Solving Blackjack with Q-learning and DQN

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

Blackjack is a popular casino game, of which the objective is to get a better hand than the dealer without busting. The goal of this paper is to investigate and compare two different Reinforcement learning techniques for approximating the optimal strategy. Determining this strategy isn't trivial, due to Blackjacks' stochastic nature. Reinforcement learning approaches has proven to be very effective in stochastic environments, and especially in Blackjack, as they are able to exploit the games' inherent reward structure. This paper is written as part of the final project in the course "Autonomous and Adaptive Systems" and will explore using Q-learning and Deep Q-learning for approximating the optimal strategy.

1 Introduction

1.1 Blackjack

Blackjack is a game played with a dealer and a player, where the objective for the player is to beat the dealers total without exceeding a total of 21. The hand of the dealer is partially hidden throughout the game, until the player stands. Initially, the player and the dealer are dealt two cards each, with the dealers last card facing down. An ace can either be worth 1 or 11 points. K, Q and J counts for 10 points each, while the other cards are worth their face value. The player has to choose the best action to take, taking into consideration their hand, the dealers up-card and of course, the actions available.

In this project, a simpler version of Blackjack has been implemented, where splitting is not possible. The possible actions are hit, stand, double down and surrender. Hitting results in receiving another card from the dealer, and this action can be taken as many times as the player wants, as long as their total is less than 21. When the player doesn't want to receive additional cards, the player stand. Upon standing, the dealer has to show their hidden card, and while the dealers total is less than 17, the dealer has to draw additional cards. Doubling down is only possible to after receiving the initial two cards. The player then receives one additional card, but now the stakes are doubled. The player can surrender any time, which will result in the player getting back half of the original bet.

1.2 Q-learning and Deep Q-learning

Reinforcement learning is a good option when the problem can be modelled as a Final Markov Decision Process (FMDP), where in this case the transition probabilities are unknown. Q-learning is a model-free approach that finds an optimal policy maximizing the cumulative reward over all successive steps, starting from the current one. Q-learning can find the optimal policy of any FMDP (1), given infinite exploration time and a partly random policy. It also requires a table-representation of all the possible states and actions, making it a sub-optimal strategy for problems with large state and action spaces. Deep Q-learning uses a neural network to approximate the Q-value function. It takes the current state as input, and outputs an estimate the Q-values.

2 Method

2.1 Implementation details

Actions that lead to win, loss or draw are given a reward of 1, -1 and 0 respectively. Actions that do not lead to the game terminating are rewarded with 0. If the agent doubled down, then the reward is multiplied by two. Surrendering always results in a reward of -0.5, while natural Blackjack always is rewarded with 1.5.

One of the challenges I encountered was determining the observation space of the agent. Some strategies in blackjack involve counting the cards in the deck. I ended up implementing the game with 4 decks of card, only refilling it when it runs out. However, my implementation does not allow for predictions to be made based on the deck, and thus a static strategy was used. Because Q-learning implementation is simpler and faster with a smaller state space, I ended up keeping it simple. The state therefore consists of the players hand total, the dealers visible card and whether the hand is soft or not.

Illegal actions are impossible for the agent to choose. I was considering penalizing illegal actions with a large negative reward, but I figured this could make learning even harder for the agent. I instead implemented the game in such a way that before an action is chosen, either randomly or greedily, it checks if it's legal first. If it's not, it'll either pick randomly or greedy among the legal actions.

2.2 Exploration

Both of the algorithms are using an ϵ-greedy policy, with exploring starts. To begin with, the agent is following a completely random policy, and then ϵ decreases linearly until it is 0.1. Having a minimum exploration rate of 0.1 ensures that the model explores surrendering and doubling more, as just choosing between only hitting and standing typically will result in positive rewards when done at the right time.

2.3 Q-learning

The Q-learning training loop was implemented as shown in Figure 1. The training was done over 6 million episodes, and the testing over 1 million episodes. I started out with a γ of 0.99 and α of 0.01. After trying out different discount factors, the agent obtained the best results with a gamma of 0.85.

```
For each episode:
    Choose action a from policy derived from Q(s,a)
    Perform a, observe s' and r
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)]$
    s $\leftarrow s'$
    Until game over
```

Figure 1: The Q-learning algorithm

2.4 DQN

The DQN training loop was implemented similarly to the one for Q-learning, the difference being training the network instead of updating the Q-value function. A replay experience buffer was used for preventing the network of overfitting to recent experiences. Each new experience is put in the buffer, and when it's full, old experiences are forgotten. I used a buffer size of 10000, with a batch size of 256. I also tried a buffer size of 1000 with 32 as batch size, but it seemed to make little difference. Using a buffer size that is too big can result in the agent overfitting on early experiences.

The network was implemented using Keras and consists of 6 layers. I used mean squared error as the loss function, and gradient descent to update the network. γ remained the same as with Q-learning, but the learning rate for the network was set to 0.001 using the Adam optimizer. The network was trained over 500 000 episodes.

3 Results

The dealer will always have an edge in Blackjack, no matter how perfect the strategy used is. One of the advantages of the dealer is that the player starts and can bust before the dealer can bust. In order to determine the performance of the agent in my version of Blackjack, I have implemented an agent that plays the optimal strategy and an agent that plays a random strategy.

3.1 Q-learning

The final policy obtained by the Q-learning agent can be seen in Figure 2. The obtained policy is a bit different from the optimal one, however, it is clear that the agent is learning, as it is hitting on all the low hands and standing on all the high ones. These are easy for the agent to learn, as choosing these actions typically always gives good results. It's in the values in the middle it gets interesting, as there is more "gambling" involved. The deviation from the optimal policy that we see here could be solved by adding more exploration during the training session, not just at the beginning. This way, we could be sure to explore more for example doubling, instead of hitting.

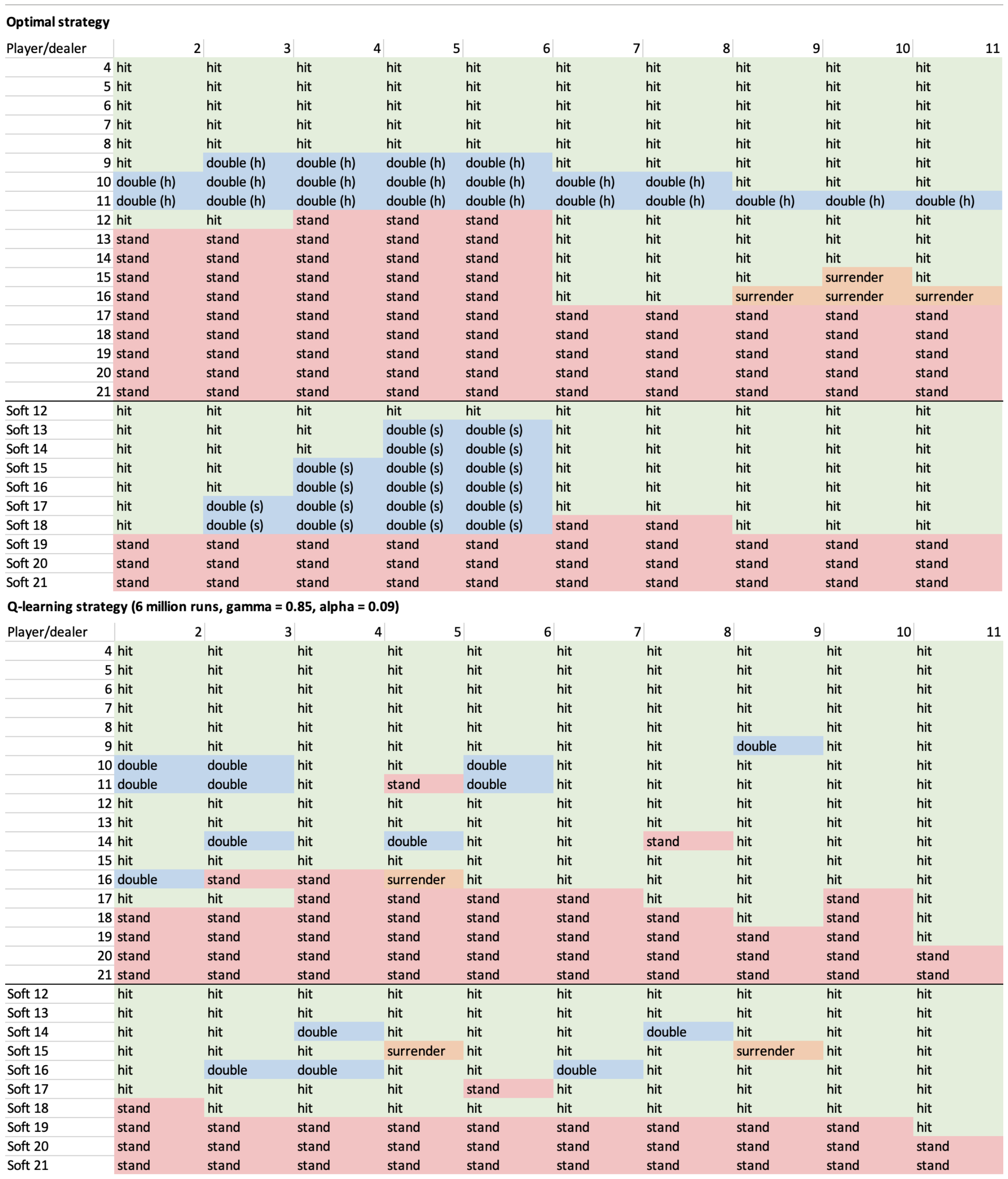
The final result of the agent, compared to the optimal and random agent, is shown in table below. The Q-learning agent, though not optimal, still outperformed the random agent by almost 17%, and wins about 4% less than the optimal agent.

	Q-learning	Optimal	Random
Win	37.7%	41.9%	21.1%
Draw	7.1%	7.9%	3.9%
Average reward	-0.1585	-0.1995	-0.4407

3.2 DQN

The DQN I implemented didn't converge to anything meaningful. Almost no matter the input, the model outputted the same Q-values. I tried adding more layers to the model. At one point, the model was training with as many as 11 layers. Obviously this requires more training, as it adds complexity to the model, but neither this proved useful. I also tried with different activation functions for the layers, different loss functions, and playing with the hyper-parameters. When I changed the input from being the raw data from the state to being normalized data, the results improved remarkably little.

One possible reason for the failure of this network could be some error in the implementation of the environment. I do however find this slightly unlikely as the same environment and reward structure seemed to work for the Q-learning algorithm. All of this leads me to believe that I need to train the network for way longer than what I did in order to see some convergence.



Optimal strategy

Player/dealer	2	3	4	5	6	7	8	9	10	11
4	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
5	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
6	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
7	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
8	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
9	hit	double (h)	double (h)	double (h)	double (h)	hit	hit	hit	hit	hit
10	double (h)	double (h)	double (h)	double (h)	double (h)	double (h)	double (h)	hit	hit	hit
11	double (h)	double (h)	double (h)	double (h)	double (h)	double (h)	double (h)	double (h)	double (h)	double (h)
12	hit	hit	stand	stand	stand	hit	hit	hit	hit	hit
13	stand	stand	stand	stand	stand	hit	hit	hit	hit	hit
14	stand	stand	stand	stand	stand	hit	hit	hit	hit	hit
15	stand	stand	stand	stand	stand	hit	hit	hit	surrender	hit
16	stand	stand	stand	stand	stand	hit	hit	surrender	surrender	surrender
17	stand	stand	stand	stand	stand	stand	stand	stand	stand	stand
18	stand	stand	stand	stand	stand	stand	stand	stand	stand	stand
19	stand	stand	stand	stand	stand	stand	stand	stand	stand	stand
20	stand	stand	stand	stand	stand	stand	stand	stand	stand	stand
21	stand	stand	stand	stand	stand	stand	stand	stand	stand	stand
Soft 12	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
Soft 13	hit	hit	hit	double (s)	double (s)	hit	hit	hit	hit	hit
Soft 14	hit	hit	hit	double (s)	double (s)	hit	hit	hit	hit	hit
Soft 15	hit	hit	double (s)	double (s)	double (s)	hit	hit	hit	hit	hit
Soft 16	hit	hit	double (s)	double (s)	double (s)	hit	hit	hit	hit	hit
Soft 17	hit	double (s)	double (s)	double (s)	double (s)	hit	hit	hit	hit	hit
Soft 18	hit	double (s)	double (s)	double (s)	double (s)	stand	stand	hit	hit	hit
Soft 19	stand	stand	stand	stand	stand	stand	stand	stand	stand	stand
Soft 20	stand	stand	stand	stand	stand	stand	stand	stand	stand	stand
Soft 21	stand	stand	stand	stand	stand	stand	stand	stand	stand	stand

Q-learning strategy (6 million runs, gamma = 0.85, alpha = 0.09)

Player/dealer	2	3	4	5	6	7	8	9	10	11
4	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
5	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
6	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
7	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
8	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
9	hit	hit	hit	hit	hit	hit	hit	double	hit	hit
10	double	double	hit	hit	double	hit	hit	hit	hit	hit
11	double	double	hit	stand	double	hit	hit	hit	hit	hit
12	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
13	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
14	hit	double	hit	double	hit	hit	stand	hit	hit	hit
15	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
16	double	stand	stand	surrender	hit	hit	hit	hit	hit	hit
17	hit	hit	stand	stand	stand	stand	hit	hit	stand	hit
18	stand	stand	stand	stand	stand	stand	stand	hit	stand	hit
19	stand	stand	stand	stand	stand	stand	stand	stand	stand	hit
20	stand	stand	stand	stand	stand	stand	stand	stand	stand	stand
21	stand	stand	stand	stand	stand	stand	stand	stand	stand	stand
Soft 12	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
Soft 13	hit	hit	hit	hit	hit	hit	hit	hit	hit	hit
Soft 14	hit	hit	double	hit	hit	hit	double	hit	hit	hit
Soft 15	hit	hit	hit	surrender	hit	hit	hit	surrender	hit	hit
Soft 16	hit	double	double	hit	hit	double	hit	hit	hit	hit
Soft 17	hit	hit	hit	hit	stand	hit	hit	hit	hit	hit
Soft 18	stand	hit	hit	hit	hit	hit	hit	hit	hit	hit
Soft 19	stand	stand	stand	stand	stand	stand	stand	stand	stand	hit
Soft 20	stand	stand	stand	stand	stand	stand	stand	stand	stand	stand
Soft 21	stand	stand	stand	stand	stand	stand	stand	stand	stand	stand

Figure 2: The optimal (top) and obtained (bottom) strategies.

4 Conclusion

Although the Q-agent was able to learn a fair policy for Blackjack, it was unable to learn the optimal one. I found that a little surprising, as blackjack has a fairly simple action and state space. This could be because of some bug in the environment, or just because it didn't train for enough episodes. The DQN didn't converge at all. This is either due to some mistake made in the implementation, the hyperparameters not being perfect, or that is needs to be trained for far more episodes than it did. Generally, alternative state spaces could be used, like for example allowing the agent to see the final hand of the dealer at the end of the game, like one can in real Blackjack.

References

[1] Melo, F.S, *Convergence of Q-learning: a simple proof*, available at: http://users.isr.ist.utl.pt/ mtjs-paan/readingGroup/ProofQlearning.pdf, accessed: 1.06.2022