Poker Project - Team 15

Gong Changda, Kong Zijin, Liu Yingnan, Liu Yiyang, Yang Sihan
National University of Singapore
{e0134064 ,e0253754, e0253726, e0253712, e0248120}@u.nus.edu

1 Introduction

In this project, we design and implement an AI-poker agent playing Limit Texas Hold'em, which is a two-player zero-sum game with incomplete and imperfect information. The agent's ability to act based on hand strength, to interpret opponents' actions and to learn from their strategies is important towards winning. To achieve this, reinforcement learning is used. Generally speaking, reinforcement learning is only capable of making models for the Markov decision process (MDP), while the poker game is defined as a partially observable Markov decision process (POMDP) with incomplete information. Considering this, we need to adjust the algorithm by selectively choosing "significant" factors to define a certain state.

Using reinforcement learning, our poker agent is expected to have the ability to design an optimal game strategy based on key information: hand strength (including hole cards and community cards), opponent's strategy and the expected reward. With these information, we built a heuristic evaluation function, which measures the expected reward for each action and makes the decision that maximizes the expected reward.

In this report, we will explain how we implement the poker agent and discuss how the agent can make an optimal decision. We will first review past research papers in section 2. Afterward, we will elaborate on our agent's strategy in section 3 and the training methods in section 4. Our overall training strategy can be categorized into two parts: offline training and online training. The offline training is used to estimate the overall hand strength. The online training is the main part of our reinforcement learning process, which includes training weights for every feature we select and learning from the opponent's strategy. In section 5 and 6, we will analyze the training results and discuss the advantages and limitations of our approach. Lastly, we will conclude this report and enlighten on future improvement.

2 Past Research

To implement an intelligent poker agent, one key point is to decide the player's next action at every game state based on the information obtained from the surrounding environment. Utility maximization is a popular approach that has been well explored and researched on. The most traditional method to optimize utility is to enumerate the results for all possible solutions in each round and select the action that gives the highest utility value [Tipton, 2012]. This method is precise and easy to implement but computationally expensive. To reduce the computational effort, some researchers proposed heuristics that evaluate the return of the game and used a decision tree to model the next action that should be taken [de Mesentier Silva *et al.*, 2018]. With sufficient and reasonable trainings, this method can yield good results.

Reinforcement learning is to learn how to map states to actions in order to maximize the reward. The learner in reinforcement learning is supposed to deduce the action that yields the highest reward after training. According to Sutton and Barto [1998], in the relatively challenging cases, actions affect not only the immediate reward but also the next and even all subsequent rewards. This enlightens the idea of our implementation. In the context of Limit Texas Hold'em, an intermediate action does not lead to an immediate reward. Thus, we calculate the expected reward at every step and adjust feature weights according to the difference between the expected reward and the final actual reward. Our implementation is based on value-function based learning. The heuristic function utilizes game state and action to estimate the expected reward.

3 Agent Strategy

The diagram(Figure 1) shows the player's strategy and the training process:

1. Our agent takes 18 card features into consideration to evaluate hand strength (can be found in `Group15.py`). In the online training part, the 18 features were combined into 8 features and incorporated into the feature vector.
2. When deciding how to act at a certain game state, our agent estimates the expected rewards of each action based on the linear evaluation model. The base weight vector is initialized by using the weight vector we obtained in the offline training.
3. The agent is supposed to choose the action with the highest expected reward among all valid actions.
4. After finishing the whole round of the game, our agent compares the actual reward with the expected rewards

calculated at every step and adjusts the weight vector accordingly.

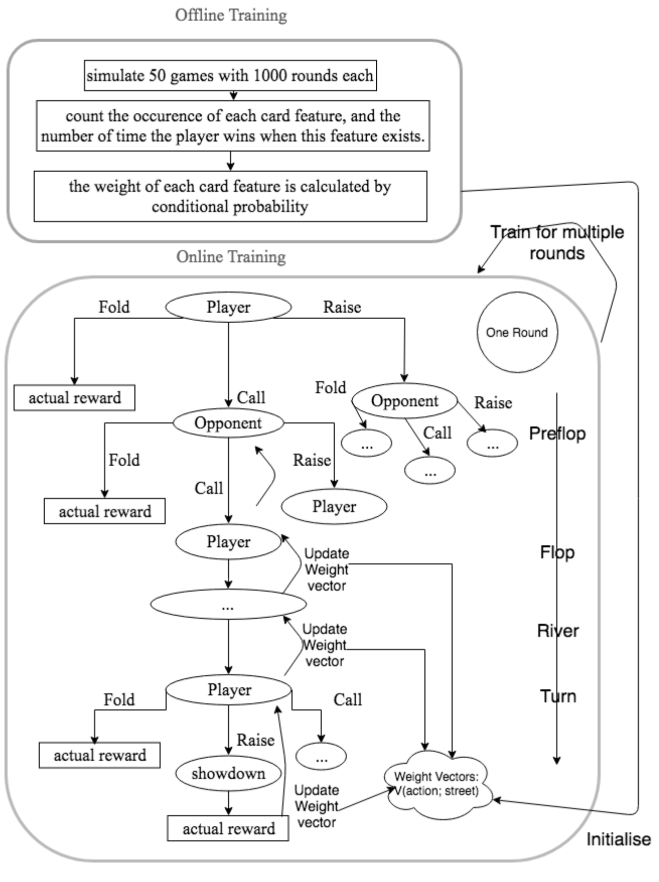
Offline Training
simulate 50 games with 1000 rounds each
count the occurence of each card feature, and the number of time the player wins when this feature exists.
the weight of each card feature is calculated by conditional probability
Train for multiple rounds
Online Training
One Round
Player
Fold
Raise
actual reward
Call
Opponent
Fold
Raise
Call
...
Preflop
Flop
River
Turn
Update Weight vector
showdown
Weight Vectors: V(action; street)
Initialise

Figure 1: Overview of strategy.

4 Training Method

Our training strategy includes two parts: offline training and online training. The offline training is used to train the weight vector of 18 card features, which will then be used to initialise the vector used in the online training part. Such evaluation only provides a general guide for our strategy, while the online training is targeted at adjusting the evaluation function according to the opponent's strategy. Only with online training, can we develop a systematic strategy to specify what action to take under the expected reward function. In addition, online training enables the agent to "learn" from the opponent dynamically. Our agent imitates the opponent's behavior if the opponent is performing well, and acts against the opponent if the opponent is not so clever. The following sections will discuss the two parts in details.

4.1 Offline Training

In the offline training part, we abstract the combination of hole cards and community cards into a series of card features and adjust the weight of each feature during the training process.
There are 18 card features in total considering both the rank and the suit of the card. The number of hole cards is fixed (2 hole cards) while the number of community cards shown is 0, 3, 4, 5 respectively in the street "preflop", "flop", "river" and "turn". Considering this, we construct and train different weight vectors with respect to different round states, in other words, different numbers of cards the player know.
We use $W_k =< w_{k1}, w_{k2}, ..., w_{k18} >$ to represent the weight vector in the k th street. With the string input representing the rank and suit of cards, we generate cards entity, fetch card features and transfer them to vectors accordingly. Then, we use the Monte Carlo algorithm to simulate 50,000 rounds (playing against the random player) to adjust the weight vectors by applying the theorem of conditional probability. The formula is defined as:

$$w_{ki} = Pr(q|s_i)$$

where q denotes the event that player wins and s_i denotes that player's hole cards together with community cards had the i th card feature.
Since 50,000 rounds of simulations are time-consuming, this part is done offline. The final weight vectors we obtained can be found in the array `street_1, street_2, street_3, street_4` in `Group15.py`, which will directly be used in the main training part – online training. There are some 0 weights, especially in the early streets. This is a reasonable outcome. For example, when in street 1 (preflop), all the 5 community cards have not been revealed yet and only 2 hole cards are known to the player, features such as "three of one kind" or "four of one kind" will be impossible to appear here and thus their weights evaluate to 0.

4.2 Online Training

With the pre-trained weight vectors of hand strength, we continue on the online training part. In this part, we use ϕ to denote the feature vector in a certain state, and θ as the weight vector that we need to train. Q and $\hat{Q}$ denote actual reward and expected reward function accordingly.

Features input to $\hat{Q}$

Input to $\hat{Q}$ consists of state and action. A certain game state mainly consists of four components: private cards, community cards, the player's bet and the opponent's bet.

Initially, we simply put the 4 components together with action ϕ in but find the result discouraging. After trial and error, the input ϕ to $\hat{Q}$ is defined as $< 1, ... >$, where 1 stands for the bias; the other 8 parameters are shown in table 1. This combination of features gives a high-quality result.

A Linear Model for $\hat{Q}$

We use a linear structure for $\hat{Q}$, and what we need to learn is a vector of weights, denoted as θ, of the same length of the feature vector ϕ. Then we evaluate the estimator $\hat{Q}$ for a particular ϕ with a formula:

$$\hat{Q}(\phi, \theta) = \sum_{1}^{n} \phi_i \cdot \theta_i$$

where n denotes the number of parameters. With the right choice of θ, this function can output a good estimate of the

f_1	high card value
f_2	hole card difference
f_3	if there is a value pair
f_4	quality of value pairs and same kinds
f_5	if there is a suit pair
f_6	quality of suit pairs and same kinds
f_7	if there is a sequence
f_8	quality of sequences

Table 1: features for online training

value of taking a particular action in a particular game state with a particular hand.
Considering the fact that action at different streets has a different level of influence on the final round result, the θ we learn is actually a list of 12 different vectors: $\theta(action, street)$, with certain action at a certain street, this function returns us a certain weight vector of the same length as the feature vector. There are 3 actions `(call, raise, fold)` and 4 possible streets `(preflop, flop, river, turn)` in total.

How to Act

At certain game state when current street is $street_i$, the player's actual action is decided by calculating $\hat{Q}$. We calculate $\hat{Q}$ with $\theta(raise, street_i)$, $\theta(call, street_i)$, $\theta(fold, street_i)$ and the feature vector ϕ. Then we choose the action with the highest expected reward $\hat{Q}$. Since it is not well-trained at the beginning, we incorporate a random factor ϵ here in order to make sure that all actions at all states at least be taken occasionally and to avoid misleading learning direction. With some small fraction ϵ of the time, the player will act randomly instead of choosing the action with the highest $\hat{Q}$. At first, we explore our options and frequently make a random choice. As more rounds are played, we expect our weight factor to be better-trained and thus act according to the highest $\hat{Q}$ more frequently.
All the player's actions, relative state features as well as $\hat{Q}$ will be stored in the action history.

Learning: Updating $\hat{Q}$

Since we can only obtain the actual reward after a round ends, θ is updated at the end of each round. Considering this, we need to store the action history of both the player and the opponents in order to track backward.
At the beginning of a game, we initialize the list of weight vectors θ by utilizing the hand strength we trained in the offline part. Stochastic gradient descent method is adopted to update θ. The formula is expressed as:

$$\theta \leftarrow \theta - \frac{\alpha}{2}\nabla_\theta L$$

where α denotes the learning rate and L denotes the loss function. We define the loss function L to be: $L = (R - E)^2$. Since the actual reward obtained at the end of each round is affected by a series of actions at different streets, we use a scale factor s to scale down the effect of a certain action as we trace down to the early streets.

```
scale = 1
scale *= scale_factor
delta = np.multiply(feature_vec,\
(actual_reward[opp_id]- expected_reward)\
* scale * self.learning_rate * self.opp_factor)
self.step_theta[step_idx][action] =\
np.add(self.step_theta[step_idx][action],delta)
```

Thus, the mathematics expression we used to adjust the value of θ for each action is

$$\theta \leftarrow \theta - s\frac{\alpha}{2}\nabla_\theta L$$

$$\theta \leftarrow \theta - s\frac{\alpha}{2}\nabla_\theta (R - \hat{Q}(\phi, \theta))^2$$

$$\theta \leftarrow \theta + s\alpha(R - \hat{Q}(\phi))\phi$$

Learn from an Opponent's Strategy

Learning from the opponent's strategy is an important part of the online training process. To fulfill this, we update the list of θ in the same way how we use the player's expected reward $\hat{Q}$ to update. The differences are:

1. When any of the players fold, we are unable to know the hole card of the opponent. To avoid introducing more noise rather than 'effective learning from strategy', we decide to only learn from the opponent when the hole cards of the opponent are revealed at the end of a round.
2. We incorporate an opponent learning factor to control 'how much' we will learn from the opponent.

```
delta = np.multiply(feature_vec,\
(actual_reward[opp_id]- expected_reward)\
* scale * self.learning_rate\
* self.opp_factor)
```

Adjust the Learning Rate and Scale Factor

Our initial learning rate is fixed at 0.01. However, later during the training process, we observed a strange phenomenon: At the first 500 rounds, the weight vector is well-trained and the player continues to win, however, after certain rounds, the player suddenly begins to lose and subsequently loses all his money. To avoid over-training and the appearance of similar phenomena, when the player achieves certain performance, we will decrease the learning rate and slow down the learning. With trial and error, we decide to fix the scale vector to 0.5 since it gives us the best performance.

5 Experiment

The performance of our agent is measured by the accumulative reward the player gains, where a stable increase should be observed to prove the agent well-trained.
With all the parameter we have settled based on these criteria, our agent's current performance is shown below. For every single experiment, our agent will play 15 games in total, where each game includes 1000 rounds with the initial stack of 10000. At the start of every experiment, the agent is initialized with only the off-line training data read-in. A scatter diagram will be created to depict the performance of our agent throughout the process of training, where every single point indicates the so far accumulative gain the agent has earned.

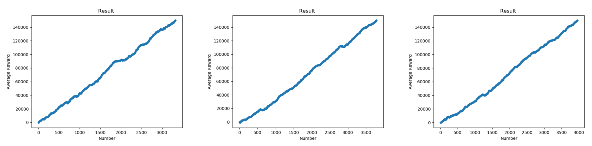

Figure 2: play with the random player

5.1 With the Random Player

When playing with the random player, our agent displayed an overwhelming performance, won nearly all money in the pots in relatively few rounds.

5.2 With the Raised Player

Though winning the raised player is not as easy as winning the random player, the agent guarantees its dominance in the game after training. Though there are much tottering

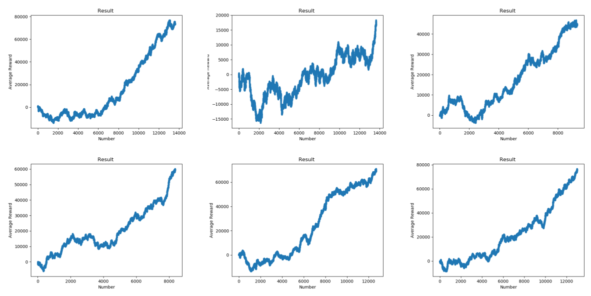

Figure 3: play with the raised player.

in the learning process, the agent's overall performance is improving as the slope of the curve becomes steeper.

5.3 With Myself

To test if our agent is a reasonable agent, we let our agent to play with itself. Since the two accumulative gain lines are

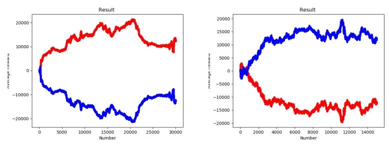

Figure 4: play with myself

converging, we can conclude that none of the players of our agent kind is at absolute advantage in the game.

6 Performance and Analysis

To learn efficiently from the game, we carefully chose features taken into consideration in online training. Rather than giving a more comprehensive observation of the game state, more features may result in more noise.

In the previous model, except for the 18 features describing the cards, we included the bets we have placed, money remains in the stack and the gain or lose obtained so far. These factors affect how human players behave, but seem to have little impact on machine players, especially when rounds are rather independent.

We further reduced the dimension of the feature space by combining some features. For example, we combined "three of a kind" and "four of a kind" into a single feature since they are very similar, rarely observed but suggest great confidence in a final positive reward once they appear. In this way, the dimension of the feature space decreased from 18 to 8, which not only makes the training process less computational intensive but also helps the model converge faster.

Below are some figures show the learning process of the agents without the above optimization. In these cases, the agents are not learning effectively due to the noise. Though they sometimes manage to learn the winning strategy, most of the time they just waver around different strategies and fail to figure out the correct path.

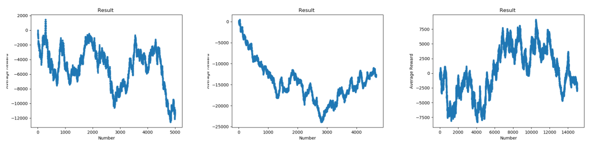

Figure 5: without optimization

Our strategy has two main advantages. Firstly, our approach is intuitive and it fits the characteristic of a poker game. The minimax algorithm is more suitable for games with perfect information. For an imperfect information game such as poker, the game tree used to conduct minimax search will use up a lot of space. Thus, instead of generating the game tree and minimax algorithm, we use the decision tree, which requires less memory and can simulate actions as well. Secondly, rather than throwing a large number of features in the model and take hours to train, we only extract 8 features from 18 card features that contribute most to the learning process. This makes our model more intelligent and understandable.

There are also some limitations in our model. Firstly, the hand strength is evaluated only based on card features of hole cards and currently revealed community cards. The potential of cards to become a strong hand in the future betting rounds is not taken into consideration. Furthermore, since our learning rate will be adjusted to be smaller when the current performance is good, in the situation when the opponent learns fast and updates their strategy fast, our learning process will fall behind. However, this limitation can be mitigated by increasing the frequency of updating the learning rate.

7 Conclusion

Through the above detailed elaboration, analysis and discussion, we can conclude that the enhanced reinforcement learning we use to implement an poker player applies to Limit Texas Hold'em games with incomplete and imperfect information. The implemented agent is able to estimate current state, make optimal decision to get maximum reward and adjust strategy according to opponent behavior.

References

[de Mesentier Silva *et al.*, 2018] Fernando de Mesentier Silva, Julian Togelius, Frank Lantz, and Andy Nealen. Generating beginner heuristics for simple texas hold'em. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 181–188. ACM, 2018.

[Sutton *et al.*, 1998] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[Tipton, 2012] Will Tipton. *Expert Heads up No Limit Holdem Play, Volume 1*. D & B Publishing, 2012.

[Tipton, 2017] Will Tipton. Playing a toy poker game with reinforcement learning, Jun 2017.