An Implementation of ReQueST Modelling for Blackjack

Rajat Sethi, *Clemson University* Dr. Feng Luo, *Clemson University*

Abstract

The ReQueST[2] model is a specially designed reinforcement learning (RL) system that prioritizes reward-ambiguous environments. ReQueST is the product of three different innovations; reward modelling, trajectory optimization, and expert policies. By utilizing these predictive measures in certain environments, ReQueST allows the user to push their own values onto the agent, in the case that the agent finds itself in an ethical gray-area like the MIT moral machine[1]. In addition, this model "predicts" future observations before actually attempting them, which allows the agent to avoid "unsafe" and "dangerous" states. While this type of design was originally meant for autonomous vehicles and navigation problems, I have repurposed the model to play a famous imperfect-information game, Blackjack.

1. Methodology

1.1. The ReQueST Model

ReQueST is a portmanteau of "Reward Query Synthesis" and "Trajectory Optimization," each defining one of the major contributions of the model. Firstly, is the "Reward Query," which defines a user-trained reward model instead of a reward function. In a normal RL algorithm, there is an obvious reward (or punishment) if the agent engages in certain behaviors. For example, in chess, the agent would gain points for a win, lose points for a loss, and no change for a stalemate[4]. However, this model does not use a standard function, but rather a neural network(NN), as the reward system. To train this NN, the agent works in several environments and the user provides feedback (approval or disapproval). After enough epochs, the reward model is rigorously trained so that the agent can learn and utilize the user's morality code.

Along with the reward model, ReQueST uses a process known as trajectory optimization to simulate its possible actions based on its current "observations." In this system, an observation is the agent's current status while the actions represent the possible changes to those observations. For example, in a navigation task, the observation is the Using the cartesian product of all possible observations and actions, ReQueST can hypothesize what the next observation is going to be, then determine if that state is dangerous. Using this forecast, the agent can then avoid an unsafe state that may conflict with the aforementioned reward model.

1.2.1. Code Abstraction – Blackjack - Betting

Blackjack is a gambling-based card game normally played at casinos or gambling facilities. In this game, there are two parties, the dealer and the players, competing in an indefinite number of rounds until the player is content with their winnings or loses everything. It is implied that the dealer has a significant amount of money at their disposal, as they work for the casino itself. The player on the other hand starts off with whatever money they wish to put in, which should always be significantly less than the dealer's funds. In this code abstraction, the agent acts as the player with a limited fund, an integer value initialized at around 1000. This number can be changed at will for testing purposes.

The dealer on the other hand is a completely automated system (explained in more detail in section 1.2.4). The dealer in this abstraction has no "intelligence," it is simply following a set of rules designated by the game. Since this is a program, the dealer does not have a variable designated for its own funds, but rather has full power to add/subtract an unlimited number of points from the agent.

Prior to the round's start, the player/agent must put down a non-zero bet. The agent has no knowledge of its own cards or the dealer's. Since the agent does not know what the environment will be like, it cannot deviate from its previous bets or strategize beforehand. Only when the first cards are dealt can the agent make a proper decision.

1.2.2. Code Abstraction – Blackjack – Cards and Hands

After the bet is placed, the dealer gives the player two cards, face-up. Then, the dealer gives themselves two cards, one face-up and the other face-down. Under normal circumstances, if the dealer's face-up card is an "Ace", then the player is given something called an "insurance" option. However, insurance has been statistically proven to be a bad move for the player, so it was not incorporated in this system[3].

In this game, each card is given a point value based on their number. Their suit is irrelevant to this game. A hand's point total equals the sum of its cards' point values. Cards 2-10 have the same point value as their number (i.e., a 5 card is worth 5 points). Jacks, Queens, and Kings are worth 10 points. The Ace is a special card that normally counts for 1 point. However, if the current point total is ≤ 11, then the Ace automatically can add 10 points to its own value to be worth 11 points (i.e., a 9-point hand with an ace is also worth 19 points). If the hand exceeds 21 points, but contains

an 11-point ace, then the ace can reduce its value back to 1 point.

In the code abstraction, the player's hand is represented with a vector. Each card that is added (explained in 1.2.3) is pushed to the end of this vector. Once a card is added, the program recalculates its total value. If an ace card needs to be increased/decreased, the system automatically changes the value of the first ace in the vector. In addition, a Boolean value named "Ace11" is flipped on/off if an 11-point ace exists in the hand, which will be included in the agent's observations.

1.2.3. Code Abstraction – Blackjack – Agent's Turn

As mentioned earlier, the game begins when the dealer issues two cards to each player after the bet. Only one of the dealer's cards are revealed to the agent. The goal of this game is to have the largest hand point value without going over 21 points. As a special rule, if the player gets 21 points in the original drawing and the dealer does not, then the player instantly wins 1.5x their original bet (i.e., if they bet 100, they gain 150). In the first turn, the player has three possible moves; hit, stand, or double. If the player "hits", then they are given another card from the deck. If the player "stands", then they freeze their hand and they can no longer make any moves. In this program, once the player stands, it instantly becomes the dealer's turn.

If the player "doubles", then three events happen. First, the original bet is doubled (assuming the player has enough funds). Next, the player hits and adds a card to their hand. Lastly, the player stands, which means they are stuck with the doubled bet and a three-card hand. In most blackjack games, the player might have the option to "split" their hand if the two original cards are the same. In this scenario, the two cards are split into two separate hands and one card is added to each. The player then adds the same bet to the second hand. Since this system only works for one agent, splitting was not incorporated into the model as of this point.

In the subsequent turns, the player loses the option to double or split, and they can only hit or stand. The system is designed so that if it tries to double after the first turn, the dealer will treat it as a hit and the bet is unchanged. To account for the difference between the first and later turns, a variable is added to the class that keeps track of the turn number.

The player continues making hits until one of three events happen. If the player stands, their turn ends, and the dealer makes their move. If the player exceeds 21 points in their hand (not including an 11-point ace), then the player instantly loses their bet without the dealer making a move. This is known as "busting." Rarely, it is possible for the player to make three hits in a row and have 5 cards in their hand. If this happens without busting, then its known as a "Charlie" and the player automatically wins.

1.2.4. Code Abstraction – Blackjack – Dealer's Turn

If the player stands, then the dealer performs an automated set of moves. While the dealer's hand is less than 17 points, continue hitting. Once the dealer hits 17 points, they must stand and determine a winner. As an extra rule, dealers can declare to hit or stand on a "soft 17" before the game starts. If the dealer has 17 points, but also an 11-point ace, then it's considered a soft 17. Dealers must disclose what they do in this scenario. For this study, both possibilities are studied. Standing on a soft 17 is the default value in the main environment, while hitting is incorporated into the "transfer" environment.

Once the dealer exceeds the 17-point mark, the round winner is determined. If the dealer busts or gets a lower value than the player, then the player wins their bet (i.e., if they bet 100, then they gain an extra 100). If the dealer gets a higher point value than the player without busting, then the player loses their bet. If the dealer and player tie, then it's a "push" and the player reclaims their original bet without gaining/losing anything.

1.3. Important Functions

To simulate a blackjack game properly, I developed three important, major functions to work in tandem with the ReQueST model; reward_func, make_expert_policy, step. Each of these functions were designated in the class interface as they are directly called by the ReQueST model.

The reward_func method normally determines the "reward" after committing a series of actions. However, in this case, the action has not happened yet. This function takes the hypothetical observations made from possible actions and determines the reward values of those hypotheticals. In the case of blackjack however, there is no reward until the round is over, so it has been modified to include the probability of "busting" and losing the best.

The expert policy serves as a kind of tutor for the reward model, giving guidelines on what actions it should take in certain situations. For blackjack, I followed three simple rules.

- Agent must stand when it hits 17 points
- Agent must double/hit when it hits 10/11 points
- Agent must hit if below 10 points

Anything else was chosen randomly by the agent to promote its learning.

Finally, was the step function. For the other examples, the step function was for 2D navigation and would move the agent at a certain angle and speed. However, neither of

those objects exist in blackjack. Instead, the step function represents one turn in the round. It would first look at the action of the player's last turn. If it was a hit or double, it would do so accordingly and determine whether the player was still in the game or if they had busted. If the player stood instead, it would simulate the dealer's line of play and determine whether the player beat the dealer, pushed, or if they lost their bet. Once the "round" was over, it would tell the system that it was "done" and provide the results (in terms of how much the player gained or lost).

2. Results

Unfortunately, the reward model outputs were designed in a way to only support 2D navigation, which means that most of the graphs were not very helpful. In phase 3, I plan to "improve" the ReQueST Model by providing support for non-2D navigation related problems.

Another issue with the results was the reliance on graphs and not numerical attributes. ReQueST was designed solely for 2D navigation, and as such relied on 2D graphs like matplotlib to show results. Because of this, many of my multiple-dimension inputs were not compatible with the outputs. In phase 3, I also hope to fix this model and allow for 3+ dimensional inputs.

However, I was able to salvage a few of loss function related statistics. As shown in the following table, the loss function continuously goes down until the loss asymptotes near 0. This suggests either an outstanding gradient descent and improvement against the dealer, or an error on my end.

```
0 10000 1.097754 1.047536
100 10000 0.002359 0.002186
200 10000 0.000642 0.000598
300 10000 0.000298 0.000274
400 10000 0.000165 0.000154
500 10000 0.000105 0.000097
600 10000 0.000070 0.000065
700 10000 0.000049 0.000045
800 10000 0.000036 0.000033
900 10000 0.000026 0.000024
1000 10000 0.000020 0.000018
```

3. Discussion

Before I discuss the models and the concepts themselves, I would like to speak a little about the code given by the ReQueST creators. When developing a high-level, complex, reinforcement learning algorithm, having some documentation would be extremely valuable. However, many aspects of the repository did not even have comments, let alone README files or Instructions. Even setting up the environment was quite the difficulty, as the team had to use a deprecated version of Tensorflow and imgresize. I would hope that in future studies involving ReQueST, the notebooks and the scripts at the very least need to have commenting explaining what the parameters are and what the methods do, as to promote replication and further improvements.

Due to this lack of documentation, I may have made a myriad of unseen errors in my own research. I have faithfully attempted to create the blackjack model by analyzing the other examples and their environments. However, several functions were custom-made with components buried in directories, so the accuracy may have been affected by this lack of customization.

4. References

"Moral Machine." *Moral Machine*, https://www.moralmachine.net/.

Reddy, S, Dragan, A.D., Levine, S, Legg, S, and Leike, J. *Learning Human Objectives by Evaluating Hypothetical Behavior*. Retrieved from https://par.nsf.gov/ biblio/10183732. *International Conference on Machine Learning (ICML)* .

Villano, Matt. "Blackjack Insurance: Odds Say It's Almost Always a Bad Bet." *SFGATE*, San Francisco Chronicle, 25 June 2014, https://www.sfgate.com/entertainment/gaming/article/Blackjack-insurance-Odds-say-it-s-almost-always-5579474.php.

Zhao, Michelle. "Understanding Reinforcement Learning through Multi-Armed Bandits." *Medium*, Towards Data Science, 5 Apr. 2020, https://towardsdatascience.com/understanding-reinforcement-learning-through-multi-armed-bandits-39095dee6846.

5. GitHub Repository

https://github.com/RajatSethi2001/CPSC_8420_ReQueST