Deep-Q Learning for the President Card Game

Pavan Kumar Bondalapati [*] [1] Shannon Paylor [*] [1]

Abstract

Since its inception, reinforcement learning has been closely tied to games. Even in highly complex games with a vast number of possible scenarios, reinforcement learning agents often have been able to learn remarkably well, exceeding the abilities of the best human players alive. Deep Q-learning is one method that uses a neural network to estimate the complex state-value functions that arise in such games. In this paper, we apply deep Q-learning to the President card game. We find that our agent is easily able to beat opponents following a random strategy, but struggles to consistently outperform opponents following a more informed strategy.

1. Introduction

Reinforcement learning (Sutton & Barto, 2018) is a powerful branch of machine learning that has been successfully applied to many problems which involve selecting the best actions to maximize some immediate or future reward. One of the most well-known applications of reinforcement learning is to games, such as Go (Silver et al., 2016) and backgammon (Tesauro, 1995). In such applications, the agent is trained to play the game, at times exhibiting a performance at the superhuman level.

In recent years, powered by improvements in computing power and the wide availability of big data, deep learning has grown quickly and shown considerable power in being able to learn complex functions. Deep Q-learning is an approach that combines reinforcement learning with neural networks to learn a Q-function when there are too many states to use a Q-table approach(Mnih et al., 2013). In this paper, we apply deep Q-learning to the President card game.

[1]School of Data Science, University of Virginia, Charlottesville, Virginia. Correspondence to: Pavan Kumar Bondalapati <pb7ak@virginia.edu>, Shannon Paylor <sep4hy@virginia.edu>.

2. Game Overview

The President card game is played with a standard 52 card deck, dealt evenly among three to six players. The goal of this game is to get rid of all the cards in your hand. The first player to finish wins the title of President, while the last remaining player receives the title of Scum. In similar fashion, the player to come in second earns the title Vice President, while the player to finish second-to-last assumes the title of Vice Scum. Given these rankings, at the start of a new round, the Scum must forfeit his two highest valued cards to the President in exchange for the President's two lowest cards. The same trade takes place between the Vice Scum and Vice President; however, they only exchange one card.

To initiate the game, the player that holds the three of clubs begins by playing this card along with any other threes in their hand. As the game progresses, the played cards are placed in the center of the table, with the most recent played cards denoted as the *target*. Given a player's hand, an eligible move is determined by the face value and card count of the target. On their turn, players can only play a single card or a set of cards that have the same face value (e.g. $\{4, 4\}$, $\{8, 8, 8\}$). An eligible move either a) matches or exceeds the face value of the target card(s) or b) has a higher card count than the target. For example, given a player's hand of $\{4, 4, 5, 5, 5, 8, 9, 9\}$, if the target is $\{6, 6\}$, then the eligible moves are $\{9, 9\}$ and $\{5, 5, 5\}$. The former move is eligible since a nine has a higher face value than a six, given that the card counts are the same. The latter move of three fives is eligible since its card count of three is greater than the target's card count of two. If the player has no eligible moves, then this player must "pass" on their turn. If all players in the round pass on the same target, then the target is cleared and play is resumed.

In this game, the lowest-ranking cards are threes, while the highest-ranking cards are Aces. Commonly referred to as "bombs," twos are special cards that clear the target. If a player has a two in their hand, then this card is always eligible to be played upon the player's turn. After playing a two, the player can elect to play another move. By rule, if a player finishes their hand on a two, then this player is automatically designated as the Scum. If the target is singular (i.e. has a card count of one) and the player plays a

Table 1. Vector Representation of State Space ($\mathcal{S}$)

Index	Description	Range
0	Number of 2s in agent's hand	$[0, 4]$
1	Number of 3s in agent's hand	$[0, 4]$
...		
12	Number of Aces in agent's hand	$[0, 4]$
13	Face value of target	$[0, 14]$
14	Number of cards in target	$[1, 3]$
15	Number of players remaining	$[1, 4]$

card that matches the face value of the target, then the next player has their turn skipped. Given four players named A, B, C, and D, suppose player A begins with the three of clubs and player B matches the target by also playing a three, then player C is skipped, and player D continues the game.

In addition to bombs and skips, completions are another event that disturb the sequential, clockwise turn order of this gameplay. A player has an eligible completion when she holds the remaining cards that would "complete" a set of four cards, given the history of the targets. At any moment in the game, the player with a valid completion is eligible to play their cards. After clearing the target by performing the completion, the player can elect to play another move. Continuing the previous example, suppose player D plays a three from his hand, skipping player A. Given that player C has the last remaining three, player C can perform the completion before player B can play her cards. For a more in-depth explanation, please visit the rules page or observe a round played on this tutorial.

3. Methods

3.1. Challenges

In our Python implementation, we simplify some elements of this card game to facilitate the construction of the environment and expedite the training process. First, we eliminate the card swapping protocol that is performed at the start of each round. If the cards are traded prior to each round, then this problem becomes a continuous task; the results of the previous round affect the formulation of the next round, which defeats the property of independence in episodic tasks. Moreover, deep Q-learning is not well-suited for continuous tasks; this complicates our goal of reaching convergence in training.

Second, we fix the policies of the opponents in order to better articulate our findings and maintain simplicity. In the game environment, the agent plays against three opponents, who all follow the same policy. This is a limitation of our Python implementation, since employing advanced techniques like multi-agent training or self-play is outside of our expertise. Nevertheless, these strategies can serve as a topic for future research.

Finally, we enable autocompletions to concentrate our training and eliminate the stochastic nature of completion events. In an actual card game, when there is a valid completion, the player with the completion must play his cards before his opponent plays their move in order to register a "completion." In a programmatic environment, this complication is an unnecessary detail since we fix the players' policy. When autocompletions are enabled, the players immediately play their eligible completions, unless doing so will result in finishing the round on a bomb. Moreover, the agent does not need to keep track of completion events in their learning, which expedites training.

3.2. Environment

Before our construction of the Deep-Q Learning model, we first must define our environment in terms of states, actions, and rewards so that the agent can then accumulate experience and learn from training.

The President card game is a game with imperfect information because the agent cannot keep track of the hands of the opponents. As a result, the agent must learn by partially observing the game. Given three opponents in the environment, an approximation for the initial state space is the following:

$$\mathcal{S}_i \approx \binom{52}{13} = 635{,}013{,}559{,}600$$

Given the magnitude of this state space, we cannot simply map the states to natural numbers for our model. Instead, we encode the state space as a linear combination of features, creating a feature vector. For our state space, we represent all components of the environment observable to the agent, including the agent's hand, the target cards, and the number of players remaining in the game. We encode a vector consisting of 16 features, as shown in Table 1.

The action space is limited to 50 actions. Unlike the state space, we can map these actions to natural numbers to construct the action space, as shown in Table 2.

Table 2. Action Space ($\mathcal{A}$)

Index	Action
0	Pass
1	Play a 2 (bomb)
2	Play a 3
...	
13	Play an Ace
14	Play two 3s
...	
49	Play four Aces

The agent receives a reward at the end of each episode. If the agent is first to finish the game without ending on a bomb, the agent earns the title of President and receives a reward of 5. Similarly, if the agent becomes Vice President, it receives a reward of 3. For the ranks of Vice Scum and Scum, the agent receives a reward of 1 and -1, respectively. If the agent finishes the game on a bomb, it receives a reward of -1. To teach the agent not to take illegal non-terminal actions (i.e. actions that do not match or exceed the rank or card count of the target), we assign a reward of -10 for these actions and subsequently terminate the episode. For all legal actions within an episode, the reward is 0.

3.3. Policies

As mentioned earlier, the policies of the opponents are fixed to simplify the Python environment. We define a step method that updates the environment from an action taken by the agent. Since our action space definition involves only the cards played by the agent, our step function must handle all other players' actions in between the agent's turns. To this end, we created two fixed policies for the opponents to follow. The first is a random policy, where a player randomly selects an action from among all legal actions in their hand. The second policy we implement for the opponents is a more strategic policy based on our knowledge of playing the game, which we refer to as the "pseudo-optimal" policy. Under this strategy, the player deterministically plays their lowest valued cards first, unless doing so would cause them to finish on a bomb. In both cases, we have all opponents in a single game follow the same strategy, either random or pseudo-optimal. As for the agent, the policy engaged during training is an ϵ-greedy policy.

3.4. Deep Q-Network

Our deep Q-network (DQN) was constructed in TensorFlow (Abadi et al., 2015) and integrated with the TF-Agents library (Guadarrama et al., 2018) to implement deep Q-learning. Given an input state vector, the DQN functions as a non-linear approximator that generates action values for the entire action space. From these Q values, the optimal action Q^* is defined as the action that yields the highest Q value.

For our neural network, we use a medium-depth architecture with five dense hidden layers, with weights initialized from a truncated normal distribution with a variance of 1.0. Our first hidden layer has 1024 units, and each successive layer has half as many units as the previous layer. All of the dense layers have a rectified linear unit (ReLU) activation function, including the output layer. Although it is traditional for the activation of the output layer to be linear, a ReLU activation for the output layer demonstrated a substantial improvement in training performance. The input layer has 16 units, corresponding to the 16 elements of our state space vector, and the output layer has 50 units, corresponding to the 50 possible actions. The architecture of our DQN is shown in Figure 1.

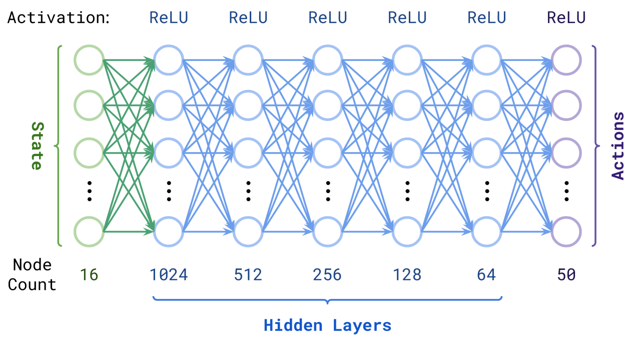

Figure 1. Architecture of deep Q-network. The input state vector is shown in green and the five dense hidden layers are shown in blue. The output layer is shown in purple. From this, the weights of the output layer map onto the action space ($\mathcal{A}$), yielding approximations for Q values.

4. Results

The average returns after training for 20,000 iterations under the two fixed policies is shown in Figure 2. From this, the agent is able to achieve average reward of between 3 and 5 when playing against opponents following a random strategy. With our reward structure, this means that on average, the agent is finishing as President or Vice President. In fact, we find that the agent is finishing as President about 70% of the time.

In contrast, when the agent competes against opponents following the pseudo-optimal strategy, the agent is considerably less successful. The average reward reached is between 0 and 2, meaning that its average finish position is around Vice Scum. We do find that our agent is able to finish as President about $20-25\%$ of the time, which is interestingly about what would be expected if all four players had equal

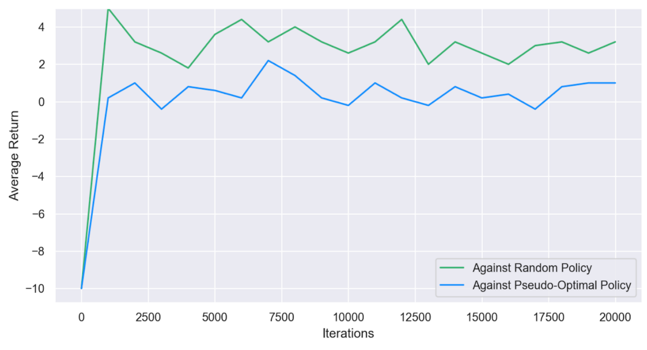

Figure 2. The average returns generated by the agent against opponents with random strategy (shown in green) and pseudo-optimal strategy (shown in blue), during the training process. Against the random policy, the agent achieves an average reward between 3 and 5. Against the pseudo-optimal policy, the agent achieves an average reward between 0 and 2.

skill and the episodes were decided by chance.

Regardless of the policy employed by the opponents, the agent appears to reach convergence within 5,000 iterations. We accredit this performance gain to our simplified Python implementation of the card game and the optimization techniques employed in the TF-Agents library. In particular, we utilized a replay buffer that stores experiences that contain information-rich trajectories. The agent is subsequently trained on a dataset that samples from these experiences.

5. Conclusions

In this paper, we outlined our approach for creating a unique Python environment for the President card game. We were able to use this environment to train a reinforcement learning agent using deep-Q learning. Our agent showed mixed results, learning enough to consistently defeat opponents playing at random, but faring worse against more advanced opponents.

One main area for future improvement on this project is in the opponents strategies. Changing the pseudo-optimal policy to be more like an ϵ-greedy pseudo-optimal policy may more closely simulate human performance. Having each opponent follow a different strategy may also do so, since some players will generally take more risks than others. A better option may be to implement a multi-agent method. Doing so would remove the bias of the opponents' strategies altogether.

Another improvement could involve further tuning of the deep Q-learning neural network. We tried several different architectures, optimizers, and learning rates, but additional tuning may still yield improved performance. All of our code is publicly available on GitHub.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

Guadarrama, S., Korattikara, A., Ramirez, O., Castro, P., Holly, E., Fishman, S., Wang, K., Gonina, E., Wu, N., Kokiopoulou, E., Sbaiz, L., Smith, J., Bartók, G., Berent, J., Harris, C., Vanhoucke, V., and Brevdo, E. TF-Agents: A library for reinforcement learning in tensorflow. https://github.com/tensorflow/agents, 2018. URL https://github.com/tensorflow/agents. [Online; accessed 25-June-2019].

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, jan 2016. ISSN 0028-0836. doi: 10.1038/nature16961. URL https://discovery.ucl.ac.uk/id/eprint/10045895/1/agz_unformatted_nature.pdf.

Sutton, R. S. and Barto, A. G. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.

Tesauro, G. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995. URL https://bkgm.com/articles/tesauro/tdl.html.