Reinforcement Learning on Spades

Theo Culhane
Department of Symbolic Systems
Stanford University
`tculhane@stanford.edu`

Abstract

Reinforcement learning on games is nothing new, but much of the literature I have read on the subject[1][2][3] has focused on games with a binary outcome (i.e. players either win or lose, and a close loss is treated identically to a blowout loss) instead of games with a graded outcome. Therefore, in this paper, I will explore how applying a similar approach to those used in the literature I saw extends into the 4 player game of Spades (for those unfamiliar with the game, see the Wikipedia page here [4]). I will eventually conclude that my particular implementation of the algorithm fails to generate useful results, but will suggest a few changes to my algorithm that, given more time and resources, may prove to fix the problems that I saw.

1 Introduction

I really enjoy playing cards with my family, and so I have made a few attempts at teaching some of my friends how to play various card games such as Spades, Hearts, and Bridge, but it takes a lot of time to get good at these games, and I don't have a very clear picture of what I do to be a good Spades player (or at least, better than a beginner). I think that it is possible that making a RL model that is good at Spades could provide interesting insight about what factors are relevant to making a good play in Spades, through an analysis of what to include in states to make the best possible Spades AI. Also, after having read the literature, I was interested in testing out similar methods to what I saw in the literature on a game with a graded outcome, such as Spades, where scoring is not precisely zero-sum and involves a very fine degree of gradation, since that seemed to be a portion missing from the literature that I found.

I am partially sharing this project with my final project for CS230, by using the same basic game engine, but using the game of Hearts instead (therefore tweaking the game engine slightly to account for the rule changes), and using Deep Q Learning instead of manual feature engineering like I do in this project. I also am using pure Q learning in the Hearts algorithm, whereas in this project I use MCTS+Q, because I was interested is seeing which was more well suited to a card game. Because the same basic game engine is used, both algorithms give identical outputs and take very similar inputs: they both output the best card to play for a given situation, and both take as input the game engines representation of the current state of the game. However, the design of the feature vector and state extractor is far more important in this project than the other one as a result of the learning being significantly more shallow, and so the way that the game data is converted into a feature vector that the actual algorithm can iterate on is slightly different.

2 Related work

The modern standard approach for reinforcement learning on games with some degree of randomness was heavily informed by TD-Gammon[5], a program that used $TD(\lambda)$ using two steps of look ahead combined with standard TD learning approaches in order to determine the best move at any particular point in time. This approach was then most recently refined in the development of AlphaZero[1] in a way that I found to be very clever and which has become the state of the art for reinforcement learning on games, which uses the same basic idea as TD-Gammon - using several moves of look ahead combined with a standard existing reinforcement learning algorithm - but made several substantial improvements. First, they switched from simple look ahead to using a Monte-Carlo Tree Search algorithm that allows searches to happen more robustly, allows finer control over how deep the look ahead happens, and allows the search to use heuristics to make the search go deeper on more probable paths without wasting time searching paths that are unlikely to turn up useful results. The second, and perhaps more important, improvement that AlphaZero made was switching from the relatively shallow network that TD-Gammon uses for its TD learning to a very deep Q learning network. This allowed the computer to use significantly less prior knowledge about how the game should be played, which in turn allowed a wider array of possible strategies. By removing the element of human decision making from AlphaZero's code, AlphaZero was able to pursue strategies that no human had ever given much thought to, and eventually lead to a much better strategy than any human had ever played, hence how consistently AlphaZero is able to beat human Go grandmasters. This approach of MCTS+Q has then been replicated in several more works that informed my work heavily. One of these is *RLCard*[6], which focuses on card games such as Hold 'Em, Blackjack, and Gin Rummy among others as examples of games in which a player has imperfect information about the state of the game, but does not cover either Hearts or Spades. Another is *A Simple Alpha(Go) Zero Tutorial*[3], which uses the board game Othello in order to illustrate the approach that AlphaZero used, but which focuses on perfect information 2 player board games and so deviates from my actual implementation of a 4 agent imperfect information game. One final project that I found particularly helpful was *Tackling the UNO Card Game with Reinforcement Learning*[2], which uses the game of Uno to illustrate MCTS+Q, and is the closest work I found to the implementation that I ended up using. However, this project used a Q table instead of a network for Q learning, which would not be feasible in Spades as the state space is far too large.

3 Dataset and Features

The dataset was generated entirely through self play and playing against other agents within the same program structure. The program progresses in two phases, pre-training and training, each with its own dataset. In the pretraining phase, the MCTS+Q model competes against my baseline model for 1000 games, with two out of the four players using the MCTS+Q model to generate actions and the other two using the baseline model, with each model being partners with a model of the same type. After the pretraining phase, the dataset is reset, and the training phase uses 4 MCTS+Q agents, with learning occurring on a maximum of 10 prior iterations, which ends up being a maximum of the past 500 games played. The main experimentation was on the features used for generating actions, and so there are 4 different sets of features that were tested out. The initial feature map that I tested out incorporated the number of each suit already played in a game, the number of each suit in the current player's hand, the number of tricks that the current player has won, the total number of tricks played in the game so far, a boolean for whether the current player is capable of taking the current trick, a boolean for whether the current player is capable of not taking the current trick, and a boolean for whether the current player's team is currently winning the trick. So, for example, lets say that in a particular game, there have been 5 Spades played, 8 Hearts played, 6 Clubs played, and 5 Diamonds played, the player has won 3 tricks, the current trick contains the 8 and 5 of diamonds, and the player has the King of diamonds, the 4 of diamonds, the 10 of clubs, the king and 3 of spades, and 8 and jack of hearts. Then, the feature vector would look like [5, 8, 6, 5, 2, 4, 1, 2, 3, 6, 1, 1, 1]. However, this feature map seemed to perform pretty poorly, and so I also tested adding a "point count" feature to the vector, in which each ace in the players hand counts for 4, each king for 3, each queen for 2, and each jack for 1. For my third feature map, I tried also appending the number of spades of rank jack or higher in their hand a player had, and for my 4th feature map I tried removing the features for how many tricks were played, whether the current trick can be won or ducked, and whether the player's team is currently winning the particular trick, and added in the number of each suit of rank jack or higher in their hand a player had. So, in the earlier example, the feature vector would look like [5, 8, 6, 5, 2, 4, 1, 2, 3, 7, 1, 1, 1, 1, 0].

4 Methods

There are three algorithms used in this particular project, 2 baselines and then a main, trainable one. The first baseline is a completely random agent. At each time step, it determines all of the legal moves it could make, and then chooses one of those at random. This algorithm doesn't do very well, but is a good minimum on which to judge the other algorithms, since if an algorithm can't beat random, it likely means it learned nothing. The second baseline is a greedy algorithm. At each time step, if it can win a trick, it will, using its highest available action. In the event that it cannot win the current trick, it will play the lowest available card, with ties being broken randomly. For leading, it will always lead its highest available card. This algorithm is very similar to the strategy that I used when I first was learning to play Spades, and so is supposed to simulate how an absolute beginner would play. The main, trainable algorithm uses a Monte Carlo Tree Search sped up using Q learning, very similar to how AlphaZero is described in the AlphaZero tutorial[3]. The Monte Carlo part works similar to a standard Monte Carlo Tree search rollout as described in *Reinforcement Learning: An introduction*[7], in which each node is explored in turn, with each node explored next being decided by maximizing the upper confidence bound described by $U(s,a) = Q(s,a) + c_{puct} \cdot \frac{\sqrt{\Sigma_b N(s,b)}}{1+N(s,a)}$ where $Q(s,a)$ is the return value of the Q network on state s and action a, $N(s,a)$ is the number of times that we have already taken action a from state s, and c_{puct} is a hyper parameter determining the exploration/exploitation constant, for which I ended up deciding on a value of 15. However, unlike a standard MCTS rollout, instead of searching to a terminal state, whenever a new, not yet seen node is encountered, instead of finding the maximum $U(s,a)$ for state s, the Q value, $Q(s,a)$ is returned, thus significantly reducing the computation time of each iteration, while also doing a reasonably good job of estimating the best play, especially after enough training has occurred that $Q(s,a)$ is accurate. We repeat this search process, going until we either hit an unexplored node or a terminal node, a set number of times, with the number of times we run this search as a tunable hyper parameter where lower numbers lead to greater speed and higher numbers lead to greater accuracy. I, after some experimentation, determined that for this application repeating this process 10 times gave me as good of accuracy as higher numbers, tested up to 30 in increments of 5, tended to, but ran significantly faster than any higher number did. Then, after this set number of explorations, we create the policy for determining the next action to take from state s by assigning the probability that each action a is taken according to the formula $\frac{N(s,a)}{\Sigma_b N(s,b)}$, and then determine our action by sampling. The network behind $Q(s,a)$ is a 2 layer network with a 100 node hidden layer with a ReLU activation, and is trained according to the loss function $l = \sum_t (Q(s_t) - z_t)^2$ where z_t is the final score that the current player at time t received. Using z_t as the target value for training makes the unfortunate assumption that every action taken in a game where a particular player won was a good action for that player to take at that time, which is not necessarily the case. For example, the player may make a terrible move, but by random chance the move works out in their favor when it shouldn't have, or the player may make a terrible move that turns out poorly, but later makes up for it with several good moves. However, the literature suggests that this assumption ends up not causing problems in the aggregate, as once enough moves are seen the good moves will on average yield better results, so after enough games the algorithm will be able to separate which were the good moves and which were the bad.

5 Experiments/Results/Discussion

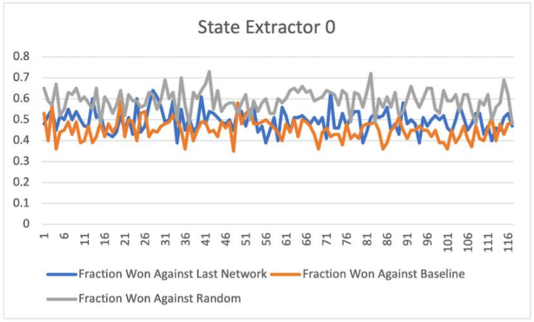
The main hyper parameters that I worked on tuning were the exploration constant c_{puct}, the number of MCTS simulations to run before choosing an action, and the number of episodes to generate in between training iterations.

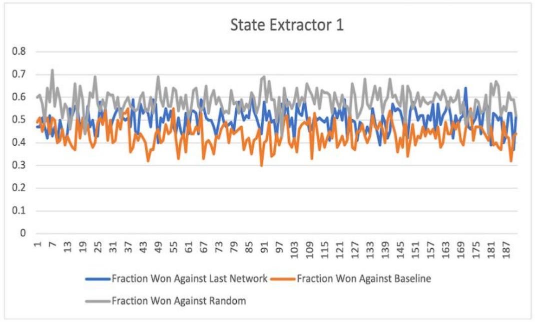
Starting out with c_{puct}, which controls the trade off between the algorithm acting greedily and the algorithm exploring other options for possible actions, the tutorial that I was basing my decisions on used a c_{puct} value equal to 1, which is what I initially tried. However, because the rewards for the game the tutorial was using was on the scale of -1 to 1, and my rewards were on a scale between -200 and 200 (with rewards concentrated in the range of -60 to 60), using a c_{puct} of 1 seemed to be giving excessively greedy results. Therefore, I tried scaling up the value, and eventually realized that searching in increments of 5 gave me results that were significantly different enough, so I tried all values between 0 and 35 in increments of 5, and found that 15 gave me the best accuracy. Higher values ended up exploring too aggressively (i.e. searched too broadly instead of using the heuristics to narrow the search), and so eliminated the benefit that MCTS gave in doing a more heuristic driven search. Smaller values ended up exploiting too aggressively, and so the algorithm never explored enough to learn whether another play might have been better, and so the algorithm ended up basically stuck on the same, almost random strategy that the initialization of the network gave.

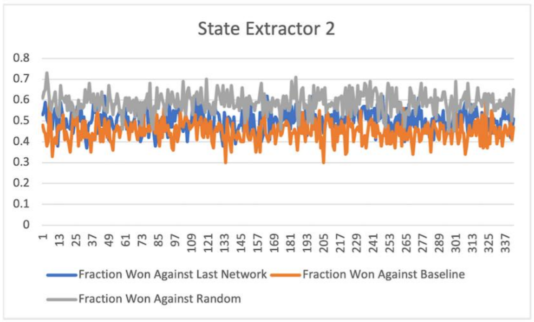
Next I worked on the number of MCTS simulations to run before choosing an action, which controls how many different actions it will simulate before deciding which action to take (Note: these actions need not be taken by the current player. It is just the total number of actions simulated per time step). This also controls the trade off between finding the ideal action and speed of finding an action, and so I was searching for the lowest possible value that still was able to learn. I tried these in increments of 5 as well, testing from 5 to 30, and also specially testing 13 as that is the theoretical maximum number legal actions a player could have at a particular time step (although actually having 13 legal actions rarely actually happens). Here, I found that the ideal was 10. Much more, and games took too long and so I was unable to test new ideas fast enough to be practical, but any lower and the results weren't good enough to justify. I also noticed an interesting dip between 20 and 25 where performance was actually worse, instead of better, than 10 and 15. This may be because the future moves the opponent makes are too unpredictable, and so simulating too far in advance may make too many assumptions, although I can't say for certain this is the reason.

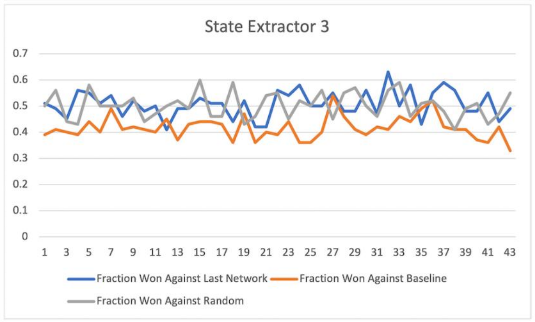
Finally, I looked for the optimum number of episodes, or number of full games on a particular strategy, to generate per training cycle. I tested between 10 and 60 in increments of 10 between 75 and 200 in increments of 25. In general, more episodes increased performance, although I found a point of significantly diminishing returns around 50 for regular training cycles. However, for pretraining cycles, I didn't see this point of diminishing returns until 100. Therefore, I used an episode count of 50 for regular training and 100 for pretraining.

My results, plotted with fraction won against various agents on the Y axis and number of iterations run on the X axis, is as follows:









I used performance against the two baseline algorithms described earlier along with performance against the previous iteration of the algorithm as my evaluation metric. Each of these algorithms was run until the loss reported that it converged, accounting for the significant difference in how long each algorithm was run for. After each training iteration, if the new algorithm could not win at least 50% of the time against the previous iteration, the new version is thrown out and the old version is restored and trained on instead. Surprisingly, despite having the most consistently good results against the random agent, state extractor 2 took the longest to converge, possibly because had the most features that needed to be fit. However, state extractor 3 was also tied for that number of features, and took the least amount of time to converge, and so it is possible that state extractor 2 simply had the most complicated features, and so were the hardest to fit. None of these state extractors got nearly as good of performance as I was hoping for, possibly stemming from how shallow the network estimating Q values was, meaning that the complexity of the function that it could learn was limited. Further, the states that were extracted didn't extract many features, as computational time was at a premium. Therefore, it is possible that the states didn't have enough data to make very good decisions, and so the 70% win rate against a random agent that state extractor 2 was able to achieve was the best any state could be expected to do given the constraints put on the problem.

6 Conclusion/Future Work

As we saw, none of the possible feature maps I was able to come up with performed very well, with the feature map that worked the best, 2, being the one that had the most data incorporated into it. The poor performance is likely due to how complicated the game of Spades is, and how a small number of features can't hope to capture enough information to make a good decision about what card to play next. However, had too many features been used, a network as shallow as the one used here, chosen because of constraints on the time and processing power available to me, wouldn't have been able to use the data given to it, since there are so many nuances in how a card hand needs to be evaluated. Therefore, I think the main path for future work would be to try using a more informative feature map combined with a more expressive neural network, and test if that would be able to learn more.

7 Contributions

As I am working on this project by myself, all work described in this paper is my contribution. Special thanks to my TAs in CS230 and CS229 for helping me with various parts of this project and the related project, and for pointing me towards the resources I needed to figure out MCTS+Q. The template for this document was taken from NeurIPS 2020.

References

[1] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, October 2017.

[2] Bernhard Pfann. Tackling uno card game with reinforcement learning, Jan 2021.

[3] Surag Nair. Simple alpha(go) zero tutorial, 2017.

[4] Wikipedia contributors. Spades (card game) — Wikipedia, the free encyclopedia, 2004. [Online; accessed 1-June-2021].

[5] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, March 1995.

[6] Daochen Zha, Kwei-Herng Lai, Yuanpu Cao, Songyi Huang, Ruzhe Wei, Junyu Guo, and Xia Hu. Rlcard: A toolkit for reinforcement learning in card games, 2020.

[7] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.