Reinforcement Learning and Blackjack

Benjamin Allévius[1], Sebastian Rosengren[1], and Erik Thorsén[1]

[1]Department of Mathematics, Stockholm University, Sweden

Abstract

1 Introduction

The purpose of this project is to use reinforcement learning to train an *agent* to play blackjack and investigate the learning capabilities of the framework. We implement two different representations of the state space, one of which, albeit intuitive, does not satisfy the model's stationarity assumptions. In comparing the two different representations we investigate the importance of the stationarity assumption. We apply these state representations and reinforcement learning to different versions of blackjack—namely difference in how many decks of cards that are used, from a finite number to infinite.

2 Reinforcement Learning

Reinforcement learning is, as many things are in machine learning, both the problems and solutions pertaining to a specific domain. The problem is how an *agent* ought to act in an *environment* with imperfect information and randomness, but with feedback. The solutions are many, but most common is the Markov decision process approach.

2.1 Markov Decision Process

Markov decision processes (MDP) is the formalism which allows us to reason about reinforcement learning in a mathematical way, and all standard reinforcement learning problems are formulated in a Markov decision process framework. We start by giving the definition to the simplest case of a *stationary* Markov decision process.

The building blocks of a Markov decision process consist of

(i) S a finite space of *states*

(ii) A a finite space of *actions*

(iii) R a finite space of *rewards*

(iv) $P_a(s, s')$ a transition probability function defined for all $(s, a, s') \in S \times A \times S$

(v) $r(s, a)$ the immediate or expected immediate reward of taking action a in state s.

(vi) $\gamma \in [0, 1]$, the discount factor (here always equal to 1).

Let $\{(S_t, A_t, R_{t+1}),\ t \geq 0\}$ be a stochastic process, where $S_t \in S$ is the state of the system at time t; $A_t \in A$ is the (potentially random) action taken at time t, and $R_t \in R$ is the immediate reward at time t. The *history* H_t of the system up to time t is the random vector given by $(S_0, A_0, S_1, A_1, \ldots, S_{t-1}, A_{t-1}, S_t)$. An important concept going further is that of a policy: a policy $\pi = (\pi_0, \pi_1, \ldots)$ is a sequence of decision rules, where π_n is a function of the history of the process up to time n (i.e. H_n) mapping to a probability measure over A.

Now, we call the process $\{(S_t, A_t, R_{t+1}),\ t \geq 0\}$ a Markov decision process if

$$\begin{aligned}\mathbb{P}(S_t = s | S_{t-1} = s_{t-1}, A_{t-1} = a_{t-1}, \ldots, S_0 = s_0, A_0 = a_0) \\ = \mathbb{P}(S_t = s | S_{t-1} = s_{t-1}, A_{t-1} = a_{t-1}) = P_a(s_{t-1}, s) \quad (1)\end{aligned}$$

and R_{t+1} given s_t, a_t is independent of H_t, with $\mathbb{E}(R_{t+1}|H_t) = r(S_t, A_t)$. The process is called stationary since the transition probabilities nor the reward kernel depends on t.

At first glance expression (1) makes sense, but there are subtle details which need to be addressed. For instance, we have said nothing about the process $\{A_t\}$ and therefore we should be careful defining conditional probabilities containing it. However, if $\{A_t\}$ is governed by a policy $\pi = (\pi_0, \pi_1, \ldots)$ then we can define the above probability. In order for $\{(S_t, A_t, R_{t+1}),\ t \geq 0\}$ to be a Markov decision process we formally require that (1) holds for all policies. For a thorough treatment of Markov decision processes see **?**.

The goal of reinforcement learning is to take a MDP and choose a policy which maximizes expected (discounted) rewards. This can seem hopeless since policy decisions in general depends on the whole history of the chain. However, it can be shown that for finite stationary MDPs this is equivalent to maximizing rewards over policies which take into account only the current state of the process and which maps this to a single action, i.e. $\pi_n(H_n) = \pi(S_t) \in A$ — this is know as a *stationary Markov policy*. This is something that make the optimizing problem considerably easier, and allows e.g. the *Q-learning* algorithm to give estimates that converges to the optimal solution. Hence, from here on out we consider only stationary Markov policies.

For a given policy π define the *value function*

$$V^\pi(s) = \mathbb{E}_\pi(\sum_{n=0}^{\infty} R_{t+1} | S_0 = s) \quad (2)$$

i.e. the expected reward obtained when following policy π. If s_t is an *terminal* state — a state from which the process never leaves — we take $R_{t+1} = 0$, which implies that $V^\pi(s) = 0$ if s is terminal.

We say that a policy π^* is optimal if

$$V^{\pi^*}(s) \geq V^{\pi}(s) \qquad \forall s \text{ and policies } \pi.$$

It can be shown that for finite Markov decision processes such a policy exists uniquely. Under the optimal policy π^* let $V^*(s) = V^{\pi^*}(s)$. Say we know $V^*(s)$ for all s, can this be used to calculate an optimal policy? The answer is yes. One can prove that

$$V^*(s) = r(s, \pi^*(s)) + \sum_{s'} P_{\pi^*(s)}(s, s')V^*(s') \tag{3}$$

$$= \max_{a \in A}(r(s, a) + \sum_{s'} P_a(s, s')V^*(s')). \tag{4}$$

The equations are known as the *Bellman equations* and they can be used to show that $\pi^*(s)$ is given by the action a which achieves the above maximum. So if we can find a way to calculate the optimal value function we get the optimal strategy. In small state spaces the value function can actually be calculated but in large spaces approximation is the best one can hope for.

2.2 Q-Learning

Q-learning is an algorithm which can be used to approximate the optimal value function and corresponding strategy. Here we will assume that the underlying MDP is terminal.

In order to specify the algorithm we need some definitions. Define the *action-value function* as

$$Q^{\pi}(s, a) = r(s, a) + \sum_{s'} P_a(s, s')V^{\pi}(s'). \tag{5}$$

That is the expected reward of starting in state s taking action a and then following the policy π. For terminal states s we set $Q(s, a) = 0$ for all $a \in A$. Furthermore, define the optimal action-value function as

$$Q^*(s, a) = (r(s, a) + \sum_{s'} P_a(s, s')V^*(s')) \tag{6}$$

i.e. the expected reward of being in state s taking action a and then following the optimal policy π^*. We see that $\max_{a \in A} Q^*(s, a) = V^*(s)$, and with the same argument as above $\pi^*(s) = \max_{a \in A} Q^*(s, a)$.

We can now specify the Q-learning algorithm:

(i) Initialize $Q(s, a)$ for all s, a, learning rate $\alpha_t(s, a) \in (0, 1]$, and decide on some policy e.g. ϵ-greedy.

(ii) Repeat for each N episodes (start $\rightarrow$ final state):
While s_t not terminal:

$$Q(s_t, a_t) \leftarrow (1 - \alpha_t(s_t, a_t))Q(s_t, a_t) + \alpha_t(s_t, a_t)(r_{t+1} + \max_{a \in A} Q(s_{t+1}, a)) \tag{7}$$

It is known that for stationary finite Markov decision process (and some additional assumptions) the Q-learning produces estimates converging to the optimal action-value function.

3 Blackjack as a Markov Decision Process

In order to test this reinforcement learning framework we have chosen to work with the simplest form of Black Jack. We assume the following setup:

- One player against the dealer, with player staking one unit on each hand;
- two actions possible: ask for another card, or stay;
- 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 player going bust it is known as a *usable* ace, the same goes for the dealer.

The goal of the player is to beat the dealer in one of the following ways

- Get 21 points on the first two cards, knows as a blackjack, without a dealer blackjack. **Net profit**: 1.5 times stake;
- Reach a final score higher than the dealer without exceeding 21. **Net profit**: stake;
- Dealer gets points exceeding 21 and player does not. **Net profit**: stake.

The house always plays according to the same strategy: draw cards until it has a card sum greater than or equal to 17. The game starts with the dealer giving the player two cards (visible) and himself two cards (one visible, one hidden). The player is then allowed to take actions until done, after which the dealer follows his strategy until done. If player's points exceeds 21 his stake is

lost regardless of dealers outcome (this is where the house edge comes from); if player's points equals dealer's points this is a push and stake is returned to player; otherwise payout is made according to above rules. Standard practice is that the cards are drawn from 6–10 decks, and we implement these situations as well as when the cards are drawn from an infinite deck so that every card has same probability of being drawn at all times.

3.1 As a Markov Decision Process

Next we fit the above situation in to the Markov decision process framework. We do it in two ways: one with a large state space such that all MDP model assumptions are satisfied; one with a smaller state space, where the resulting process is no longer stationary. In both cases actions and rewards are as specified above, and we need only specify the state space in order to have the model.

3.2 Stationary Markov Decision Process

We begin by noting that the color and suite of the cards does not matter, only their numerical values. We use this as the cards identifier, with aces equal to 1. We use the following representation of the state space

$$S = \{(s_{p1}, \ldots, s_{p10}, \Sigma_d) \,|\, i(s_{pi} - 1) \leq 21 \text{ for } i = 1, \ldots, 10, \text{ and } \Sigma_d \leq 26\}.$$

Hence, an element of the state space is a vector of length 11, where the 10 first elements indicates the number of each card the player is holding, and the last element indicates the dealer's *visible* card sum.

Note that for a card of value i, the maximum number s of cards with this value the player can have is such that $n - 1$ keeps the player's card sum less than or equal to 21, but drawing one more card will lead to a bust (assuming the player has no cards of other values). At first glance this state space seems enormous, and while that is true the *effective* state space is much smaller, i.e. the states which the system realistically will visit is relatively small. Also, with this state representation it is always possible to determine if a state is terminal—we need only check if the player or dealer is bust, or if dealer's card sum exceeds 17. The rewards $R_1, R_2, \ldots$ follows the above payout specification, with $R_t = 0$ if s_t is not terminal. It is also possible, albeit tedious, to calculate the transition probabilities and to show that the resulting process is a Markov decision process.

3.3 Non-stationary Markov Decision Process

Most strategies in blackjack involves keeping track only of your own card sum (with some possible extensions) and not every card you are holding. Hence, it seems natural to base the state space on this observation. It is also practical to know if the player has an usable ace, so we include this in the state representation. Let

$$S = \{(\Sigma_p, a_p, \Sigma_d)\}$$

where Σ_p, Σ_d is the card sum of the player and the dealer, and a_p indicates if the player is holding a usable ace or not. This seem to be a popular representation of the state space for blackjack across the RL-community. Although natural and popular, this representation does not yield a stationary Markov process. Consider,

$$s_0 = (21, 1, x), \qquad s_5 = (21, 1, x).$$

The first situation is a blackjack and the second is not. Hence, the expected rewards of the action "staying" will differ for the two states

$$r(s_0, \text{stay}) = 1.5 \neq r(s_5, \text{stay}).$$

For finite decks this state representation also yields non-stationary transition probabilities, however they are stationary in the infinite deck case.

4 Implementation

In implementing a Q-learning algorithm for Blackjack, we face a number of choices. First, as discussed in Sections 3.2 and 3.3, is the choice of state space. Second, the parameters α and ϵ of the algorithm, which may be functions of the current state s and the action a. Third, the implementation itself—how do we implement the algorithm in code? The first and third points are interconnected here, so we deal with the second point first, then the other two.

4.1 Parameter choices for Blackjack Q-learning

The (possibly) state- and action-dependent learning rate $\alpha_t(s, a) \in (0, 1]$ determines to what extent we update the Q function each time we visit state s and take action a. A small α_t means that Q changes slowly from its initial value (which is an additional parameter to choose), and a larger α_t means

that we put more emphasis on the immediate (well, at $t+1$) reward and currently optimal action of the state s' we transition to. Intuitively, our estimate of $Q(s,a)$ should improve with the number of times we have visited s and taken action a, meaning that it becomes less important to update the function in this state-action pair. Vice versa, in state-action pairs we have not encountered many times before, our estimate of the function value is uncertain, and we should put more emphasis on the reward we received for taking action a, as well as the currently known best action out of the state s' we transition to. For our Blackjack setup, we may note again that the immediate rewards are 0 in all states except for a terminal state, where the payout is made. By convention the function Q is defined to be zero for terminal states. Thus, the update rule 7 becomes

$$Q(s_t,a_t) \leftarrow \begin{cases} (1-\alpha_t(s_t,a_t))Q(s_t,a_t) + \alpha_t(s_t,a_t)R_{t+1}, & s_{t+1} \text{ terminal} \\ (1-\alpha_t(s_t,a_t))Q(s_t,a_t) + \alpha_t(s_t,a_t)\max\limits_{a\in A} Q(s_{t+1},a), & s_{t+1} \text{ non-terminal.} \end{cases} \tag{8}$$

A choice of $\alpha_t(s_t,a_t)$ that has the intuitive properties mentioned above is $\alpha_t(s_t,a_t) = \#[(s=s_t,a=a_t)]^{-1}$, i.e. the reciprocal of the number of times action $a=a_t$ has been taken in state $s=s_t$. This choice of α_t satisfies the convergence criteria $\sum_t \alpha_t(s_t,a_t) = \infty$ and $\sum_t \alpha_t^2(s_t,a_t) < \infty$ outlined in e.g. **?**. Further investigation shows that one can do better than this when the function Q is updated asynchronously, i.e. one (s,a)-pair at a time. **?** show that in this case, the optimal choice is $\alpha_t(s_t,a_t) = \#[(s=s_t,a=a_t)]^{-\omega}$, where $\omega \approx 0.77$. This is the choice of α_t we choose, although it should be noted that the scenario considered by **?** included a discount factor $\gamma < 1$.

Another important choice in implementing Q-learning is the choice of policy—what action should we take in a given state? Under certain conditions, our learned Q will eventually guide us to the optimal policy, but this is not the case initially. If we used the greedy policy of always choosing the action a with the highest value of Q when in state s, an unlucky start may lead us to never explore some states and find (approximately) the true value $Q^*(s,a)$ for all actions a available when in state s—some perhaps more valuable than the greedy action chosen. Indeed, the second condition for convergence to Q^* is that all pairs (s,a) are visited infinitely often, asymptotically (**?**). Commonly, this is done by choosing a random action with some small probability ϵ, rather than choosing the greedy action. Again, it is reasonable that if we have visited a given state s many times, and also taken all actions our of this state many times, our estimate $Q(s,a)$ for all actions a available in s should be fairly certain. Thus, there is less need to choose a random action; we may go with the greedy choice with a high probability.

There are several ways of implementing such ϵ-*decay*. We choose the rather simple option of choosing a random action when in state $s_t = s$ with probability $\epsilon_t(s_t) = \frac{c}{\#[(s=s_t)]}$ for some positive constant $c \leq 1$, though it should be noted that more advanced options such as Boltzmann exploration exist (**?**).

4.2 Software implementation

Our initial idea for implementing a Q-learning algorithm for Blackjack was to find a pre-existing code base that could simulate a Blackjack environment, enabling us to focus mostly on the reinforcement learning aspect of the project. AI Gym (`https://gym.openai.com/`) is "a toolkit for developing and comparing reinforcement learning algorithms", which can be installed as a Python library and which offers a simple Blackjack environment—indeed, seemingly the same environment as that in **?**. In this version of Blackjack, cards are dealt from an infinite deck and one is provided the state representation (player sum, dealer's 1 showing card, player has usable ace), much like that described in Section 3.3. On closer inspection of the source code, however, we found that this environment was lacking. With only the (first) visible card representing the dealer in any state, some state transitions will go from one state (call it s) to itself when the episode ends (and a reward is given). This means that some terminal states are missing, and further that the action 'hit' may be permissible at the first entry into s but not the subsequent one. For these reasons, $Q(s, a)$ will be given an erroneous value in its updates for these states. We also found the infinite deck setting to be less interesting, since it makes counting cards impossible.

Thus, a somewhat large effort was put into improving the Blackjack environment class from AI Gym. We replaced the dealer's first visible card in the state representation by the dealer's visible card sum, made it possible to have an integer number of decks in addition to infinitely many, and also made some other smaller changes. These changes were made with inheritance in mind, as our next effort was spent on making a subclass of this base class; the subclass implementing the extended state space discussed in 3.2. We have made our implementation publicly available at `https://github.com/Ethorsn/SF2957_project` and are considering a submission (pull request) of it to AI Gym.

Insert joke about the movie 21 or the MIT Blackjack Team

5 Results

In this section we let the Q-learning algorithm act on the the state spaces described in section **??**. First, let $S_{hand} = \{(s_{p1}, \ldots, s_{p10}, \Sigma_d)\ s_{pi} \in$

$\{1, 2, \ldots, 21\}, \Sigma_d \in \{1, 2, \ldots, 10\}\}$ and $S_{sum} = \{(\Sigma_p, a_p, \Sigma_d)\}$ denote the two state spaces. For abbreviation we may call these "hand"- and "sum"- environments or state spaces respectively. Also, we will simply call our Q-learning algorithm the "algorithm".

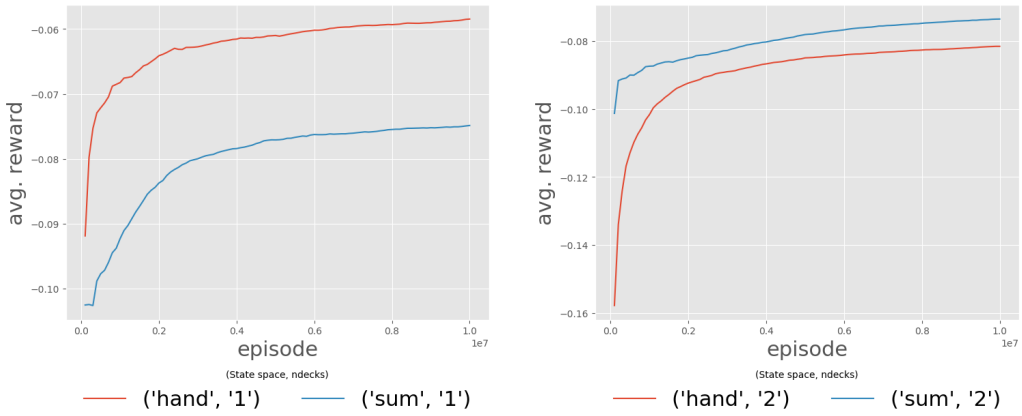
We will employ the algorithm on the two state spaces. In each simulation the values of the matrix Q are initialized to zero. The constant ϵ is set (dynamically) according to the algorithm and theory described in Section 4.1. The number of episodes is set to 10^7 if not stated otherwise.

In Figure 1 we display the average return by episodes, of the algorithm for different number of decks. The x-axis is scaled to a proportion of the number of simulations performed. In the top left Figure of 1, denoted 1a, we can see the performance of the Q-learning algorithm for 1 deck. Here, it becomes increasingly important to remember what cards that have been played. This is to be expected. When playing using one deck, there are only 4 of each card with the exception of all suites which are ten. Conditioned on the dealers outcome (where only one card is visible), the probability of seeing a arbitrary sequence of cards is relatively large for small deck sizes. However, as the deck size increases the information of what cards that are in the agents hand become redundant. This can be seen subsequently in Subfigures 1b, 1c and 1d.

There is a specific shape to the left tail of the avg reward on the hand space seen in all figures of Figure 1. The behavior can be motivated by the cardinality of the hand space. In the first iterations, the algorithm visits a new state *all the time*. It will take random actions almost always, just because of the cardinality. However, we can also note that the algorithm on the hand space learns faster in comparison to the algorithm on the sum in terms of average rewards.

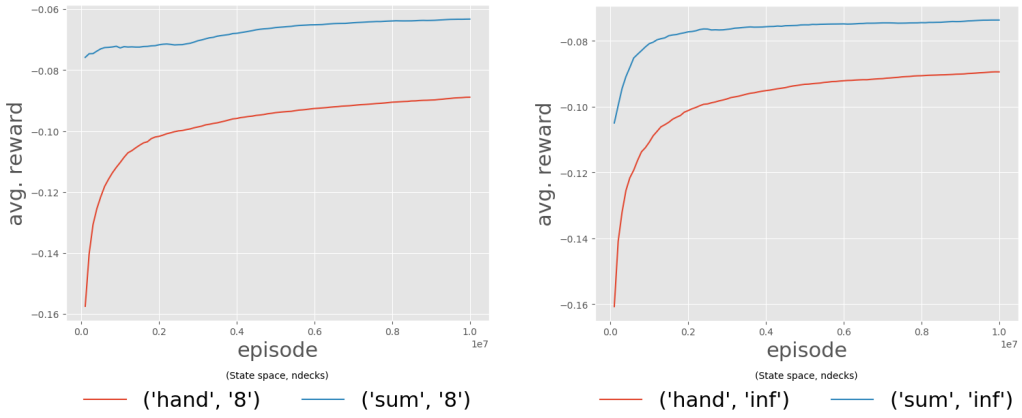
One can note that in all subfigures of 1, the algorithm seem to have some reward left to accumulate. It has not yet converged in neither state space. More simulations where not performed because of the lack of time and computational power.

Since the cardinality of the hand space is much (!) larger then the sum's, the number of states the algorithm must explore will be larger. In Table 1 we show the number of states the algorithm has explored together with the effective training time for the specific environment and the number of decks in play. We can clearly see that the number of explored states grows large for the hand state when the number of decks is increased. The number of states explored by the algorithm on the hand state space seem to converge to a level of 70000. As noted in section 3.2, the number of effective states to be explored are not as grand as the actual state space. Hence, it is not surprising that the algorithm converge to level of states that are both necessary and



(a) Avg. return with 1 deck

(b) Avg. return with 2 decks



(c) Avg. return with 8 decks

(d) Avg. return with infinite decks

Figure 1: The average return of the Q-learning algorithm for $1, 2, 8$ and ∞ number of decks.

valuable to explore. The algorithm fluctuates below 700 explored states for all deck sizes when learning on the sum space.

The training time for the algorithm on the hand space is large compared to the sum's. There is an indication that there is a 50% increase in training time for the algorithm on the hand state compared to the sum. Note that the effective training time is simply the time it took to train this specific instance. All other measures of training time (which might be more accurate) are omitted because of the lack of computational power and time.

# decks	$\#S_{hand}$	t_{hand}	$\#S_{sum}$	t_{sum}
1	7651	1510.681	641	1003.765
2	38266	1565.314	677	1079.656
8	70034	1606.944	695	1079.626
inf	70307	1636.044	671	1097.971

Table 1: The number of explored states $\#S$ and the training time t (seconds) for a given number of decks in play and its respective state space. The number of simulations was set to 10^7.

Since estimates of Q-learning will converge to the true optimal value-action function we would like to visualize these said values. However, an obstacle in comparing the algorithm between the two spaces the cardinality of the hand poses a problem. The two state spaces do not align. Therefore, we will perform a (somewhat arbitrary) aggregation scheme. Each state of S_{hand} will be transformed to a sum of the card values of that hand and if the ace was usable or not. We will take a simple average of the q-values when doing so. Post this aggregation scheme, the results from the algorithm on the hand space are now on the same form as the sum's. Then, we use our policy and take the maximum value of the two q-values corresponding to different action values of that state. In Figures **??** through **??** we have visualized the aggregated q-values for the hand state and the sum states, for different a deck size of size 1. The figures are divided into the case when the player holds a usable ace or not. Since the states where $\Sigma_d > 11$ are terminal we chose not to include them in this visualization.

From Figures 2 through 5, it is clear that holding a card sum of 21 indicates a large reward when using our policy. This is almost regardless of what the dealer is showing. This is satisfactory since it should be the best possible hand for an agent to hold. The only exception is when the dealer draws an ace as its first card. As seen in Figure 3, the value our policy when the dealer is showing an ace (first card) is relatively low. This is true through all states of the player sum. The same phenomena is seen in Figure 4 which,

to some extent, eliminates the possibility of that this would be a result of the aggregation scheme.

In general, when an ace is usable it cuts losses. This can be seen by comparing the pair of Figures 2 and 4 compared to Figures 3 and 5. The amount of losses are generally less (less blue) in each state space for usable aces. This can be motivated by the flexibility a usable ace gives the player. By using the same pairs of Figures, it is also apparent that if the player has no usable ace then the value of the policy is higher for values close to 21. The slope of the "hill" is less when the player has no usable ace.

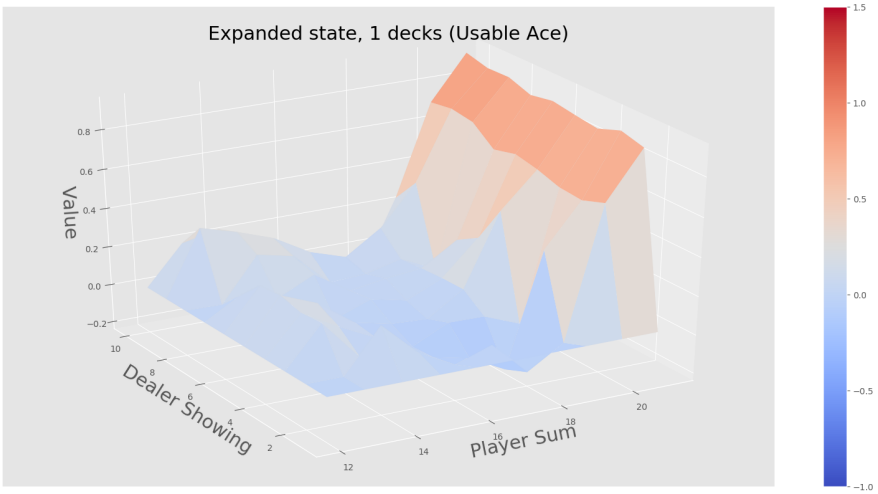


Figure 2: Aggregated q-values when the player has an usable ace.

6 Summary

As we saw in section 5 the performance of the Q-learning algorithm was no good on the expanded state space. The average return of the algorithm was always higher when it learned on the state space using the sum. The algorithm using the state space of the hand was better when 1 deck was used, which is nice but not applicable in real life when blackjack is played.

Another loss was also the increase in computational time by a factor of 50%. This is not scalable for large experiments. It could possibly be

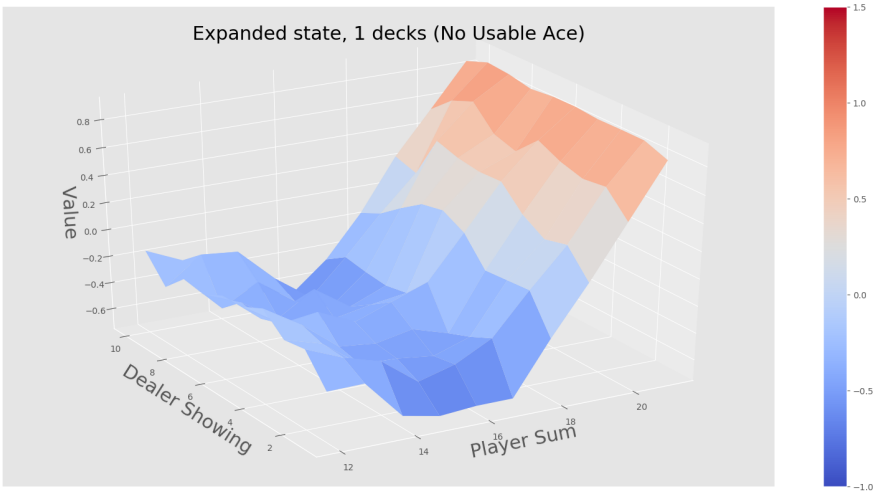


Figure 3: Aggregated q-values when the player has no usable ace.

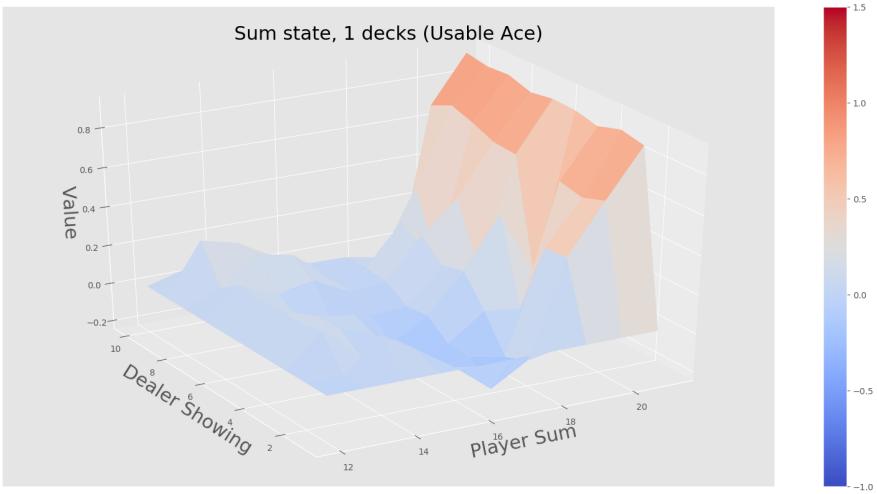


Figure 4: Q-values when the player has an usable ace.

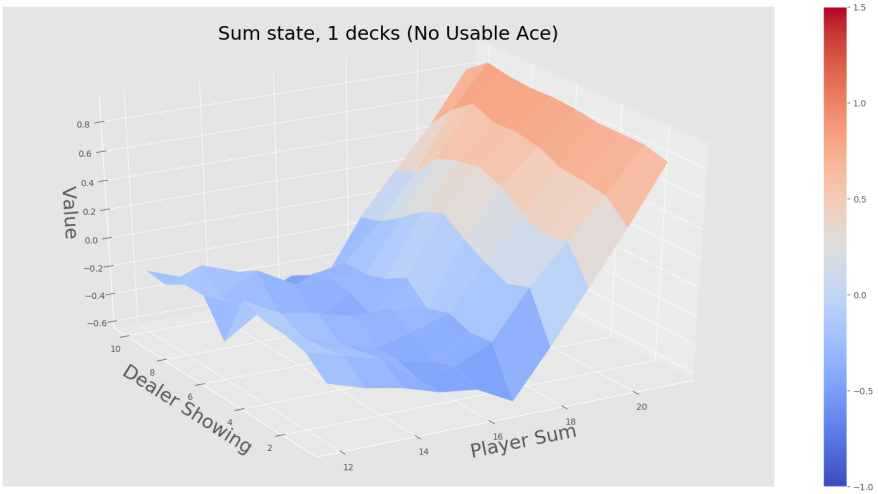


Figure 5: Q-values when the player has no usable ace.

circumvented by creating more efficient algorithms or by using another API instead of "Gym".

There are a number of interesting extensions to this problem. One could include several agents, playing on the same "table". Could the the agents then learn to interact (react) to what card the others are showing? Furthermore, the concept of counting cards is also interesting. If we introduce a memory of a general type, say the number of played cards of each type, would the agent learn how to count cards? Surely the extended state space should provide a lot more information in such a scenario.

- What have we done
- Future work