory)

    sum_player_pos = range(22) #posible player sums
    sum_dealer_pos = range(11) #posible dealer sums
    ace_pos = [True, False]

    if (savetype == 'numpy' or savetype == 'both'):
        actions = np.full([len(sum_player_pos),len(sum_dealer_pos)], np.nan
)
        Qmax = np.full([len(sum_player_pos),len(sum_dealer_pos)], np.nan)
        actions_string = np.full([len(sum_player_pos),len(sum_dealer_pos)],
 ' ')

        for ace_TF in ace_pos:
            for sum_dealer in sum_dealer_pos:
                for sum_player in sum_player_pos:
                    state = (sum_player,sum_dealer, ace_TF)
                    act = np.argmax(sumQ[state]) # Take the best possible
action
                    Qmax[sum_player,sum_dealer] = sumQ[state][act]
                    actions[sum_player,sum_dealer] = act
            if save:
                if ace_TF:
                    string = "T"
                else:
                    string = "F"
                csvfile = "saved_optimal/actions_ace"+ string + ".csv"
                np.savetxt(csvfile, actions, delimiter = ',')

                csvfile_Qmax = "saved_optimal/Qmax_ace"+ string + ".csv"
                np.savetxt(csvfile_Qmax, Qmax, delimiter = ',')
            else:
                print(actions)

    if (savetype == 'df' or savetype == 'both'):
        max_index = len(sum_dealer_pos)*len(sum_player_pos)*len(ace_pos)
        deal_list = ['']*max_index
        play_list = ['']*max_index
        act_list = ['']*max_index
        act_TF_list = [0]*max_index
        ace_list = ['']*max_index

        i = 0
        for ace_TF in ace_pos:
            for sum_dealer in sum_dealer_pos:
                for sum_player in sum_player_pos:
                    state = (sum_player,sum_dealer, ace_TF)
```

```python
                    act = np.argmax(sumQ[state]) # Take the best possible action
                    #values for DataFrame
                    if act:
                        #actions_string[sum_player,sum_dealer] = 'hit'
                        act_list[i] = 'hit'
                    else:
                        #actions_string[sum_player,sum_dealer] = 'stay'
                        act_list[i] = 'stay'
                    deal_list[i] = sum_dealer
                    play_list[i] = sum_player
                    #act_list[i] = actions_string[sum_player,sum_dealer]
                    ace_list[i] = ace_TF
                    act_TF_list[i] = act
                    i+=1


        df = pd.DataFrame({'deal_sum': deal_list,'play_sum': play_list, 'action':act_list, 'action_TF': act_TF_list, 'usable_ace(TF)':ace_list })
        if save:
            df.to_csv('saved_optimal/actionDF.csv')
        else:
            print(df)

    return

def show_action_df():
    df = pd.read_csv('actionDF.csv', index_col = 0)
    #sub_df = df[(df['usable_ace(TF)'] == ace_TF)]
    #fig = px.imshow(df)
    #print(sub_df)

    fig = px.density_heatmap(df, x='play_sum', y='deal_sum', z = 'action_TF',nbinsx=35, nbinsy=28, facet_col="usable_ace(TF)", histfunc="sum")
    fig.show()
    return

def load_action_Qmax(ace_TF):
    """Helpfunction to load actions/Qmax from csvfiles. Used by show_action_plotly, show_action_matplot"""
    if ace_TF:
        string = "T"
    else:
        string = "F"

    directory = "{}/".format(sys.path[0])

    csvfile = "saved_optimal/actions_ace"+ string + ".csv"
    if not os.path.isfile(directory+csvfile):
        raise ValueError("Can not load {} \n file does not exist".format(
    csvfile))
    actions= np.genfromtxt(csvfile, delimiter=',', dtype=None)

    Qfile = "saved_optimal/Qmax_ace"+ string + ".csv"
    if not os.path.isfile(directory+csvfile):
        raise ValueError("Can not load {} \n file does not exist".format(
    Qfile))
    Qmax = np.genfromtxt(Qfile, delimiter=',', dtype=None)

    return actions, Qmax

def show_action_matplot(ace_TF):
    """Show optimal action using matplotlib"""
    actions, Qmax = load_action_Qmax(ace_TF)
    plt.matshow(actions)

    #plt.axis('equal')
    plt.axis([0.5,10.5,3.5,21.5])
    #plt.grid(visible = True)
    plt.xlabel("Dealers open card")
    plt.ylabel("Sum of players hand")

    plt.show()
    return

def show_action_plotly(ace_TF, save = False, Q_color = False):
    """Visualize the best policies(from calculated files) using plotly.
    ace_TF (True/False) denotes uasble ace,
    save == True: saves heatmap as png. save == False: shows plot in
    interactive window
    Q_color == False: color is optimal action. Q_color == True: color is
    optimal Q-value"""
    actions, Qmax = load_action_Qmax(ace_TF)

    #get the strings for annotation
    actions_string = np.full([len(actions),len(actions[0])], ' ', dtype = '
    U5')
    for i in range((len(actions))):
        for j in range(len(actions[0])):
            if actions[i,j]:
                actions_string[i, j] = 'hit'
                #print("hit when player %d, dealer %d" %(i,j))

            elif not actions[i,j]:
                actions_string[i, j] = 'stay'

    #create annotated heatmap
    #set x/ylabels
    x = list(range(len(actions[0])))
```

```python
    y = list(range(len(actions)))

    if Q_color: #Want the color to represent Q-value
        zero = -np.min(Qmax)/(np.max(Qmax)-np.min(Qmax)) #find zero for colorscale
        #fig = ff.create_annotated_heatmap(Qmax, x = x, y =y, annotation_text=actions_string, showscale = True,  colorscale='ReYlGn')
        fig = ff.create_annotated_heatmap(
            Qmax,
            x = x,
            y =y,
            annotation_text=actions_string,
            showscale = True,
            colorscale=[[0,'red'],[zero, 'white'],[1,'green']],
            )
        fig.update_layout(legend = dict(title= "Q-value"), showlegend = True)
    else:
        fig = ff.create_annotated_heatmap(actions, x = x, y =y, annotation_text=actions_string, colorscale=[[0, 'lightgreen'], [1, 'lightyellow']])

    fig.update_layout(
        #title = "usable ace = "+ str(ace_TF),
        font = dict(size = 14),
        width = 600,
        yaxis_title = "Sum of players hand",
        xaxis_title = "Dealers open card"
        )
    fig.update_xaxes(
        gridwidth = 1,
        range = [0.5,10.5]
    )
    if ace_TF:
        fig.update_yaxes(
            gridwidth = 1,
            range = [12.5,21.5]
        )
    else:
        fig.update_yaxes(
            gridwidth = 1,
            range = [3.5,21.5]
        )
    if save:
        if ace_TF:
            string = "T"
        else:
            string = "F"
        if Q_color:
            figname = "saved_optimal/open_actionsQ_ace"+ string +".png"
        else:
            figname = "saved_optimal/open_actions_ace"+ string +".png"
        fig.write_image(figname)
    else:
        fig.show()
    return


if __name__ == "__main__":
    #train model
    sumQ = train_model() #add variables if wanted
    #get best actions
    calculate_best_actions(sumQ, savetype = 'numpy', save = True)

    #visualize actions
    for ace_TF  in [True, False]:
        show_action_plotly(ace_TF, save = True, Q_color=True)
        #show_action_matplot(ace_TF)
    #show_action_df()
```

Listing 3: Optimal policy