CS230

Reinforcement Learning on Hearts

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

Abstract

Deep reinforcement learning on a zero sum game is nothing new, but very few of the papers that I was able to find dealt with games of more than two agents. In this paper, I explore how deep reinforcement learning can be applied to the 4-person game of hearts, using Deep Q Learning. I conclude that Deep Q Learning is a good candidate for solving a 4-person, imperfect information zero sum game such as Hearts, but that the computational power available to me was insufficient to fully solve it in this instance.

1 Introduction

I enjoy playing Hearts with my family, but most computer players are generally terrible. I was curious why computerized Hearts players are so unskilled, while the computer opponents for games like chess and go are excellent (to the point that even a cheap consumer chess app these days is better than most human grandmasters [1]). While some of this can be attributed to chess being game of perfect information, and card games such as Hearts rely on randomness and incomplete information for any given player, I realized that there might be more to the story. Specifically, I was unable to find much work outlining how reinforcement learning applies to games of more than 2 players, meaning that the average programmer is unlikely to come across a robust, winning algorithm for games like Hearts, since they would have to search very hard to find anything even passable. I decided to explore how Deep Q Learning applies to Hearts, in which 4 agents compete for the same value instead of 2, making the value estimation slightly more complicated, and the rules create opportunities for the scoring to be slightly off from zero sum (such as shooting the moon allowing the total score to be 78 instead of 26). Beyond that, Hearts is a good candidate for deep RL, because the state space of a hand of cards is very large, and so some method of reducing the size of the state space is necessary to run an RL algorithm in any realistic amount of time. The input for my algorithm is a simple Hearts simulator, where at each time step the current state of the game is distilled into a 217 entry vector including a 52 entry one hot encoding for which cards have already been played in a game, 52 entries dedicated to a one hot representation for what cards are currently in play in a particular round, or "trick", 4 entries dedicated to a one hot encoding for who the current player is, 4 entries dedicated to a one hot representation of what suit the current trick is, 52 entries dedicated to a one hot encoding for what cards the player has in their hand, one dedicated to a boolean for whether it is possible for the player to "follow suit", and 52 dedicated to what the current action being evaluated is. The play is then evaluated, and for each play the model outputs an expected reward for making that particular play, which will be a floating point somewhere between -26 and 103 (although very few predictions should be greater than 26).
The program shares a basic game engine with a project I am doing concurrently for CS229, in which I am doing reinforcement learning on the card game Spades. However, the actual algorithm used is different for the two projects, as the Spades project uses a shallow Q network and Monte Carlo Tree

CS230: Deep Learning, Spring 2021, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

Search, and this project uses a deep Q network with no MCTS, and the game engine is modified for that project to account for the differences between Spades and Hearts. Note, however, that due to the basic premise of the problems being very similar, the related work section is very similar.

2 Related work

The current approach to reinforcement learning in games with randomness as a key premise was first seen in TD-Gammon [2], which played the game backgammon and used $TD(\lambda)$ using two steps of look ahead combined with standard TD learning to determine the best action at each move of the game. However, TD-Gammon changed the field by using a shallow neural network to learn a more complex evaluation function, instead of using some form of hard coded evaluation function or a simple Q table. This allows a simpler program to evaluate a larger state space, and to learn general patterns faster. This approach was then refined into what is now the state-of-the-art approach for reinforcement learning on games by DeepMind in their AlphaGo/AlphaGo Zero/AlphaZero series [3]. These programs started with the TD-Gammon approach of looking several moves into the future and then using an evaluation function to determine which of those moves leads to the best result, but made two significant improvements. First, AlphaZero uses a better heuristic search algorithm, Monte Carlo Tree Search, to find which moves it should evaluate, whereas TD-Gammon's search algorithm is fairly simple. MCTS finds high-potential future moves, and only evaluates those, ignoring low-potential moves, which significantly cuts down the computation time expanding the search tree and also allows deeper searches on the actions that are likely to be useful, allowing for an overall stronger algorithm. Second, and more important, AlphaZero used a significantly deeper network to estimate the expected return of certain actions, which allows more complexity in what the network is able to evaluate. This allows the computer to use less prior knowledge about the game, as the input can be more strategy neutral, such as board positions instead of human designed feature maps. By removing the element of human decision making, AlphaZero pursued strategies that no human had ever given much thought to, and eventually led to a much better strategy than any human had ever played, hence how consistently AlphaZero is able to beat human Go grandmasters. MCTS+Q has been replicated in several more works that heavily informed my work. One is *RLCard*[4], 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*[5], which uses the board game Othello to illustrate the AlphaZero approach, but which focuses on perfect information 2 player board games and so deviates from my actual implementation of a 4 agent imperfect information game. *Tackling the UNO Card Game with Reinforcement Learning*[6] was particularly helpful, and applies MCTS+Q to Uno; this 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 Hearts as the state space is far too large.

3 Dataset and Features

The dataset is generated entirely through self play, with dataset generation progressing in two steps. In the first step, pretraining, 5000 games are generated using the baseline model, which is a greedy algorithm designed to simulate how a beginner plays. The data is then trained on for a total of 13 cycles of 20 epochs each. Note that it is important that it does 13 cycles of 20 epochs, instead of simply 260 epochs, because the target outputs during a training cycle relies on the current predictions of the evaluation function to solve Bellman's equation $y = r_s + \gamma \underset{a'}{max}(Q(s', a'))$, and so these targets must be recalculated several times throughout training for the algorithm to learn the proper targets. I chose 13 because there are 13 tricks in a given game, and so max Q values can only influence at most 13 targets ahead of them, if the max Q value is that of a terminal state. After pretraining, the dataset is reset, and training begins. The dataset for training is used by stepping forward in a game one trick at a time, and then adding that trick from each players perspective to a replay, and randomly sampling 800 datapoints out of the replay as a minibatch to train on. The replay extends back for a maximum of 100 games worth of datapoints, which is enough that new episodes aren't over-weighted in the algorithm's learning, but is small enough that datapoints won't grow too stale to be useful before being evicted from the replay.

When a datapoint is added to the replay, it is in the form (s_i, r_i, s'_i, t_s), where s_i is a vector representation of the state from player i's perspective and the action player i took, the same vector representation described above in the Introduction section (a concatenation of the s and a in $Q(s, a)$ such that the s_i stored in a datapoint is ready to put directly into the evaluation network), r_i is the reward received from taking action a_i from state s_i, s'_i is the next state, the state reached by taking action a_i from state s_i, and t_s is a marker denoting whether state s_i is a terminal state. This means that any time t_s is true, s'_i will be NULL, and any time s'_i is not NULL, t_s will be false. The reward r_i is determined by taking the point differential from state s_i to state s'_i for player i. So, for example, if player 1 had 3 points in state s_1, and 5 points in state s'_1, then r_1 would be -2 (negative because points are bad in Hearts), and r_2, r_3, and r_4 would all be 2 (no more than one player can score points in a particular trick except in special circumstances, such as the use of a rule called "shooting the moon", and so it is usually the case that three of the 4 players have the same reward, and the fourth player has negative that same reward).

4 Methods

The algorithm follows a standard Deep Q Learning model. Start out by initializing a Q network of the correct size and shape (more on what the correct size and shape is, as that was the subject of some of my experimentation), and initialize a replay memory pool, which I used a Python list for since order into memory matters for my implementation. Next, I generate the 5,000 pretraining games described above, and add them all into my replay. Because the dataset is generated entirely through self play, I saw significantly better results significantly faster after implementing pretraining, because it helps to mitigate the cold start problem that many algorithms that use self play as their primary source of data struggle with. I train on the full replay for 13 iterations of 20 epochs, and then reset my replay and begin training. For training, in each episode, I generate a new random start state for the game by randomly initializing a deck of cards and then dealing it out to each player. I then loop over the tricks in the generated game, and in each trick loop over each player who plays according to an epsilon-greedy action algorithm. In epsilon greedy, at each state s, player takes valid action a such that $a = \underset{a}{max} Q(s, a)$ with probability $1 - \epsilon$, and with probability epsilon instead takes a random valid action a. The algorithm then observes the reward r and state s' that taking action a from s took it to, and adds the experience (c, r, s', t) to the replay, with c being the same concatenation of s and a as described above, and t being a boolean for whether state s was a terminal state. As noted above, s' will be NULL whenever t is true. Also note that due to implementation constraints, s' wasn't specifically observable immediately, and so instead there was a cache that would handle adding (c, r, s', t) to the replay once either s' had become observable or t was true, but the algorithm should work the same regardless of the specifics of the implementation here. Next, a random minibatch of 800 datapoints was sampled from the replay, and each of these datapoints was processed to generate a target y according to the function $y = r + (1 - t)\gamma(\underset{a'}{max}(Q(s', a')))$, in which t is evaluated as an integer instead of a boolean and this equation short circuits such that if t is 1, i.e. true, $\gamma(\underset{a'}{max}(Q(s', a')))$ would be multiplied by 0 and is not evaluated, which is important because s' would be NULL in this event and so finding $\underset{a'}{max}(Q(s', a'))$ would be non-nonsensical. Finally, the model is trained on this minibatch of (c, y) pairs using mean squared error, i.e. the loss function $L = \frac{1}{m} \sum_{i=1}^{m} (Q(c^{(i)}) - y^{(i)})^2$, where m is the size of the minibatch (in this case, 800), and $(c^{(i)}, y^{(i)})$ is the ith pair. This algorithm is run until either 5000 games have been simulated, or until the change in loss from one trick to the next is less than .1 for 5 tricks in a row.

5 Experiments/Results/Discussion

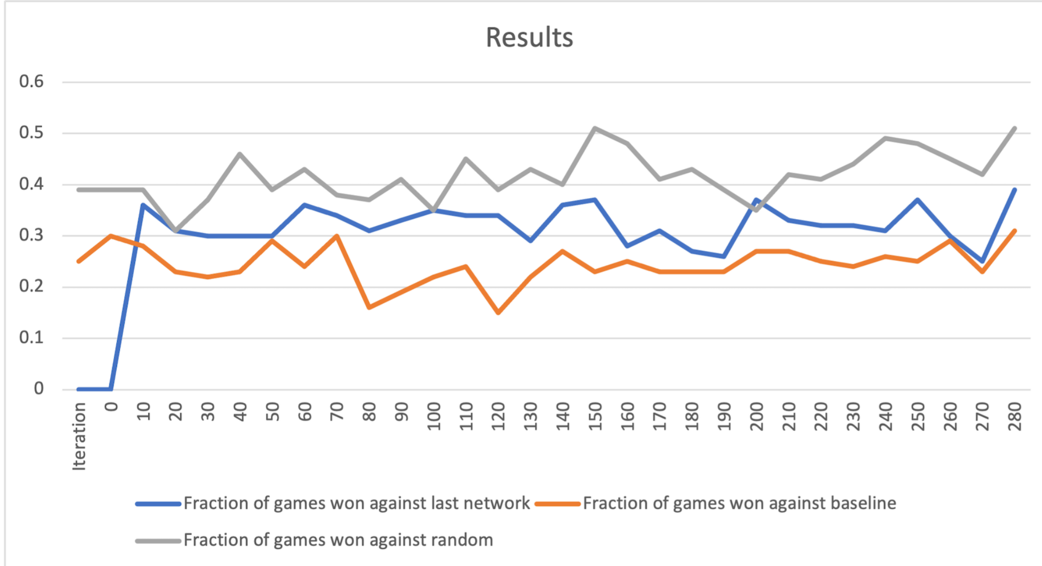
Due to the nature of deep Q learning, there were too many hyper parameters in this algorithm to feasibly test different values for all of them, so for many I accepted default values or the values suggested in the literature without testing variations. For maximum number of possible episodes, I tried to pick a limit that would hardly ever get reached, but that would be low enough that if the algorithm failed to learn it would be able to hit the upper limit before wasting too much time waiting for convergence. Convergence generally occurred between 200 and 500 episodes, and so I

picked 5000 as the upper limit of episodes (10x the maximum expected convergence time). I chose 5200, or 100 full games for max replay size, because *A Simple Alpha(Go) Zero Tutorial*[5] trains on 100 full games during each training cycle. Finally, my algorithm uses Adam as its optimization algorithm, and I used most of the TensorFlow defaults for Adam, operating on the belief that the TensorFlow defaults are well tested-except that I changed the epsilon value when I had an issue with instability in my loss, and one helpful blog post[7] suggested that changing Adam's epsilon value to .1 from the default might help. The hyper parameters that I spent the most time tuning was γ in the Bellman Equation, number of training epochs to train per training iteration, frequency of training, mini batch size, number of layers in my network, and various aspects of each layer like activation and regularization which I tested in groups.

The γ value in the Bellman equation had the largest impact on my results. I tried values between 0 and 1 inclusive in increments of .1, expecting from my prior knowledge of how I play cards that the optimum value would likely be .6 or .7, since it is important to think several rounds ahead during Hearts, but due to the unpredictability of what other players have in their hand and what they do, thinking too far ahead can lead to an overly conservative strategy. Upon trial, however, I found that γ of 1 - no discount on future rewards at all - gave the best results, meaning that the computer wins most frequently when it considers to the last trick starting when it evaluates the first trick. There was an almost linear relationship between γ and performance, with $\gamma = 0$ having equal performance to a fully random, strategy-less algorithm such as the first baseline. The next biggest changes came from number of epochs trained for during each training cycle. I found that a single epoch led to the best results, with more epochs tending to lead to over-fitting on the training data and therefore worse performance at test time. Excessive epochs also led to very low loss being reported and very high stability in the loss being calculated, and since the program decides when to stop training based on stability of loss, higher epoch counts also halted training prematurely, compounding the problem of a lack of generalization since more training time leads to a wider variety of scenarios to train on, by the nature of generating the dataset through self play. I found a similar problem with over-fitting when I tried to train after each player had played, and this problem decreased when I trained only after each trick. For minibatch sizes, I found that training time scaled roughly linearly with minibatch size, and I tried values between 52 (one full game's worth of datapoints) and the full replay, testing 104, 208, 520, 800, 1000, 1500, 2080, and 5200. Anything below 520 gave unpredictable (and nearly useless) results, and anything above 800 gave roughly the same results, with only very slightly increased stability as the full 5200 was approached. Therefore, 800 seemed like the best option, since it was fast enough for training and testing different ideas to be feasible, but gave almost as good of results as training on the full replay. For the number of layers in my network, I based my decision heavily on the number of layers used in *A Simple Alpha(Go) Zero Tutorial*[5], which used an 8 layer network. I also tried 5, 10, and 16, before settling on 8. I found that 5 layers weren't able to differentiate subtly different positions, and specifically struggled to tell whether a particular action was "following suit" (meaning that the suit of the card played is the same as the suit of the card led), or if the action was instead "dumping", (meaning that the suit of the card played is different from the suit of the card led), which in Hearts is a critical difference that determines whether it is possible for the player to win the trick or not. Therefore, 5 couldn't learn much more than the general heuristic that in Hearts high cards are bad and low cards are good. By contrast, 10 and 16 were able to over-fit to the training data, which like too many epochs, worked extremely well on familiar positions but very poorly in new situations; in Hearts, where there as so many possibilities, this meant consistently poor results. Therefore, I returned to 8, which was expressive enough to capture the subtleties in positions but did a good job of learning general patterns of play instead of over fitting to the data points. Finally, I worked on fine-tuning the parameters that my network used. For number of nodes per layer, I noticed that *A Simple Alpha(Go) Zero Tutorial*[5] used powers of 2 for all of his layer sizes, and so I decided to do the same, and I also noted that, like many machine learning projects, they also had their layer sizes decrease or remain the same size at each successive layer, and so I also did the same with that. I found that a layer size of 2048, followed by 3 layers of 1024, then 2 layers of 512, then 3 layers of 256 gave the best results. I tried several tests with initial layers larger than 2048, such as a 4096 node layer, but this led to excessive training time with almost no increase in accuracy. Smaller layers, like 128 and even one as small as 64, decreased accuracy too much to justify the speed. For layer activation, I started with ReLU for all of the layers, but found that my accuracy suffered greatly as training progressed, and by examining the weights discovered that this was due to a "dying ReLU" problem, in which weights became negative, leading to a saturation of the $max(0, x)$ operation which means that some nodes always returned 0. Therefore, I switched to a Leaky ReLU activation with

an α of .03 (i.e. the operation $max(.03x, x)$), which allowed the network to recover from the dying ReLUs, and create more stability. Finally, partway through training I saw a few occasions where one anomalous datapoint would significantly impact the weights the network learned, and so to fight over-fitting to anomalous datapoints I use both L2 regularization and gradient clipping, which helped to avoid the exploding loss problem that these datapoints were creating.

The results for my main program are as follows:



Note that data before pretraining is not included in the graph, but that in my testing, a fully untrained network beat the baseline model 15% of the time, and the random model 25-30% of the time. So, even though the graph only shows a shallow upward trend, the starting point after pretraining is significantly better than a fully untrained algorithm. Also note that, due to this being a 4 player game, 4 perfectly matched opponents would each win 25% of the time, so all values being below 50% is expected. This means that the algorithm eventually became close to evenly matched with my baseline, and significantly better than random, and as the blue line shows was consistently improving, as the blue line rarely dips below 30%. Though I was hoping for my algorithm to get to human level play, and this wasn't achieved, these results imply the general approach is one the right track, and that perhaps with just a few more tweaks it could achieve human level play.

6 Conclusion/Future Work

I think there are several possible avenues for future work on this algorithm. The first, which I think might yield the most improvement, would be to incorporate an element of MCTS to action decisions, as AlphaZero uses. This would allow the algorithm to test its possible actions before actually deciding that the action with the highest expected return is the best one, which I think would significantly improve its play, especially early on before it has seen sufficient examples for its Q estimations to be very accurate. That would lead to more learning faster, which I think would speed up the overall process, with the added bonus that those results might be even better than those reached here. Another possible area of future work would be testing if learning on Hearts transfers to a similar game, like Bridge, or whether Hearts is a unique enough game that there isn't room for transfer learning.

7 Contributions

As I am working on this project by myself, all work described in this paper is my contribution. Though this project shared some base game engine code with a project done for CS229, the Deep Q Algorithm itself was used solely for this class. 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 Deep Q Learning. The template for this document was taken from NeurIPS 2020.

References

[1] Garry Kasparov. The chess master and the computer. *The New York Review of Books*, 57(2):16–19, 2010.

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

[3] 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.

[4] 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.

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

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

[7] Lak Lakshmanan. Debugging a machine learning model written in tensorflow and keras, Feb 2019.

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