CS238: Reinforcement Learning for solving Coda Game

Yanlong Ma
yanlong@stanford.edu

Yu Zeng
zengyu@stanford.edu

Jiahao Zhang
jiahao21@stanford.edu

December 7, 2019

Abstract

This paper presents a reinforcement learning approach to the famous board game Coda (also called DaVinci Code). We modeled the game as a Markov Decision Process and attempted to solve it using methods learned in CS238. Due to the game's massive state-action space, we adopted global approximation and implemented both online and offline solution techniques to solve the problem. We tested the developed policies according to theirs win rates against an opponent doing random actions. The result indicates our best agent outperforms the naive benchmark by five percent while remains potential for further improvement.

1 Introduction

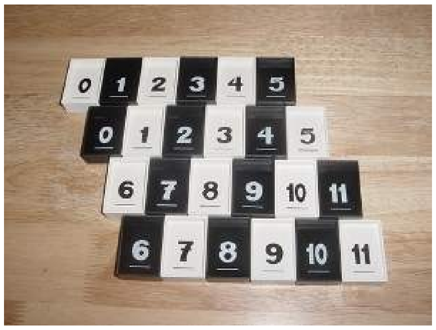

Figure 1: Coda (aka. DaVinci Code) Preview.

The Da Vinci Code game is a card game played by multiple players. The goal of the Da Vinci Code game is to reveal the player opponent's secret codes before they uncover the player. Each player begins by drawing a certain (depending on the number of players) numbered tiles from a pool. After each player lines up the tiles in numeric order so that only he or she can see them, play begins.
On the player turn, draw a new tile from the pool and set it to the side of your lineup. Choose a tile in the opponent's array, and then attempt to identify it (e.g. "This is a Black 1."). If the player is correct, the opponent reveals the tile. If the player is wrong, he/she must show the tile drawn from the pile and insert it into your lineup, in sequential order. As long as the player make correct guesses, the player can keep guessing additional tiles.

- Section II: reviews related work, including previous approaches for solving card games;
- Section III: define the game model, include state and action;
- Section IV: details our reinforcement learning approach;
- Section V: compare and analyze our experimental results;
- Section VI: make conclusions for our approach.

2 Related Work

Da Vinci Code game is a typical card game that has well definite rules. This kind of imperfect information card game with a relatively small state space and action space could be solved by reinforcement learning well. Examples range from simple Blackjack and UNO games to much more complicated games such as Mahjong and Go. In fact, there are many toolkits developed to solve card games. In RLCard [5], the authors build several environments for different card games with different algorithms. For the Da Vinci Code game, in particular, there are heuristic approaches [4] but no prior work has been found. We review several works done similar to this game. Deep Q-Learning as a relatively simple algorithm has been used widely, some other algorithms include Neural Fictitious Self-Play (NFSP) [3], Deep Counterfactual Regret Minimization (DeepCFR), [1], etc. In our project, rather than using a single algorithm, we try out using several basic algorithms to compare the performance with Deep Q-Learning and Double Deep Q-Learning.

3 Definition

The game can be represented as a typical reinforcement learning problem which consists of states, actions, transitions, and rewards. In addition, in order to simplify the problem, we make three assumptions below:

- The game ignores the 2 Jokers in the normal game.
- The number of players will be 2.
- The player has to continue guessing if he/she makes a correct guess.

3.1 State

In a game with a total of n cards, We define the state space S as a vector with $3n$ components, which can either be 0 or 1. As shown in the Figure 2, the first $2n$ components of the vector are used to denotes whether each card from the pool is displayed or hidden by the agent and the last n components represent the cards that have been displayed by the opponent. The state vector can be seen as the representation of information a human player can gather during playing the game.

3.2 Action

Since in the worst case one player can hold n cards and for each card there are n possible values, we can define the action space as a n^2-vector. Action $kn + j$, where

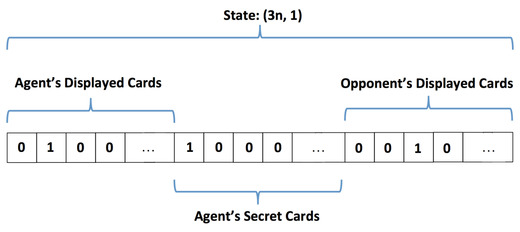

Figure 2: Vector representation of the state space.

$k \in [0, n-1], j \in [0, n-1]$, denotes the guess of the k^{th} card being card value j. The components, which we would later obtain, are the Q values for the actions.

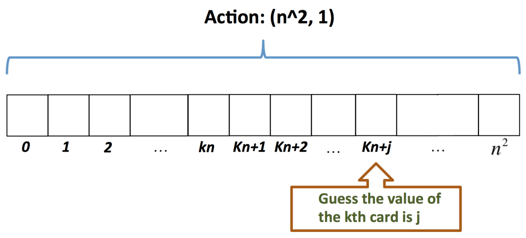

Figure 3: Vector representation of the action space.

3.3 Transition

Since the state vector consists of binary numbers, the transition is simple. If the agent gets a new card or has to show a card, then change the number 0 at the corresponding position to number 1.

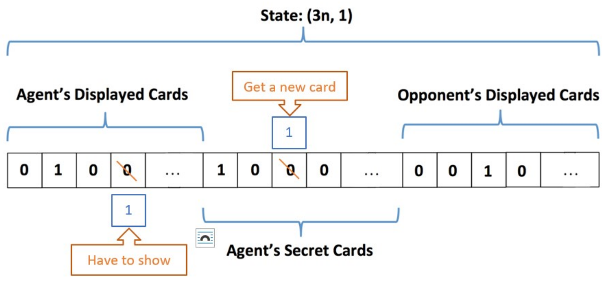

Figure 4: Vector representation of the transition.

3.4 Reward

There are two types of rewards:

- Step reward: the reward after each guess. If correct receive 10 points else reduce 10 points;
- Final reward: the reward after each game. If won receive 10000 points else receive -10000 points.

In order to win the game in fewer steps, a discount factor γ will be applied for each step.

3.5 Addition

Note, the game will end in three conditions:

- Agent wins: the opponent has no hidden card;
- Opponent wins: the agent has no hidden card;
- The last player wins: the pool is empty; the player draws the last card wins since he can infer all the cards.

4 Methods

In this project, we build 5 policy for the agent and the opponent. The performance is estimated by the winning rate of a certain policy to the basic random policy. ....

4.1 Base - Random Policy

This policy just randomly chooses a random feasible action (position, guessing). The random policy is only used by the opponent to estimate the performance of other policies. By calculating the winning rate of the agent, we can estimate the performance of a certain policy.

4.2 Benchmark - Heuristic Policy

The heuristic policy is an approach to the problem that employs a human player's practical method. The approach is concluded as the following 6 steps [4]:

- Filter out the game's secret which finds all the possible number of the opponent's hidden card;
- Keep the same color with the chosen card. Since we know the color of the chosen card, we can get rid of the possible numbers with a different color;
- Find the lower and upper bounds based on the card position. Since the i-th card has rank $>$ I and last card have rank $<$ n – I, we can get rid of the impossible potential numbers.
- Get the lower and upper bounds based on the nearest shown card. Since the cards are ranked, we can refine the potential solutions.
- Exclude the rank from the same color to refine the possible numbers.
- Calculate the probability of each potential number and return the largest one.

The heuristic policy does not base on any learning algorithm, but due to the complexity of the game, it works well in general. We use heuristic policy as a benchmark to check how well the other policies perform.

4.3 Q-Learning and Sarsa

The size of state space is 2^{3n} and the size of the action space is n^2. Therefore, the problem's complexity makes it hard to use Mode-Based learning algorithms (e.g. Maximum Likelihood Model-Based Methods). Instead, using model-free learning which do not need the transition and reward matrices is a better choice. In fact, all the principal approaches of this paper are model-free reinforcement learning. Q-learning is the core approach of all the model-free reinforcement used in this paper.
Although the ideal size of the state space is large, a fair proportion of states will never be shown up. For example, $[1, \ldots, 1; 1, \ldots, 1, \ldots]$ is an impossible state because a card cannot be both shown and hidden. Thus, the Q-Learning and Sarsa algorithms are not too memory intensive. The maximum epochs used in this paper is 10000.

4.4 Deep Q-Learning

As the state size and action size become larger and larger, it is hard to infer the Q value from the states we already explored. The amount of both memory and time required boom. By using the neural network to approximate the

Q value, we modify the original Q-Learning algorithm to Deep Q-Learning algorithms.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

The key steps of Deep Q-Learning summarize in the following:

- Store all the past explore;
- Choose the action by maximizing the output of the Q-network;
- Calculate the loss function which is the squared error of the predicted Q value and the target Q value.

$$\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_\theta \mathbb{E}_{s' \sim P(s'|s,a)} \left[\left(Q_\theta(s,a) - \text{target}(s') \right)^2 \right] \Big|_{\theta=\theta_k}$$

where target $= r + \gamma \max_{a'} Q(s', a'; \theta)$

The figure below shows the difference between Q-Learning and Deep Q-Learning.

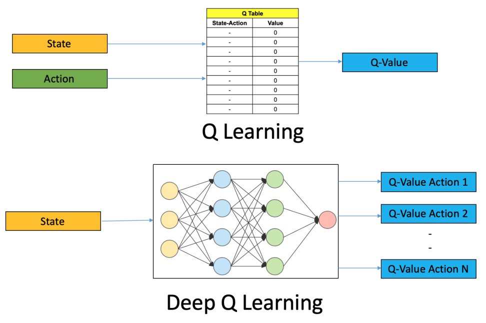

Figure 5: Q-Learning v.s. Deep Q-Learning.

4.5 Double DQN

Double Q-learning is an off-policy reinforcement learning algorithm, where a different policy is used for value evaluation than what is used to select the next action. In practice, two separate value functions are trained in a mutually symmetric fashion using separate experiences, Q^A and Q^B [2]. The double Q-learning update step is then as follows:

Define $a^* = \arg\max_a Q^A(s', a)$
$$Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a) \left(r + \gamma Q^B(s', a^*) - Q^A(s,a) \right)$$

Define $b^* = \arg\max_a Q^B(s', a)$
$$Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a) \left(r + \gamma Q^A(s', b^*) - Q^B(s,a) \right)$$

Similar to Deep Q-Learning, we use the neural network to approximate the Q values and obtain the Double Deep Q-Learning algorithm.

5 Results

The parameters used for each approach is summarized as the table below:

Parameters			
Approach	Number of Cards (n)	Initial Cards ($nStart$)*	Epoch
Heuristic Policy	24	1	N.A.
Q-Learning	24	1	10000
Sarsa	24	1	10000
Deep Q-Learning	24	1	2000**
Double Deep Q-Learning	24	1	2000**

* initial number of cards for each player.
** due to the complexity of the neural network computing, we limit the Epoch for Deep Q-Learning and Double Deep Q-Learning to 2000.

The performance of each approach is summarized as the table below:

Performance		
Approach	Winning Rate (n)	Running Time (sec)
Heuristic Policy	0.839	6.43
Q-Learning	0.963	11.58
Sarsa	0.965	13.73
Deep Q-Learning	0.895	18 / 32*
Double Deep Q-Learning	0.915	25 / 128*

* batch size = 32 / batch size = 128

The Heuristic Policy approach does not need to train the model. We ask the agent played 10000 games with the random opponent directly. For both Q-Learning and Sarsa approaches, the training phase consists of 10000 games versus a random opponent. For Deep Q-Learning and Double Deep Q-Learning, the training phase consists of 2000 games versus a random opponent. After training the model, we ask the agent played 200 games with the random opponent. The results show:

- The performances of Sarsa and Q-Learning are very close. They have the best performance among all the approaches;
- The performance of Double Deep Q-Learning is better than Deep Q-Learning. Both of them perform well;
- The Heuristic Policy has the worst performance among the 5 approaches. However, 84% winning rate is still acceptable.

All the 4 model-free reinforcement learning approaches work well in this game. Both Deep Q-Learning and Double Deep Q-Learning use neural networks to estimate the Q values. However, since the game is not very complex, both state space and action space are not large. It is possible to infer the Q values for all the potential states and actions. Thus, the performances of Q-Learning and Sarsa are better because they can estimate Q values more accurate.
If we keep increasing the state space and action space size, the Deep Q-Learning and Double Deep Q-Learning will show their advantages.
As expected, the cost of the two neural networks-based models is expensive since the additional time cost on training one more neural networks.

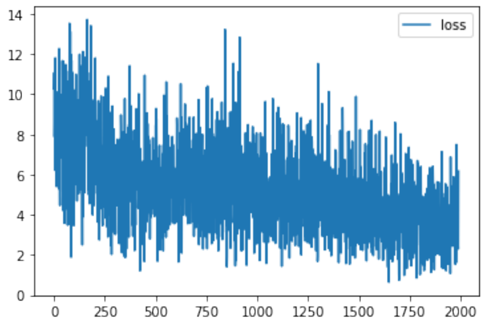

Figure 6: Loss of Deep Q-Learning.

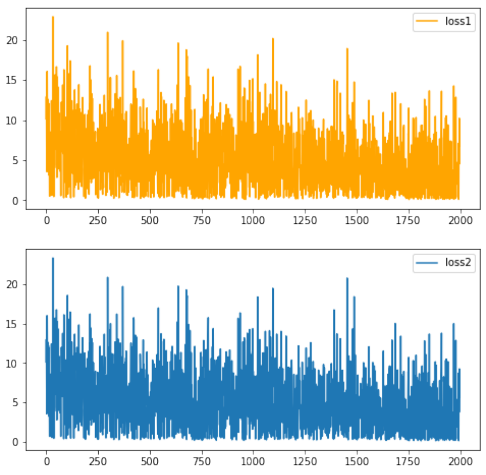

Figure 7: Loss of Double Deep Q-Learning.

6 Conclusion/Future Work

In this paper, we presented a heuristic policy and 4 reinforcement learning approaches to the Da Vince Code game. All the approaches were able to win the random policy opponent. The 4 reinforcement learning approaches had better performance than the benchmark agent. Due to the complexity of this game, Q-learning and Sarsa achieved the best success with an over 95% winning rate.
For future work, we need to consider the exemptions above. The agent could choose to stop guessing. Besides, we could add the 2 Jokers which will hugely increase the complexity of the game. Moreover, we could try the other reinforcement learning approaches mentioned in the related work section.

7 Contributions

Jiahao Zhang's work focused on the model simulation and the Deep Q-Learning approach as well as surveying the relevant works and making suggestions.
Yu Zeng's work focused on the practice of all the approaches, improving the performance of the code, and looking for the related works.
Yanlong Ma's work focused on the summary of the approaches, results, and creation of the framework of the paper.

References

[1] N. Brown, A. Lerer, S. Gross, and T. Sandholm. Deep counterfactual regret minimization, 2018.

[2] H. V. Hasselt. Double q-learning. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2613–2621. Curran Associates, Inc., 2010.

[3] J. Heinrich and D. Silver. Deep reinforcement learning from self-play in imperfect-information games, 2016.

[4] hello-world zsp. The-da-vinci-code-board-game, 2017.

[5] D. Zha, K.-H. Lai, Y. Cao, S. Huang, R. Wei, J. Guo, and X. Hu. Rlcard: A toolkit for reinforcement learning in card games. *arXiv preprint arXiv:1910.04376*, 2019.