Teaching Killer Using Reinforced Learning

ECE 196
Fall 2020

Seth Litman

Abstract
Killer, also known as Big 2 or 13, is a Vietnamese card game where the goal is to lose all your cards by beating the previous hand. Four players, starting with 13 cards each, take turns either playing successively higher cards or passing. Card games like this can be taught to a computer using Reinforced Learning, which involves assigning values to certain game-states and actions. This paper simplifies the game to a manageable number of states, introduces a custom environment to simulate the game, and teaches 4 independent agents to play amongst themselves.

Keywords
Reinforced Learning, Q-Learning, Card games, Killer

Introduction

Reinforced Learning (RL) is a form of machine learning where a computer learns to complete a task by receiving rewards from taking certain actions in certain states. It remembers these rewards in a Q-table, which is a matrix where each row represents a state and each column represents an action. The value at each index is how preferable that action is. For example, suppose you have an autonomous taxi that is supposed to pickup passengers at one location and drop them off at another location. To teach it this, you would have the taxi incur a negative reward for failing to complete its task, and a smaller negative reward for each block it drives past. This way, the taxi would learn to deliver passengers properly and would do so as quickly as possible. Behind the scenes, the taxi's Q-table would probably adjust so that in a state where it has a passenger who wants to go someplace west, the action ”go west” would have the highest value. (Kansal, Martin)

Killer is a Vietnamese card game usually played between 4 players with starting hands of 13 cards each (a deck split 4 ways without jokers). The fundamental principle of Killer is that one needs to get rid of all their cards to win. There are many house rules and different versions, but my programmed environment uses the following rules:

(i) Each card is valued according to it's number and suit. The number hierarchy goes 3, 4...9, 10, Jack, Queen, King, Ace, 2, with 2's being the most valuable cards.
(ii) There is a suit hierarchy which goes Spades, Clubs, Diamonds, Hearts, with spades being the lowest valued suit.
(iii) A player can only play a combination of cards where the highest card (”topcard”) is greater than the topcard of the last played combination.
(iv) A player can only play the same type of combination as the last played combination. The types of combinations are: single cards, pairs of same number, triples of same number, 3-long straights, 4-long straights, and 5-long straights. A straight is a string of cards with consecutive numbers.
(v) A player must either play a legal move as described above or pass. Then it's the next player's turn.
(vi) If after a player's move, all 3 others pass, then that player can play any combination type with any topcard value.
(vii) The player with the 3 of spades starts the game.

My motivation for constructing this environment and programming this AI is that I want to improve my Killer skills. One time, I lost a game and was forced to eat a raw clove of garlic. I do not want to eat a raw clove of garlic again. By thinking through the recreation of this game in Python, I was able to analyze how I choose moves and the features I consider. By letting agents play against each other, taking these features into account, I hoped to learn which decisions lead to greater wins.

Problem-Solving Process

Elements of Reinforced Learning

The first step for me was to do research on the general idea of RL. I learned about the Q-table, which is a matrix where each row is a state and each column is an action. I also learned about policies. Policies are baseline strategies that can help speed up learning. For example, randomness can be a policy; an agent will try random actions until it accomplishes a goal. Those actions will then be “learned” by receiving a higher value in the Q-table. However, this can be very slow. It is often quicker to give an agent a policy to reasonably limit its actions. In the taxi example, one may implement a policy that forces the taxi to perform the “pick up” action if it is empty and next to a passenger.

Implementing RL in Python

From there, I learned about the OpenAI Gym library and went through the classic taxi problem. OpenAI Gym also has a Blackjack environment, which I tested. The Blackjack environment is very simple; Blackjack can be simplified to around 50 possible states and only

2 actions. There was a big problem with using OpenAI Gym though, and that was the fact that it only supports one agent. Because I don't know three people who would be willing to play against my AI for hours and hours, I would need to find a multi-agent compatible library.

There are a few libraries like this, such as ma-gym and Pettingzoo. However, similar to OpenAI Gym, these are meant to be used for making agents and policies, not entirely new environments. As such, installing custom environments is difficult. In the end, I decided to write my own multi-agent Killer environment, using Gym's format as a general guide.

Representing Killer for RL

On the face of it, Killer is a very complex game. On your first turn, you have a random 13 card hand, a random rule to follow, and any number of topcards to beat. If you were to represent the state space of this situation, it would have a size of:

$$|S| = {}_{52}C_{13} * 7 * 52 = 2.3e14$$

Even if the action space was small, this value is way too big for two reasons. The first is that it would be impossible for the agents to visit all these states, let alone multiple times. Secondly, my PC doesn't have enough RAM to store a Q-table that large (yet).

I realized early on that I would need to drastically simplify my state space and action space to be able to realistically use this game in RL. I started by deciding what I did not need RL for. For one, I did not need to teach my agent how to play; I implemented functions like get-moves that would return only valid, legal moves given a player's hand, the topcard, and the rule. Suddenly, these three features could be excluded from the action and state space calculation without losing information.

Solution

Overview of My Environment

KillerEnv uses the following notation.

(i) Card numebrs are represented as integers 3–10 represent those cards, 11 represents a Jack, 12 a Queen, 13 a King, 14 an Ace, and 15 represents a 2.
(ii) Add a decimal to represent the suit. +.0 is spades, +.25 is clubs, +.5 is diamonds, and +.75 is hearts.
(iii) Rule 0 is singles, 1 is pairs, 2 is triples, 3 is 3-straights, 4 is 4-straights, 5 is 5-straights, and 6 is new rule.

You can use the render_game() function to see a text print-out of everyone's hands.

Simplifying Killer Game-States

Even with my environment nicely coded, the game-states were still too complex to use for RL. I imagined myself playing a round of Killer and went through my thought process. Then I played a few virtual games online. I realized that I was not exactly paying attention to the objective value of each of my cards but rather their relative value in my hand. This lead me to my new idea: give moves tiers.

Creating move tiers solved two problems. For one, it greatly reduced the state space. Rather than use around 650 billion possible hands, I could simply observe if each of 13 tiers was playable. I implemented the tiers by associating each move with it's topcard. The topcard's relative value in the starting hand would determine that move's tier. As such, there are 13 tiers, and $2^{13} = 8192$ possible hands throughout the game.

Secondly, it simplified the action space. Rather than thousands of possible, mostly illegal moves, I could represent actions as playing a certain tier of legal moves. I ended up with an action space of 14, which is maximum 13 possible tiers to play or pass.

I asked my friend about his thought process when he plays a game of Killer. He said he often considers how other players will react to his own move. This struck me as a important part of gameplay that I had not thought of. For example, if the next player has only one card remaining, an intelligent agent would play a higher tier rather than a lower one. This is so the next player has a less likely chance of being able to legally play their last card and win.

Finally I ended up with my current state space representation. There are around 107,000 states composed of a player's playable tiers (as a one-hot vector) and the next player's remaining amount of cards (maximum 13). For reference, the last state in the Q-table is the state of every player at the start of the game, and the 0th state is where a player cannot play any tier legally and the player next to them has only one card left.

Playing Games

My KillerEx file contains the code used to actually have 4 agents play against each other. Each agent has their own Q-table. The discount factor is 1, which means that every state-action pair a player takes during a game will receive the same reward (-1 for a loss or +1 for a win) at the end of that game.

Epsilon is the percentage chance to use the most valuable action in the Q-table for a state, as opposed to doing a random action. It is higher for states that the agent has seen more times. That way, the agent will only use its knowledge if it has enough experience to justify doing so. This can be considered our agents' policy.

The functions step_player and step_game will manage one player's turn and an entire game respectively. The record_wins function will adjust the Q-tables based on each state each player saw during a game.

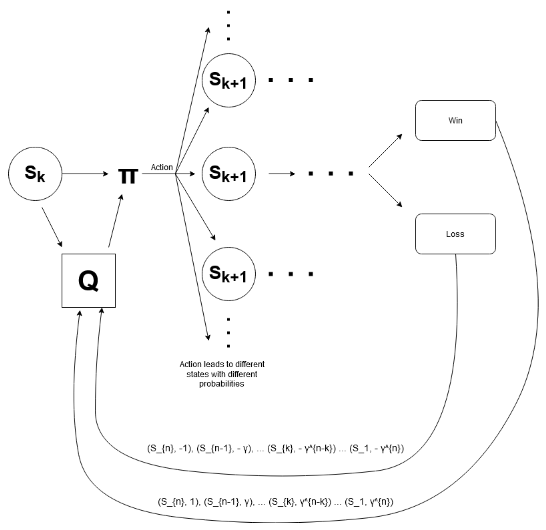
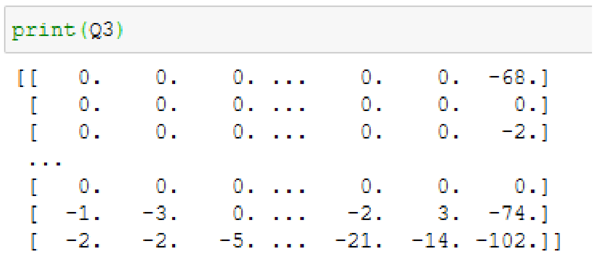

S_k
π
Action
S_{k+1}
S_{k+1}
S_{k+1}
Win
Loss
Q
Action leads to different states with different probabilities
(S_{n}, -1), (S_{n-1}, - γ), ... (S_{k}, - γ^{n-k}) ... (S_1, - γ^{n})
(S_{n}, 1), (S_{n-1}, γ), ... (S_{k}, γ^{n-k}) ... (S_1, γ^{n})

Figure 1. Algorithm Diagram

After running 500 games, the players seem evenly matched. They each have win rates of around 25%. However, their Q-tables are well-filled.



```
print(Q3)
```

```
[[   0.     0.     0. ...    0.     0.   -68.]
 [   0.     0.     0. ...    0.     0.     0.]
 [   0.     0.     0. ...    0.     0.    -2.]
 ...
 [   0.     0.     0. ...    0.     0.     0.]
 [  -1.    -3.     0. ...   -2.     3.   -74.]
 [  -2.    -2.    -5. ...  -21.   -14.  -102.]]
```

You'll notice that the last column of every Q-table is very low. That's because that is the pass action. The agents tend to pass a lot when choosing random actions because passing is always available (it has to redraw if it chooses a tier it cannot legally play). Although I could program that behavior out, it seems the agents are learning on their own that passing is not ideal in most circumstances.

Another interesting insight is that the beginning columns of the final state are higher than the following columns. This may mean that playing a very low tier move at the start of the game is more conducive to winning. This makes sense; if you are starting a game then you should use your free turn to get rid of as many low cards as possible.

To test my agent, I made new step functions that would have Player 4, my smart agent, play against three other players that play random moves. However, when I had my players start going, disaster struck! To my horror, P4, who should be at least a little smarter than the others, only won 18% of all games!

Future Work

I have some theories why this happened. Since the agents tend to pass a lot, they might develop a bias for passing in some states. Therefore, P4 passed way too often. I think in the future I will change the policy so that agents never pass when they have a free turn (the rule is 6). I should also experiment with the epsilon function so that agents use their knowledge more readily.

It may also simply be that my agents need a lot more training, maybe hundreds of thousands more games. It takes my computer about 5 minutes to run through 100 games. It would be interesting to see the results of running the simulation for a day or so.

On another note, I also worked on making a GUI interface for my Killer environment in tkinter. I think it would be cool to get it working with the simulations, with the end goal being to let a human play against one of my agents.

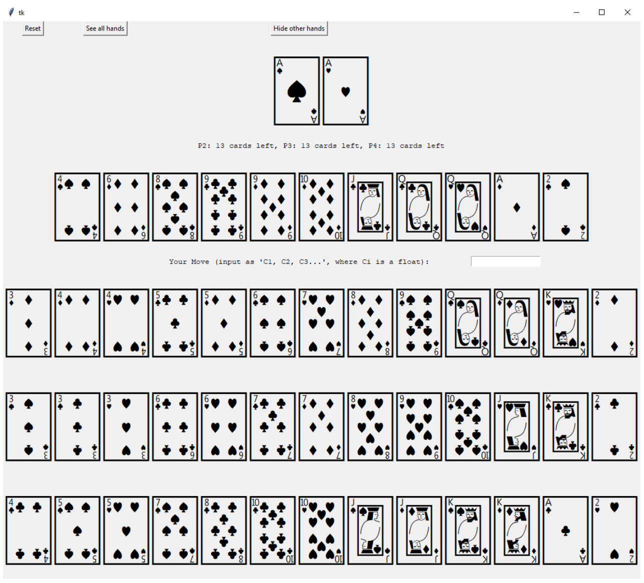
Although my RL project has not yet reached success, I have created a very strong and detailed foundation to build upon. Not only did I build a fully functioning custom environment for Killer, but I also coded the first draft of a reinforced learning scenario between four agents. Going forward, I will work out the bugs and finally unlock the Q-tables that will allow me to conquer this complex card game.

References

[1] Kansal, Satwik. Martin, Brendan. "Reinforcement Q-Learning from Scratch in Python with OpenAI Gym". *Learn Data Science.* Web. https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/

[2] Hamish. "Blackjack with Reinforcement Learning". *Kaggle.* Web. https://www.kaggle.com/hamishdickson/blackjack-with-reinforcement-learning

Supplemental Material



```
P2: 13 cards left, P3: 13 cards left, P4: 13 cards left
Your Move (input as 'C1, C2, C3...', where Ci is a float):
```

Figure 2. My Killer GUI made in Python with tkinter

My code can be found in this Github repository: https://github.com/salitman/Killer/