Playing Stylised Blackjack with Q-learning

(Group B3) Tham Yik Foong and Qi Fang Rui
School of Physics and Astronomy, University of Nottingham.
(Dated: December 2019)

This project address the task assigned to train an agent to play a stylised version of Blackjack using Q-learning algorithm. Reinforcement learning algorithm usually performs generally well in probabilistic environment; technique such as Q-learning algorithm is specifically handy for environment with small state-action spaces. Therefore, we have implemented Q-learning as basis for our agent while also adding some extra capabilities to aid the performance of our agent. To compare the performance, we compare our agent with a Q-learning agent without extra capabilities and a rule-based agent. We discover that even our agent can perform better than a basic Q-learning agent, but it only performs equally as a rule-based agent.

I. INTRODUCTION

A. Stylised Blackjack

Blackjack is one of the oldest casino games and it remains popular today. The goal of Blackjack is to accumulate value of the player's hand as close to 21 as possible without going over.

In this project, a stylised Blackjack is introduced for simplicity where there is no opponent. This version of Blackjack follows the majority of the classic Blackjack rules, except: there is only a single-player, dealer only deals the cards, and the game is played with a number D of decks of poker cards shuffled together at the start of the game. Only two possible actions can be performed, 'hit' which player can draw an additional card and 'stick' when player is satisfied with their score in hand and decide to end the hand. Once the player draws a card and cause total value in hand exceed 21, new hand will be dealt and no reward will be received. Note that Aces are valued 11 unless value in hand goes over 21 and then they are valued 1.

When a hand is finished, let a card's value be C1, and the reward given to agent is

$$R = \begin{cases} (C_1 + \cdots + C_{n+1})^2 & \text{if } C_1 + \cdots + C_{n+1} \leq 21 \\ 0 & \text{if } C_1 + \cdots + C_{n+1} > 21 \end{cases}$$

Where, the reward R is quadratic in the total if card value does not exceed 21, and zero otherwise. The game lasts until all the cards are used. This defines an episode of the game.

B. Q-learning

Q-learning is known as an off-policy temporal difference control algorithm [1][2], which its objective is to approximate the optimal action selection strategy using Q function. Q-learning learns what rewards can be obtained after taking specific action in a given state. In this project, a Q-table [3] is created to help us keep track of and find the best action for each state. Q-table is essentially a mapping table between state-action and estimated future rewards. By finding the best of all possible actions, it can maximize the expected reward. Q-learning updates $Q(s, a)$ iteratively using the Bellman equation by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[R(s_t, a_t) + \gamma \max_a Q(s_{t+1}, a_t) - Q(s_t, a_t)] \quad (1)$$

Initially, we explore the environment and update the Q-table by the rewards of possible actions. Q-learning utilizes ϵ-greedy strategy and always take greedy actions (action with maximum value in a state) in exploitation step.

II. IMPLEMENTATION DETAIL

A. State-Action Space

State in Markov Decision Process represents the information that describes the current environment. Selecting a state to process by our policy is crucial as it impacts how an agent views the environment and act on it. A traditional Q-learning algorithm is particularly handy for environment with small state-action spaces. Therefore, it makes sense to select a small state space and at the same time retain useful information.

We model environment state with two characteristics: total card value in player's hand (from 2 to 21) and distribution of cards. Note that distribution of cards is essentially registering the occurrence of cards. However, given that occurrence of cards rely on the number of decks shuffled together, and tracking each card will drastically increase the dimension of Q-table. Hence, the approach we took is defining two categories of cards: low-value cards (ace, two, three, four, five, six, seven) and high-value cards (eight, nine, ten, Jack, Queen, King). Two counters are used to register the number of occurrence of low-value cards or high-value cards. We then compare those occurrences and categories them into five states: excessive or more occurrence of low-value cards, same amount of occurrence, more or excessive occurrence of high-value cards. This help agent to understand more

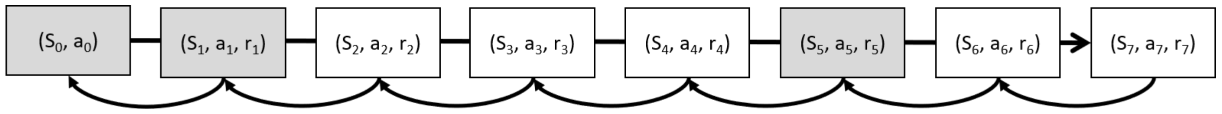


FIG. 1. Update state-action value pair along the whole episode using Q-learning algorithm. Boxes with different shade indicate different hand

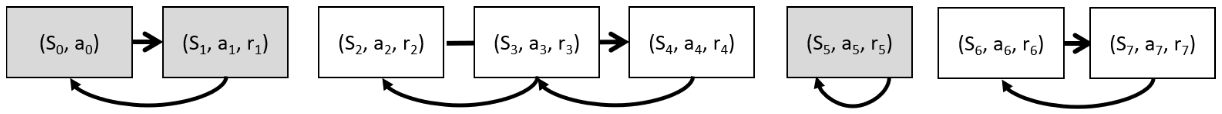


FIG. 2. Divide episode into sub-episodes. Update state-action value pair with Q-learning algorithm and update end state of sub-episodes using Eqs. (3). Boxes with different shade indicate different hand

about the likelihood of next card, for instance: if there is an excessive occurrence of high-value cards, meaning that more low-value cards remain in the deck, and intuitively, it is more likely that performing 'hit' will not cause player's hand to go over 21.

B. Exploration Policy

We adopt ϵ-greedy policy [1] as our agent's exploration policy. ϵ-greedy policy suggests that every time an action is selected, it has a probability of ϵ a random action is selected. Else, greedy action will be taken: which agent will select an action with the greatest state-action value. This help agent to discover actions that are possibly beneficial. Exploration and exploitation of the environment must be balanced to ensure the model does not over-fit or under-fit. ϵ-greedy policy usually involved a decay factor k that scale down ϵ over time by [4]:

$$\epsilon \leftarrow \epsilon_{min} + (\epsilon_{max} + \epsilon_{min})e^{\frac{-k(episode)}{total\_episode}} \tag{2}$$

where ϵ_{min} and ϵ_{max} is the minimum and maximum of ϵ. *episode* is the current episode and *total_episode* is the total episode, *episode* and *total_episode* are introduced to standardize decay of epsilon depending on the total number of episodes. This causes ϵ to decrease by an exponential rate. This implies that agent tend to perform exploration action more often at the beginning of training and take a minimal amount of exploration action defined by a minimum factor toward the end of training.

C. Divided Episode

The stylised Blackjack environment provides an option to configure how many decks are used and shuffled together, and this finite episode end when all cards are distributed. Initially, we update state-action value pair corresponds to the current state and its next state along with the episode (Fig. 1). We discover that the performance of our agent vary arbitrarily. This is due to: if the agent performs 'stick' or player's hand exceed 21, the action does not dictate the next state which new two cards are drawn randomly from the deck. Thus, propagate value of the next irrelevant state-action value is considered as noise. Therefore, to address this issue, we decided to further divide the whole episode into sub-episodes (Fig. 2) starting from two new cards is drawn and ending with the agent perform 'stick' or player's hand exceed 21. At the end of sub-episode, update of state-action value pair does not take the next state into consideration:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[R(s, a)] \tag{3}$$

D. Experience Replay

Another method we implemented to improve our agent is using experience replay buffer [4]. This approach work by storing past experience in a buffer instead of throwing them away after training. Following that, at a fixed time steps, randomly sample a batch of experience from the buffer and updates their state-action value again. Note that buffer can only hold a certain length of experiences, adding new experience into a full buffer will dequeue old experience to prevent over-fit with old experience. Experience replay help to speed up training, allow agent to learn on sparse reward schemes and avoid overfitting of recent experience.

III. MODEL EVALUATION

To evaluate the performance of our agent, we have implemented three agents whereas two of the agents are implemented as additional baselines. The first agent being a Q-learning agent with all the capabilities mentioned above. The second agent implemented Q-learning algorithm but without the capability of divided episode and experience replay. Lastly, we created a rule-based agent that always performs 'hit' when card value is below 12 and 'stick' otherwise.

To ensure the 'fairness' of our model evaluation, deck are shuffled with the same permutation using the same seed during training of each agent. We run 1000 episodes with 5 decks of card shuffled together, then consider first 900 as training episodes and measure the performance based on the mean of total reward of the last 100 testing episodes.

IV. RESULT

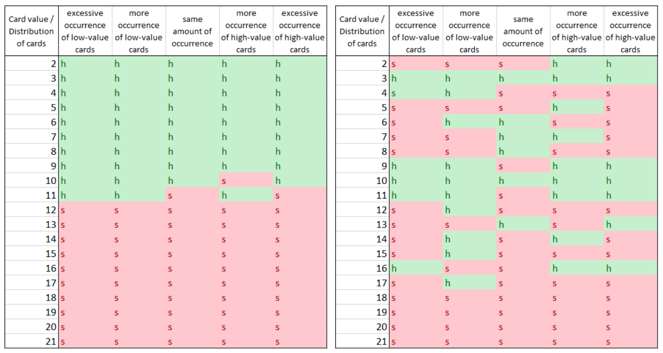
Ultimately, we yielded results below by running 1000 episodes with 3 agents.

TABLE I. mean of total reward of each episode from test episodes.

Agent	Mean of total reward
AG_{Q_extend}[a]	30848
AG_Q[b]	28160
AG_{rule_based}[c]	31422

[a] Q-learning agent with extend capabilities
[b] Q-learning agent without the capability of divided episode and experience replay
[c] rule-based agent

Notice rule-based agent achieved the highest reward and followed by our Q-learning agent with extended capabilities. This proves that our agent can perform a good action, even it is not the best action, by learning from this probabilistic environment without human interference. Meanwhile, Q-learning agent without extra capabilities perform relatively weak from the comparison and this once again proved that breaking down episodes into sub-episodes and experience replay really help to improve the performance of an agent. If we look at the Q-tables at the end of the training below:

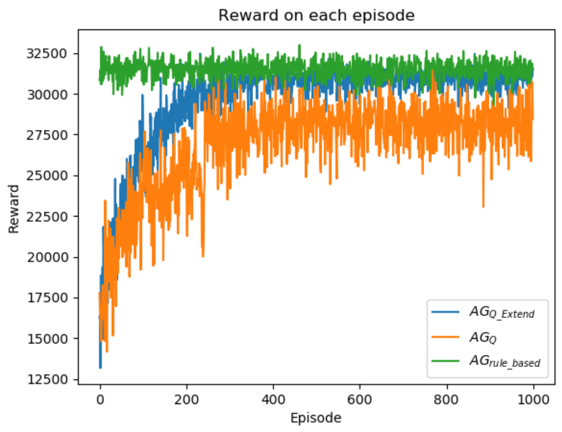


Card value / Distribution of cards	excessive occurrence of low-value cards	more occurrence of low-value cards	same amount of occurrence	more occurrence of high-value cards	excessive occurrence of high-value cards
2	h	h	h	h	h
3	h	h	h	h	h
4	h	h	h	h	h
5	h	h	h	h	h
6	h	h	h	h	h
7	h	h	h	h	h
8	h	h	h	h	h
9	h	h	h	h	h
10	h	h	h	s	h
11	h	h	s	h	s
12	s	s	s	s	s
13	s	s	s	s	s
14	s	s	s	s	s
15	s	s	s	s	s
16	s	s	s	s	s
17	s	s	s	s	s
18	s	s	s	s	s
19	s	s	s	s	s
20	s	s	s	s	s
21	s	s	s	s	s

Card value / Distribution of cards	excessive occurrence of low-value cards	more occurrence of low-value cards	same amount of occurrence	more occurrence of high-value cards	excessive occurrence of high-value cards
2	s	s	s	h	h
3	h	h	h	h	h
4	s	h	s	s	s
5	s	s	s	h	s
6	s	h	h	s	s
7	s	s	h	h	s
8	s	s	h	s	s
9	h	h	s	h	h
10	h	h	h	h	h
11	h	h	s	h	h
12	s	h	s	s	s
13	s	s	h	s	h
14	s	h	s	h	s
15	s	h	s	s	s
16	h	s	s	h	h
17	s	h	s	s	s
18	s	s	s	s	s
19	s	s	s	s	s
20	s	s	s	s	s
21	s	s	s	s	s

FIG. 3. (left) Q-table of AG_{Q_extend} and (right) Q-table of AG_Q

It is clear that AG_Q's Q-table contains more 'noises' and this is due to the propagation of irrelevant next state-action value to current state-action value. In contrast, AG_{Q_extend} somehow resemble the action taken by rule-based agent. Additionally, below are a graph showing the reward gained from our agents on every episode.



It shows that our agents quickly discovered how to play the game in first 200 episodes, and took less and less exploration step toward middle of training due to the decay of epsilon.

V. FUTURE WORK

Further improvement can be done to improve our agent, for instance: a more sophisticated state can be used, such as memorizing all the occurrence of cards. This help to retain more useful state information for our agent without compressing the information like what we have implemented in this project. Nevertheless, this is difficult to implement as mentioned in section *II.A*. However, a deep Q-learning algorithm is able to generalize to such large state-action spaces by allowing it to learn and approximate Q-matrix instead of storing whole Q-table in memory. Deep Q-learning generally outperforms traditional Q-learning in case of large or infinite state-action spaces [4].

Another improvement can be done is to further optimize our parameter, for instance: learning rate, e-greedy policy-related parameters, experience replay related parameters, and et cetera. Furthermore, a better state representation will also help to improve the performance of our agent.

VI. CONCLUSION

To summarize, this project has shown how we implemented Q-learning algorithm to seek the optimal policy to play on a stylised version of Blackjack. We discussed how we defined the state-action spaces and exploration policy, using experience replay and divided episode to improve the performance of our agent. Further, we also compare our agent to other agents as a baseline. As result, we noted that our agent can learn a considerably good policy, but not an optimal policy in such probabilistic environment. Lastly, using of DQN, a more robust state representation can be a possible improvements for our agent.

Appendix A: Code

Code and source of this paper can be found in this Github repository: `https://github.com/zfoong/RL-Blackjack`

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction, second edition* (MIT Press, 2018).

[2] A. Karpathy, Temporal difference learning gridworld demo, online Demo of Temporal Difference Learning.

[3] Openai gym and python for q-learning - reinforcement learning code project (2018), tutorial of Q-learning.

[4] A. Wu, *Playing Blackjack with Deep Q-Learning*, Tech. Rep. (Stanford CA 94305, 2018).