Empirical Evaluation - Difference between Temporal Difference and Monte Carol Control in short-trajectory game

Yulin Xue
y79xue@uwaterloo.ca
The University of Waterloo
Waterloo, ON, Canada

Abstract

In reinforcement learning, Temporal Difference evaluation is always preferable to Monte Carlo Control, because Temporal Difference evaluation does outperform Monte Carlo Control in most cases. Temporal Difference requires fewer trajectories and can reduce the impact of bias estimation over multiple iterations. Therefore, I wonder if Monte Carlo can perform better in a particular environment. The trajectory has to be short and sensitive to bias in estimation. The card game blackjack came to my mind. Each trajectory in blackjack is very short as the player need to keep the sum of cards below 21. Blackjack offers stronger randomness than other games, so the bias caused by estimation is less likely to be reduced over iterations. Thus I planned to use different RL algorithms to tackle blackjack problems and observe their performance to prove or disprove my assumption. The algorithm includes: Q-learning with Temporal Difference Evaluation, GLIE[1] with Monte Carlo Control and SARSA

Introduction

Rule of blackjack

Blackjack is one of the most popular casino banking games also known as Twenty-One, which uses decks of 52 cards. In blackjack, there is no competition between players. Instead, each player competes against the dealer. The object of the game is to get closer to 21 than the dealer but will lose immediately by exceeding 21. In each round, all players and the dealer receive two cards and the dealer needs to reveal the first card. Players can choose either Stick[2] or Hit[3]. If the total of players exceeds 21, it is known as a bust, lose immediately. When all players choose to stick, the dealer will start taking cards until the sum of cards reaches 17. Whoever is closer to 21 than the dealer wins. Number cards count as their number, face cards[4] count as 10, and Ace can be counted as either 1 or 10.

Motivation

Not surprisingly, there is an optimal strategy for blackjack. It does not guarantee you a win, but if you can make the right action in every situation, you can maximize your benefits. As a result, blackjack provides a very good platform for testing the RL algorithm. The goal is to see if the agent can learn from the environment by interacting with environment and discover the optimal strategy eventually. We can compare the win rate of different RL algorithms with the win rate of the optimal strategy to see if there is room for improvement in the algorithm. Also, we can compare the agent's action with the optimal action. The most important thing is that we can compare the win rate of Temporal Difference with the win rate of Monte Carlo Control to prove or disprove my assumption mentioned in the abstract.

Algorithm

In this paper, we will propose three different algorithms to solve the blackjack: Q-learning, GLIE, SARSA. The Q-learning algorithm is one of the most classic in RL, and it's a specific Temporal difference evaluation for learning Q-values. GLIE is very similar to Q-learning, but GLIE uses Monte Carlo evaluation to update Q-values. SARSA is another popular RL algorithm when losing to the agent is expensive. Also, we propose three different strategies for the agent to choose an action: epsilon greedy, softmax and optimistically.

Related Work

Monte Carlo ES

Sutton and Barto evaluated the optimal strategy for blackjack by using Monte Carlo ES. As they use the same blackjack rule as we do in our experiment, we can use their results as a baseline. They choose to fix the action to be stick when the sum of cards is 20 or 21, which is reasonable in blackjack(Sutton and Barto 2018). However, the agents do not have any restrictions or additional information in our experiments. The goal is to observe agents' performance in unknown environments.

[1] GLIE: Greedy in the Limit with Infinite Exploration

[2] Stick: Take no more card

[3] Hit: Take another card

[4] Face Cards: jack, queen, king

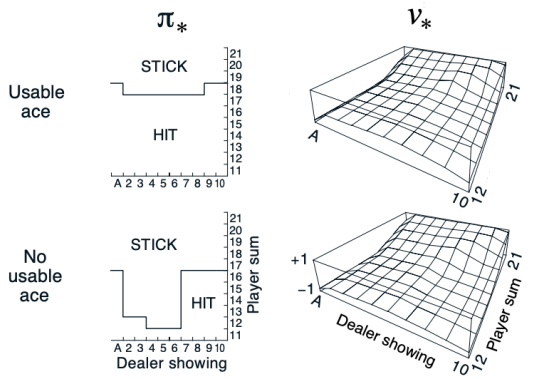

Figure 1: Optimal strategy derivated from Monte Carlo ES (page 100)

Blackjack in Neural Network

The author here also implemented Q-learning and SARSA with epsilon-greedy to solve the blackjack problem. Additionally, the author uses a Neural Network to determine the Q value. Also, the author uses punishment when busting instead of losing in SARSA, causing the agent to make more conservative decisions. In the end, the author achieved an average win rate of 40.6%.

(Perez-Uribe and Sanchez 1998)

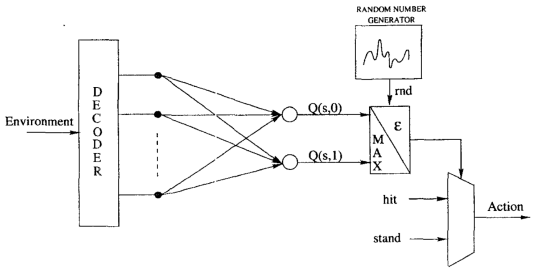

Figure 2: Neural Network Architecture

Temporal-difference search in computer Go

Deepmind introduces a new approach to solving the game of 9x9 Go, called temporal-difference search, inspired by temporal-difference learning. They found out that the TD search can be more efficient than Monte-Carlo tree search if the number of the simulation is the same. In addition, the Monte-Carlo search has to wait until the end of the trajectory to make update. In their framework, they used TD learning that can bootstrap from subsequent values, which can improve performance and reduce variance (Silver, Sutton, and Müller 2012). Deepmind's discovery that TD search performs better than Monte Carlo tree search is reasonable. Their research focuses on long-trajectory games such as Go and cheese. On the contrary, our experiment focuses on the short-trajectory game such as blackjack.

Methodology

Environment

Action Space	Stick(0) or Hit(1)
Observation Space	Tuple(Discrete(32), Discrete(11), Discrete(2))
Reward	Win(+1), Lose(-1), Draw(0)
Terminal State	Player choose to stick or bust
Import	gym.make("Blackjack-v1")

The observation consists of 3 states as followed:

s_1	the player's current sum
s_2	the value of the dealer's showing card
s_3	whether the player holds a usable ace

It is worth noting that when s_3 is True (holding an Ace), the Ace will be counted as 10 for the player's current sum. For example, if the player holds an Ace and an 8, the dealer reveals a 5, the state will be (18, 5, True). Then if the player chooses to hit and take a 6, the state will be (15, 5, False)

Select Action

In the RL algorithm, the strategy to choose the next action is always very important, as it may lead to different results. It is difficult to say which strategy for choosing actions is optimal, as they all have different scenarios of application and the cost can vary. Alternatively, they are different trade-offs between exploration and exploitation.

Epsilon Greedy:

Epsilon Greedy is a simple method to find a balance between exploration and exploitation. With the probability of ϵ, the agent will choose a random action. Otherwise, the agent will choose the estimated optimal action. The ϵ can be tuned based on the desire for exploration or sometimes a decay value will be preferable.

$$\text{Action} = \begin{cases} \text{any action a}, & \epsilon \\ \max_a \ Q(s), & 1-\epsilon \end{cases}$$

Softmax:

In the Epsilon Greedy algorithm, when the agent chooses to explore, it will consider all the actions equally good. While in softmax, the agent is more likely to choose actions they think are better and less likely to choose actions they estimate are bad. There is a hyperparameter T called temperature, which controls the shape of the distribution. When T is close to 0, the agent will always choose the action with the highest Q value. On the other hand, when T is large, the distribution will become uniform. The probability can be calculated with the formula below.

$$\frac{\exp(Q(s,a)/T)}{\sum_{a'} \exp(Q(s,a')/T)}$$

Optimistically:

Optimistically is another strategy to choose actions, which will encourage the agent to explore the under-visited state. As the name suggests, when a state has been visited less than N_e times, the agent will optimistically estimate the state will lead to maximum reward R^+. After the state has been explored enough time, the agent will always choose the action with the highest Q value.

$$Q(s,a) \leftarrow Q(s,a) + \text{reward} + \gamma \max_a f(Q(s,a), N(s,a))$$

$$f(Q,N) = \begin{cases} R^+, & N < N_e \\ Q(s,a), & \text{otherwise} \end{cases}$$

Q-learning (Temporal Difference)

Q-learning is an off-policy algorithm for agents to learn how to maximize the cumulative reward while interacting with the environment(Watkins and Dayan 1992). Each state has a corresponding Q value when taking different actions, which represents the estimated value of an action in a specific state. The goal of Q-learning is to correctly or closely estimate the Q value of an action in every state. Q-learning is a model-free RL algorithm and there is no known transition probability and no known reward model. Q-learning uses the Temporal Differences evaluation to update the Q-value through exploration. Temporal Difference is an algorithm that can learn to predict a value that depends on the future estimated value. More specifically, when the agents are evaluating the value of the current state by taking a specific action, it needs to predict the next state and what the value of the next state will be. In Temporal Difference evaluation, the agents will choose a maximum Q value of the actions that can be taken in the next state as the estimated value of the next state. The Temporal Difference evaluation requires less trajectory, but the estimate is biased because we cannot guarantee the value of the next state is correctly estimated.

Algorithm 1 Q-learning

Initialize Q-values and count N
for i = 1, 2, ..., E **do**
  s ← env.reset()
  while not done **do**
    a ← select action(s)
    s', r, done ← env.step(a)
    $N(s,a) \leftarrow N(s,a) + 1$ ▷ Update Count
    $\alpha \leftarrow \frac{1}{N(s,a)}$ ▷ Learning rate
    $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma max_{a'} Q(s',a') - Q(s,a))$
    $s \leftarrow s'$
  end while
end for

GLIE (Monte Carlo Control) Essentially, GLIE is very similar to Q-learning. The difference between them is that they use different model free evaluations to update the value of each state. Instead of Temporal Difference evaluation, Monte-Carlo evaluation is used in GLIE. Monte Carlo Control does not suffer from biased estimation, as each update is based on the final reward or penalty. It will collect all the states, actions and rewards from a trajectory, then the Q values of the states and actions in the trajectory will be updated accordingly, depending on the reward. However, it also has the obvious disadvantage that it requires more trajectory to fully update the value of each state and action.

Algorithm 2 Greedy in the Limit with Infinite Exploration (GLIE)

Initialize Q-values and count N
for i = 1, 2, ..., E **do**
  s ← env.reset()
  episode ← []
  while not done **do**
    a ← select action(s)
    s', r, done ← env.step(a)
    episode.append((state, action, reward))
    $s \leftarrow s'$
  end while
  $G_k \leftarrow \sum_t \gamma^t r_t^{(k)}$
  for s, a, r in episode **do**
    $N(s,a) \leftarrow N(s,a) + 1$ ▷ Update Count
    $\alpha \leftarrow \frac{1}{N(s,a)}$ ▷ Learning rate
    $Q(s,a) \leftarrow Q(s,a) + \alpha(G_k - Q(s,a))$
  end for
end for

SARSA

SARSA can be considered an on-policy version of Q-learning. When the penalty is expensive, SARSA is a very popular method for estimating the value of each state(Miyazaki, Kojima, and Kobayashi 2007). For example, if we want a robot to explore an unknown environment. We want to explore the environment as much as possible and we do not want to lose the robots. As a result, SARSA is an RL algorithm that can minimize exploration losses, but this also means that it can only achieve sub-optimal cumulative rewards in comparison to Q-learning and GLIE in most cases. The techniques involved are simple. Instead of choosing the next action that can maximize the estimated value in the next state(Q-learning), it performs select action function again for the next state. In blackjack, reward and penalty are equally weighted. As a result, the purpose of implementing SARSA is to observe the performance of SARSA compared with other RL algorithms.

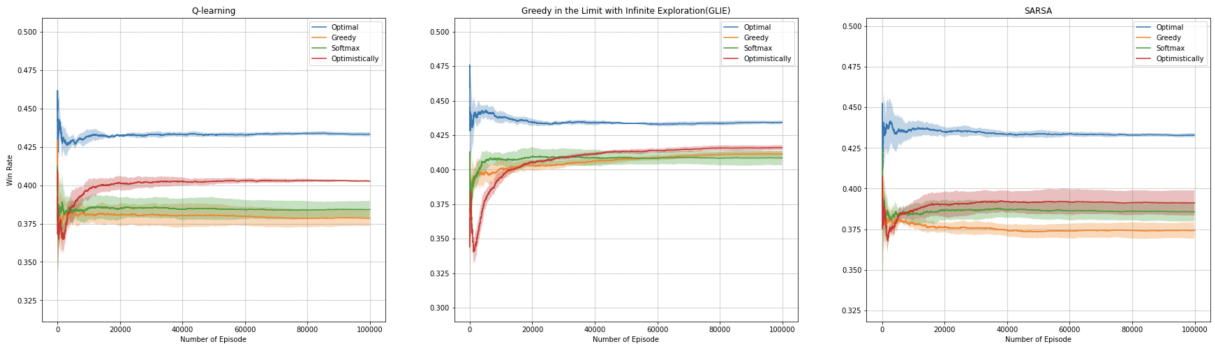

Figure 3: Win rate of RL algorithm for different strategies to choose action

Algorithm 3 SARSA

Initialize Q-values and count N
for i = 1, 2, ..., E **do**
  state ← env.reset()
  a ← select action(state)
  while not done **do**
    s', r, done ← env.step(a)
    $a' \leftarrow$ select action(s')
    $N(s,a) \leftarrow N(s,a)+1$ ▷ Update Count
    $\alpha \leftarrow \frac{1}{N(s,a)}$ ▷ Learning rate
    $Q(s,a) \leftarrow Q(s,a)+\alpha(r+\gamma Q(s',a')-Q(s,a))$
    $s \leftarrow s'$
    $a \leftarrow a'$
  end while
end for

Evaluation

Performance

	Mean	Std
Baseline	0.4326	0.0020
Q learning	0.4013	0.0005
GLIE	0.4176	0.0030
SARAS	0.3926	0.0077

Table 1: Average Win rate and standard deviation of different RL algorithms through 10 Epochs

The baseline is the highest win rate players can achieve by following the strategy in Figure 1. As can be seen from Table 1, all the RL algorithms achieve a decent win rate compared with Baseline. This proves that our algorithms is effective in dealing with the blackjack problem. Our subsequent discussion of all the algorithms is based on their ability to achieve a decent result. Instead of comparing performance among three algorithms that fail to perform as expected.

Select Action

As can be seen in Figure 3, regardless of which RL algorithm we use, agents can always achieve the highest win rate by using optimistic utility estimates to choose an action, which is expected. Because there is so much randomness in blackjack, sometimes even if a player makes a suboptimal action, the player still has a chance to win. For example, when a player gets a 20, the optimal strategy should be stick, in this case, the player is likely to get closer to a 21 than the dealer. However, there is also a very small chance that the player will get an Ace if they choose to take another card. These randomnesses can easily mislead agents. Therefore it is difficult for agents to correctly estimate the value of each state through a small number of experiences. Thus, optimistic utility estimates guarantee enough exploration achieves the optimal results. It is worth noting that the optimistic utility estimates are the ones that take the most iteration to converge, which is the cost of guaranteed exploration. In blackjack, this is not a high cost, because there are only 704 states in blackjack and each trajectory is short.

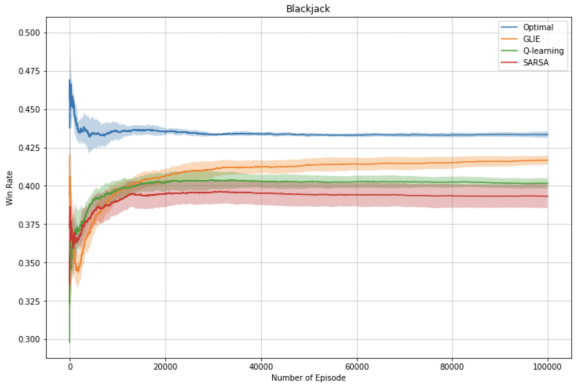

Figure 4: Win rate of different RL algorithm

RL algorithms

As can be seen in Figure 4, unlike most environment, GLIE (Monte Carlo Control) performs better than Q-learning(Temporal Difference), which approve our assumption. This is because each trajectory is very short. As a result, updating at the end became less of a disadvantage for Monte Carlo Control. Meanwhile, the time spent per trajectory is very short, taking only 20 seconds to get 100,000 trajectories without GPU. As a result, the performance of Monte Carlo

Control is better than Temporal Difference in blackjack. It is worth noting that Monte Carlo Control still requires a maximum number of trajectories to converge.

In addition, SARSA(0.3926) has a slightly worse win rate compared with Q-learning(0.4013), which is expected. As SARSA is more sensitive to penalties, exploring can become less appealing. To my surprise, the variance of SARSA(0.0077) is much larger than that of Q-learning(0.0005). This could be due to the small sample size(10).

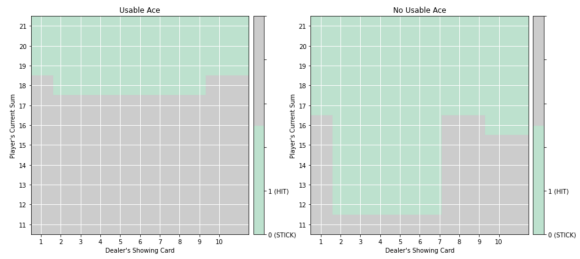

Figure 5: Policy generated from the GLIE algorithm, the left figure represents the player has a usable Ace(can be counted as 1 or 10), while the right figure represents the player does not have a usable Ace(No Ace or Ace can only be counted as 1)

Policy

As can be seen from Figure 5, the policy we generated from the GLIE algorithm is very similar to the policy in Figure 1. The policy was derived by choosing the maximum Q-value from two actions. It also proves that our agents can find the near theoretically optimal policy by constantly interacting with the environment.

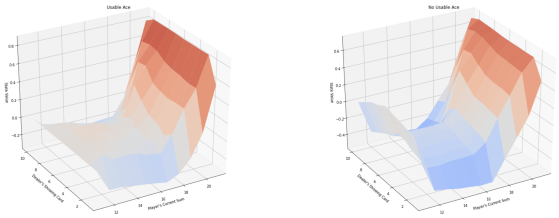

Figure 6: Estimated value of each state

Estimated Value

Figure 6 displays an estimated value of each state. If the estimated value is higher, then players is more likely to win if they reach the state, otherwise, more likely to lose.

Conclusion

Overall, the performance of the RL algorithm in blackjack is pretty good. It can achieve a 41.8% average win rate by using Monte Carlo Control in GLIE while the average win rate of the optimal strategy is 43.2%. Also, we have proven that Monte Carlo Control performs better than Temporal Difference Evaluation in blackjack. It is because each trajectory in blackjack is short and more sensitive to bias estimation. But Monte Carlo Control will still take more iterations to converge. Moreover, it seems that optimistic utility estimates have better performance than epsilon-greedy and softmax in each RL algorithm in blackjack, but it also takes more iterations to converge.

Future improvement

We can develop a classification Neural Network to blackjack to update the Q value, then we can implement reinforce with baseline and PPO algorithm.

References

Miyazaki, K.; Kojima, T.; and Kobayashi, H. 2007. Proposal and evaluation of the penalty avoiding rational policy making algorithm with penalty level. In *SICE Annual Conference 2007*, 2766–2773. IEEE.

Perez-Uribe, A., and Sanchez, E. 1998. Blackjack as a test bed for learning strategies in neural networks. In *1998 IEEE International Joint Conference on Neural Networks Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98CH36227)*, volume 3, 2022–2027 vol.3.

Silver, D.; Sutton, R. S.; and Müller, M. 2012. Temporal-difference search in computer go. *Machine learning* 87(2):183–219.

Sutton, R. S., and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press.

Watkins, C. J., and Dayan, P. 1992. Q-learning. *Machine learning* 8(3):279–292.