CS3243 Poker Project - Team 10

Goh Yin Hao, Lim Ming En, Neo Si Hao, Tan Ying Zi Rebecca
A0155167B, A0155160N, A0155994L, A0158203M
National University of Singapore

1 Introduction

Mathematically, a Limit Texas Hold'em game can have as many as $3 * 10^{17}$ states (Pedro, 2013). An ideal agent searches through an entire game tree and outputs the best decision based on actual outcomes, but is extremely inefficient. Given time constraints and limitations in computational power, it is practically impossible and unrealistic to design such an agent for real game scenarios.

Using a depth-limited minimax search algorithm and an evaluation function, the goal of our agent is to deliver optimal game play decisions in an efficient manner. We chose to implement a minimax algorithm because it has proven to be effective for adversarial game such as chess and checkers (Anton, 1989). These are perfect-information games which differ in nature from stochastic poker games with limited information. For a poker game, there is a much greater degree of branching, which increases the search space and thus the time taken to complete the search. At the same time, it is found that a deeper search increases the effectiveness of a minimax strategy (Anton, 1989).

In our project, our goal is to utilize good heuristics for an accurate estimation of the "worth" of each node. This is coupled with pruning to avoid making redundant searches, so that a greater depth can be reached with the same amount of time. We retain the idea of a backtracking search, so as to decide on the best move based on the favourability of resulting outcomes.

1.1 Our Approach

The following outlines our approach and methodology:

1. Implement a depth-limited minimax search algorithm for tree generation and search (*Refer to section 2.1*)
2. Design an evaluation function for estimating node favourability (*Refer to section 2.4*)
3. Train the agent in an iterative fashion, pitting our agent against one other in order to obtain the most optimal set of weights (*Refer to section 2.5*)
4. Pit the agent against other agents, observing and analyzing its performance (*Refer to section 3*)

1.2 Results Achieved

We pitted our agent against a random player, a raise-only player and a call-only player. Subsequently, we also made it play against the agents of other groups. However, our agent did not fair as well as we expected. The reasons for its sub-par performance are analyzed in sections *3 Observations and Analysis* and *4 Limitations*.

2 Agent Design and Implementation

2.1 Minimax Search

The purpose of the minimax search algorithm is to decide on the optimal move to make such that it could potentially lead to a win for our agent. The minimax search algorithm will generate nodes, each corresponding to a possible action. The available legal actions are "fold", "raise" and "call". Each time the minimax algorithm is called, the current node, known as the MAX player node, will generate 3 nodes, each representing one action. The nodes generated by the MAX player represent either a MIN player or a chance node. The MIN player nodes are nodes simulating the actions that might be taken by the opponent, while the chance nodes simulate the revelation of community cards.

The MIN player and chance nodes will then generate their own child nodes, and these nodes generate further child nodes, until the terminal node is reached. The MIN player will generate nodes corresponding to the actions it is allowed to make. The chance node will simulate all possible combinations of cards that could be revealed in the community cards, then produces nodes that corresponds to the actions the next player can take. The terminal node is achieved when either all community cards have been revealed whereby no further actions could be called or when either player decides to fold.

Once the terminal node is reached, the evaluation function will be called to determine the value of each node. For a depth-limited search, this function is also called when a predetermined maximum depth has been reached. The MAX player will choose its child node with the highest value (ie maximising utility) while the MIN player will choose its child node with the lowest value (ie minimising utility). The node that is chosen by the root node (MAX player) will determine the action that will be played by our agent during that round.

Alpha-beta pruning was also implemented into the minimax search algorithm. This will decrease the number of nodes being evaluated, hence improving the total run time of the minimax search algorithm.

2.2 Bucketing

Due to the nature of the game, there is a need to deal with imperfect information and uncertainty, which in this case refers to the unpredictability in the revelation of community cards after each round. As a result, the search tree becomes incredibly large, since a chance node has to be generated for each combination of cards that can be possibly revealed. For example, after the preflop round, 3 community cards are generated. The number of possible combinations for this would be $\binom{48}{3}$ =17,296. Subsequently, 2 community cards are revealed one at a time, again introducing a huge number of possibilities. It is not possible to generate such a large search tree in the time constraint of 200ms per move.

One solution to this is a technique called bucketing. This technique involves categorizing sets of cards into buckets, and card sets within the same bucket will be dealt with the same strategy. This mimics the way humans play poker. For example, our strategy might be the same if we have a King and a 4, as opposed to a King and a 3 (assuming that no other cards are known yet) (Pedro, 2013). Bucketing significantly reduces the number of situations that need to be considered by the agent especially during the revelation of community cards.

To determine which bucket a card combination falls into, the hand strength of the player's hole cards combined with the community cards under consideration was computed. A function that evaluates hand strength (similar to the one used in evaluating terminal nodes but does not take into account pot and street values) was used. During the revelation of community cards, each possible combination were evaluated using the evaluation function and then categorized into "buckets" according to the returned value rounded down to the nearest whole number, we term a "bucket number". Each chance node has an attribute "bucket number", as well as a "bucket count", i.e. how many card combinations have been placed into this bucket. For each possible combination, we check if the bucket number already exists. If so, the bucket count of the particular child node is incremented. Otherwise, a new child node with this bucket number will be created. The result is that instead of generating 17,296 chance nodes, the number of chance nodes generated and to be enumerated later on can be reduced to an order of 10.

However, despite the significant reduction in nodes generated, the time taken was still outside of the time constraint. Thus, we finally decided to generate a single random chance node instead of enumerating through all possible combinations of cards.

2.3 Decision Tree

Other attempts were made to mitigate the issue of imperfect information and the uncertainty of the card draws. One notable method was to simulate multiple runs within each turn, with each run utilizing the minimax tree as described in the previous sections. The overall win rate out of all these runs will then be computed and compared against certain thresholds, which will be determined by training the model. Essentially, the method entails utilizing the minimax search algorithm within a simple decision tree.

However, the main drawback of this method is its time complexity. With the minimax search algorithm alone, each turn requires up to 30ms to compute. This means that each simulation of a full round would minimally take 150ms and it would not be possible to run more than one simulation given the turn limit of 200ms. As such, this method was not put in place due to its need for a long computational time.

2.4 Evaluation Function

The evaluation function's main purpose is to gauge a hand's strength and probability of winning while taking into account the size of the pot and the bet amount, at a particular game state (Dittmar, 2008). This function consists of 6 different factors - a base value, pair value, flush value, straight value, pot value and a street value. The main aim of these 6 factors combined together is to replicate the consolidation of strategies of several professional poker players (Hilger, 2003; Sklansky, 2005).

Base Value

The base value was the main factor to be taken into account during preflop. This is mainly due to the fact that the other five factors do not play a significant role until after the community cards were revealed, i.e. flop and post-flop. Hence, the agent's hand strength had to be computed before any community cards are revealed. Hand strength refers to the probability that a given hand is stronger than that of the opposing player (Felix and Reis, 2008). The most intuitive way to implement this was to enumerate each card. A simple example is assigning the value 14 to an ace, since it has the highest strength. As such, both cards were assigned a value based on this criteria. With this method, it is more likely that the agent would raise premium hands with a higher likelihood of winning, call with decent hands that stood a good chance of winning post-flop and flop if dealt a marginal hand.

Pair Value

For the pair value computation, its initial value was initialized to 0. Subsequently, each community card shown would be compared with each hole card. If they were found to be the same, the pair value would go up by that card value, based on the same enumeration system utilized in the base value function. If they were not the same, the pair value would be deducted by that card value as well.

Flush Value

The flush value essentially represents the number of cards that have the same suit. Also, it was noted that for each street, a minimum number of suited cards must be available for a possibility for a flush. For example, during the flop, one would need at least 3 suited cards. If the flush value was less than this minimum number, then the flush value would be returned as -1, indicating a zero chance of achieving a flush in this round. Otherwise, the flush value would return as the number of suited cards. This was achieved by combining the hole cards and community cards into one list. If the number of cards with the same suit was greater than or equal to the minimum requirement, the flush value would return the number of those cards. This is a simplified form of calculating the probability of a flush (Chen and Ankenman, 2006), given the constraints of this project.

Straight Value

Similarly, the straight value was designed to count the number of consecutive cards. The hole cards and community cards were combined into a single list and sorted in ascending order. It is also important to take into account that an ace could be used in a wheel (A2345) as well as a broadway (AKQJT) (Sfetcu, 2014). Contrary to the other factors, the straight value started from a value 1. Each card would be compared to the card following it. If their difference was found to be 1, this meant a higher possibility of a straight and the straight value would increase by 1. Otherwise, the straight value would decrease by 1. As such, the greater the straight value, the higher the possibility of a straight.

Pot Value

The pot value is a factor designed in order for the agent to consider how much stake he has in the game. It is calculated by taking how much money the agent has in the pot, multiplied by a factor of -1 if the agent chooses to fold, else it will be multiplied by a factor of 1. It was constructed in this manner to simulate the increasing loss the agent will incur should he fold when he has more chips in the pot.

Street Value

One of the main challenges in designing an evaluation function is about balancing the myriad of factors involved. Surely, one could simply assign a weight to each constant within each factor and just train the agent. However, this comes at a huge cost as each additional weight results in an exponential increase in training time. Thus, instead of adjusting the constants within each factor which will consist of many different possible permutations, the street value factor was designed to act as a balancing mechanism. Throughout numerous testing phases, the agent was noticed to be incredibly passive and would fold very early on in a round. Thus, the street value factor assigns increasing constants to each street, with the river holding the highest value. This will then create a bias for the agent to be less inclined to fold in the early stages.

2.5 Training

Overview

The evaluation function takes in a weights parameter, which is an array consisting of six different weights, one for each factor. Combined with the minimax search algorithm, the evaluation function controls the actions of the agent. Thus, the values of the individual weights which heavily affect the output of the evaluation function essentially controls the agent as well. Therefore, there is a need to optimize the weight values to ensure that the agent makes an optimal decision at all times. This optimization is done by combining elements of reinforcement learning together with certain learning methods used in training neural networks.

Reinforcement learning is teaching the agent how to map particular situations to actions, in this case whether to fold, raise or call. This is done by not explicitly telling the agent what action it must take, but rather allowing the agent to take exploratory actions and reward them accordingly when the most optimal action is taken (Sutton and Barto, 1998).

Methodology

To begin the training process, the agent is first initialized with random weights, each ranging from zero to one. The agent is then made to play against another similar agent but with differently initialized weights. In the case of Limit Texas Hold'em, the rewards are not immediately available, but rather only known when the round comes to an end. As such, the weights are kept constant throughout each round and are not updated after each action. In addition, being a game with incomplete and imperfect information, there is a need to eliminate the factor of randomness before ascertaining that the actions taken by the agent is sub-optimal. This is done by running simulations of 500 rounds and looking at the resulting stack of each agent before concluding whether the agent chose the optimal set of actions.

In the event that the agent beats his opponent, the agent is "rewarded" in the sense that it will keep its same weights while the opponent will be re-initialized with a new random set of weights and they will be made to play each other again. Should the agent continue to win 100 times consecutively, it can be concluded that the agent has truly been making optimal decisions and that his victories over his training opponents were not a fluke.

In the event that the agent ties with his opponent, the agent will replay the exact same opponent again so as to eliminate the possibility of a coincidence. Should the agent tie with the same opponent three times consecutively, it can be concluded that the both sides are generally of equal strength. As such, the agent's weights will be re-initialized to take on the average values of their combined weights.

In the event that the agent loses to the opponent, the agent is "penalized" by having its weights changed by learning from the opponent it just lost to. This method of learning is inspired by a combination of the hill-climbing search (Russell and Norvig, 2016) and the stochastic gradient descent optimization method that is popularly used by many in training neural networks (Mitchell, 1997). One of the flaws of the hill-climbing search is that the search algorithm commonly gets stuck in a local maximum, thus making it difficult to find the global maximum. This issue is mitigated in two ways - firstly, whenever our agent wins against its opponent, the opponent's weights are re-randomized and are not the neighboring values of the agent's original weights. Thus, it is essentially impossible to be stuck at local maxima and the global maximum will always be found as long as the number of iterations of gameplay is huge enough. However, due to the practical limitations of the training time, the training is capped once the agent successfully beats 100 different opponents consecutively. In addition, to help improve the rate at which an optimal set of weights is found, a factor called learning rate is introduced. By setting the learning rate, n = 0.1, instead of replacing the weights of the agent with the weights of the opponent that it lost to, each of the agent's individual weight is updated as such:

$$w_a = w_a + n * |w_o - w_a| \quad (1)$$

where w_a is the agent's weight, w_o is the opponent's weight and n is the learning rate.

After updating the weights as such, the agent, with its newly updated weights, will replay the same opponent. If the agent keeps losing, this process will continue to repeat itself and the agent's weights will slowly approach that of the opponent. Only when the agent manages to beat the opponent, will the agent face off with a newly initialized opponent.

In comparison to simply replacing the entire set of weights with the opponent's weights, the introduction of the learning rate is superior because it provides a systematic and sequential approach in moving towards the global maximum as opposed to randomly initializing new weights while hoping to hit the global maximum.

Ultimately, as a safeguard, multiple agents with different weight values were trained using said method and were made to compete with each other. The one that won over all the other agents consistently was selected as the final agent.

3 Observation and Analysis

With a well-trained agent, expectations were that the poker agent will fare extremely well against any opposing agent. However, when our agent was tested against the standard agents, i.e. a raise-only player and a random-action player, its performance was decent but below expectations. The agent is able to consistently win against the random-action player but occasionally loses to the raise-only player. As such, attempts were made to modify the training procedure slightly. Instead of pitting two of our agents against each other, the agent was made to play against a raise-only player. Each time the agent loses, its weights will be re-initialized and the training will only stop once the agent beats the raise-only player consecutively 100 times. The same was done against the random-action player. The results are shown in Table 1.

Agent	flushValue	potValue
RaisedPlayer	0.125	0.966
RandomPlayer	0.707	0.394
MinimaxPlayer	0.008	0.829

Table 1: Final weights (correct to 3 decimal place) for flush and pot value obtained when training the agent against the raise-only player, random-action player and another of our agent respectively. Only factors that are significantly different are shown.

From Table 1, the weight corresponding to the pot value is much higher when the agent is being trained against the raise-only player at 0.966. Also, the weight for the flush value is much smaller when the agent is being trained against itself at 0.008. The conclusion is that there is no optimal set of weights that could be used against every opponent. Our agent's performance also depends on the behaviour of the agent it was playing against. This is a major issue since there is no way of knowing what sort of agent our agent would be facing.

4 Limitations

The minimax strategy has several limitations when used for a poker-playing agent, despite its effectiveness for several perfect-information games. When our agent competes against other complex agents employing different algorithms, the agent tends to fair poorly. In this section, the limitations of the minimax algorithm are reviewed and suggestions for future improvements are made.

4.1 Minimax Algorithm

Firstly, the main issue with the conventional minimax algorithm is that it does not perform well at games with imperfect information and uncertainty, such as poker (Heinrich and Silver, 2016). In order for the minimax algorithm to do well, it needs to consider multiple chance nodes, one for each possibility (Ballard, 1983). In order to do that, the branching factor will be enormous and there will be a need to expand the tree deeply (Gal and Avigal, 2010). Despite attempts to reduce the breadth of the search tree, with techniques such as bucketing and alpha-beta pruning, the tree is still too massive for the agent to search deeply within the time constraint of 200ms. The final implementation of our agent was to generate one chance node representing a specific random card combination, instead of generating all possible chance nodes. This was so that the agent could produce a decision within the time constraint. However, this implementation severely restricted the effectiveness of the minimax strategy, as the actual combination of cards revealed could be very different in effect to the random combination upon which the decision was made. The minimax strategy with numerous chance nodes could perhaps be better used in a case where there is a more lenient time constraint.

4.2 Evaluation Function

Secondly, a good evaluation function could take a long time to be built (Gal and Avigal, 2010). It is also possible for the heuristic to contain human errors due to a subjective understanding of the game (Gal and Avigal, 2010). The heuristic function essentially decides if a particular terminal node is favourable as an outcome. Thus, the success of the minimax algorithm could possibly hinge on how well the function is able to capture the state of the game.

According to Gal and Avigal, there are several ways neural networks can overcome the limitations of a minimax algorithm. These include using a neural network as a guide to decide which nodes to prune, or as a substitute for the heuristic function by teaching the agent which moves lead to victory and which lead to a defeat. We leave the exploration of neural networks to future works.

5 Conclusion

Despite being aware of the various evaluation methods within the poker engine, the team still decided to build most of the agent, inclusive of the evaluation function, from scratch in an attempt to deepen our knowledge and understanding of artificial intelligence and machine learning. This was made possible by combining elements of various tools learnt in class together with the extensive research that was done, ranging from other machine learning methods to various poker strategies employed by poker players. Ultimately, there is still much to be done and future work will most likely include strategies to tackle the issue of imperfect information and uncertainty within the game of poker.

References

Anton (1989). The reason for the benefits of minimax search. volume 1.

Ballard, B. W. (1983). The *-minimax search procedure for trees containing chance nodes. *Artificial Intelligence*, 21(3):327–350.

Chen, B. and Ankenman, J. (2006). *The Mathematics of Poker*. ConJelCo LLC.

Dittmar, P. (2008). *Practical Poker Math*. desLibris: Books collection. Ecw Press.

Felix, D. and Reis, L. P. (2008). In Zaverucha, G. and da Costa, A. L., editors, *Advances in Artificial Intelligence - SBIA 2008*, pages 83–92, Berlin, Heidelberg. Springer Berlin Heidelberg.

Gal, Y. and Avigal, M. (2010). Overcoming alpha-beta limitations using evolved artificial neural networks. *2010 Ninth International Conference on Machine Learning and Applications*.

Heinrich, J. and Silver, D. (2016). Deep reinforcement learning from self-play in imperfect-information games. *CoRR*, abs/1603.01121.

Hilger, M. (2003). *Internet Texas Hold'em: Winning Strategies from an Internet Pro*. Dimat Enterprises, Inc.

Mitchell, T. M. (1997). *Machine learning*. McGraw Hill.

Pedro, J. (2013). Reinforcement learning applied to the game of poker.

Russell, S. J. and Norvig, P. (2016). *Artificial intelligence: a modern approach*. Pearson.

Sfetcu, N. (2014). *Poker World*. Nicolae Sfetcu.

Sklansky, D. (2005). *The Theory of Poker*. Two Plus Two Publishing, 4th edition.

Sutton, R. S. and Barto, A. G. (1998). *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition.