DeltaDou: Expert-level Doudizhu AI through Self-play

Qiqi Jiang, Kuangzheng Li, Boyao Du, Hao Chen and Hai Fang
SweetCode Inc, Beijing
{jiangqiqi, likuangzheng, duboyao, chenhao, fanghai}@itgwn.com

Abstract

Artificial Intelligence has seen several breakthroughs in two-player perfect information game. Nevertheless, Doudizhu, a three-player imperfect information game, is still quite challenging. In this paper, we present a Doudizhu AI by applying deep reinforcement learning from games of self-play. The algorithm combines an asymmetric Monte Carlo Tree Search (MCTS) on nodes representing each player’s information set, a policy-value network that approximates the policy and value on each decision node, and inference on unobserved hands of other players by given policy. Our results show that self-play can significantly improve the performance of our agent in this multi-agent imperfect information game. Even starting with a weak AI, our agent can achieve human expert level after days of self-play and training.

1 Introduction

Computer programs are able to beat the best human players in a lot of popular perfect information games, such as backgammon [Tesauro, 1995], checkers [Schaeffer et al., 2007], chess [Campbell et al., 2002], and go [Silver et al., 2016]. AlphaZero [Silver et al., 2018] introduces a general reinforcement learning framework, which is able to master chess, go, and shogi without human knowledge via self-play.


In this paper, we describe how to create an expert-level AI player for the game of Doudizhu by integrating two techniques into an AlphaZero-like self-play framework.

The first technique is an asymmetric Monte Carlo Tree Search (MCTS) on information set for each player, called Fictitious Play MCTS (FPMCTS). To alleviate state explosion problem, we construct a game tree of information set nodes for each player. The acting player chooses his action in a PUCT [Silver et al., 2016] fashion, and the other two players’ actions are sampled from their own policy networks. This approach avoids the issues of Strategy Fusion and Non-Locality. Moreover, FPMCTS converges very fast, since the branching factor is the size of the action space instead of the size of possible determinizations.

The second technique is a policy-based inference algorithm. [Buro et al., 2009] proposes a good inference algorithm that improves performance of their Skat AI drastically. We developed our own inference algorithm in our self-play framework, and found that policy network is a better fit than hand-crafted features in our case.

FPMCTS and the inference algorithm are integrated into a self-play framework. Each self-play episode takes all players’ neural networks as input. The search phase returns a vector of probability distribution over moves, and the resulting distribution along with the game results are then used to update the neural network of each player. In the next episode, this updated neural network used in inference and looking-forward search. To bootstrap the reinforcement learning procedure, we first generate a policy network from the self-play results of a heuristic AI.

Our algorithm is evaluated by two sets of experiments, with existing Doudizhu AI’s and human players respectively. We observe that our algorithm improved itself through self-play, and made significant progress after days of training with modest computational power. It dominates all other Doudizhu AI’s we are aware of, and shows a comparable strength against human experts. The playing style of our program looks more like human than other AIs.

The main contributions of this paper are:

- FPMCTS that enables policy iteration framework to work for multi-player imperfect-information games;
- Inference algorithm that takes advantage of players’ policies directly;
- AlphaZero-like deep reinforcement learning framework that combines neural networks with FPMCTS and the inference algorithm in a self-play procedure;
- An expert-level Doudizhu AI via self-play.

2 Related Works

Perfect Information Monte Carlo (PIMC) is used by the classical bridge algorithms like GIB [Ginsberg, 1999]. Despite
its success on imperfect information games [Long et al., 2010], PIMC still suffers from problems like Strategy Fusion and Non-Locality in reality.

MCTS is a popular search framework in perfect information games. In imperfect information games, the main challenge is that the asymmetry of information in an extensive form game makes constructing a single collective search tree problematic.

Several approaches are proposed to fix the problems caused by imperfect information. Information Set MCTS (ISMCTS) [Cowling et al., 2012] aimed to resolve the fusion problems. It grows a tree over information sets for each player instead of constructing a separate tree for each determination. ISMCTS suffered from the information leaking problem, as pointed out by [Furtak and Buro, 2013]. [Heinrich and Silver, 2015] introduced Self-play MCTS and used a separate search tree for each player. The tree was grown on each player’s own information state and was guaranteed to be a proper tree by perfect recall. Although each node in self-play MCTS contained an information set, the algorithm still had convergence problems. Smooth-UCT introduced a smoother action choosing scheme for better convergence property, but the algorithm might still need a lot of simulation steps to converge because of the iterative tree structure.

There are attempts to solve imperfect information games using deep reinforcement learning. [Heinrich et al., 2015] and [Heinrich and Silver, 2016] introduced Neural Fictitious Self-Play (NFSP). It calculated the best response for a given stationary policy and used a deep reinforcement learning approach to update that policy.

Doudizhu has not seen too many efficient solutions. [Powley et al., 2011] observed that games like Doudizhu with large branching factor were difficult for ISMCTS.

Our work combined the thoughts of self-play MCTS and NFSP. We proposed FPMCTS, a variant of MCTS which uses a neural network to store the policy. The best responses were computed via FPMCTS and the neural networks were updated in an AlphaZero-like framework. We implemented this idea and got a strong Doudizhu AI.

3 Background

3.1 Doudizhu

Doudizhu (a.k.a. Fight the Landlord) is one of the most popular card games in China. There are more than 800 million registered users and 40 million daily active players on the Tencent mobile platform for Doudizhu. The game is in the genre of climbing and shedding, and is described as easy to learn but hard to master.

The standard version of Doudizhu is played by three players with a pack of 54 cards including two jokers, red and black. The cards rank from high to low: Red Joker(B), Black Joker(L), 2, A, K, Q, J, 10(T), 9, 8, 7, 6, 5, 4, 3. Suits are irrelevant. The game begins with each of the three players being dealt 17 cards, with the remaining 3 cards dealt face-down.

The play of the game consists of two phases: bidding and cardplay. The bidding phase, which is not considered in this paper, designates one of the players as the landlord, which is analogous to the declarer in bridge. Unlike bridge, there are no permanent alliances of players cross games, and the two players who lose the bidding become partners (i.e. peasants) for the current game. We use letters to denote the positions: C for the landlord, D and E for the peasant on the right-hand side and left-hand side of the landlord respectively.

Once the landlord picks up the three face-down cards, card play begins. The game is played in rounds, and the landlord starts the first round. In any round, the first player may play any legal combination (single, pair, chain, etc.). Each subsequent player must either pass or beat the previous hand by playing a higher combination of the same hand category. There are just two exceptions: a rocket (a pair of jokers) can beat any combination, and a bomb (four cards of the same rank) can beat any combination except a higher bomb or rocket. The round continues until two consecutive players pass. When one round ends, the person playing the last cards in current round leads the next round.

The landlord wins the game if she gets rid of all cards on hand first; the peasants win otherwise. Each bomb or rocket played during the card play phase doubles the score for everyone. More thorough introductions to Doudizhu are available online at http://www.pagat.com/climbing/doudizhu.html.

3.2 Extensive-Form Games

Extensive-form games are a model of sequential interaction involving multiple agents. The representation is based on a game tree and consists of the following components: \( \mathcal{N} = \{1, 2, \ldots, n\} \) denotes the set of players. \( \mathcal{S} \) is a set of states corresponding to nodes in a finite rooted game tree. For each state node \( s \in \mathcal{S} \) the edges to its successor states define a set of actions \( \mathcal{A}(s) \) available to a player in state \( s \). The player function \( \rho : \mathcal{S} \rightarrow \mathcal{N} \) determines who is to act at a given state. For each player \( i \) who cannot observe the full state, there is a corresponding set of information states \( \mathcal{U}^i \) and an information function \( \mathcal{I}^i : \mathcal{S} \rightarrow \mathcal{U}^i \) that determines which states are indistinguishable for the player by mapping them on the same information set \( u \in \mathcal{U}^i \). Finally, \( R : \mathcal{S} \rightarrow \mathbb{R}^n \) maps terminal states to a vector whose components correspond to each player’s payoff. The reward vector sums to zero for zero-sum games. A policy \( \pi^i \) for player \( i \) maps each state \( s \) with \( \rho(s) = i \) to a probability distribution over \( \mathcal{A}(s) \).

For Doudizhu, each player can only observe his own cards \( c^i_j \) and action history \( H^i_j = a_0, a_1, \ldots, a_j \) at time \( j \). Therefore, the decision-making process in \( u^i \) is a single agent MDP since other players’ actions can be treated as feedback from the environment.

4 Extending AlphaZero to Multi-Player Imperfect-Information Game

In this section we describe the components introduced to make the AlphaZero-like training possible. The first component is FPMCTS, which is a variation of MCTS in an imperfect information game. The second component is policy-value networks that serve as opponents strategies. The third component is inference of other players’ cards. Last but not least, we describe how to adjust self-play in our case.
4.1 Fictitious Play MCTS
The computational complexity brought by updating other players’ choice recursively [Heinrich and Silver, 2015] motivated us to grow a separate search tree for each player. The fact that human players might choose pre-determined over rational actions [Ponsen et al., 2010] made us believe we could use policy network to approximate opponent responses. We call our algorithm Fictitious Play MCTS (FPMCTS). The term fictitious play [Brown, 1951] stands for our assumption that other players would always adhere to static but powerful strategies. We designed FPMCTS to make AlphaZero work for Doudizhu. [Ponsen et al., 2010] proposed a similar idea in the setting of Poker. These two approaches were developed independently for different purposes. Despite their similarities at the top-level, the two approaches have some differences: [Ponsen et al., 2010] used traditional MCTS with rollout and UCT. We use a value network to approximate the value of the leaf node, and p-UCT with a prior probability computed from a neural network for each action.

In the search tree for a specific player, there are two types of nodes: The decision nodes are the ones where the current player acts, and the rest are chance nodes. We merge the two opponents’ consecutive actions into one chance node. In the section below, we use $i$ to denote the current player.

At decision nodes, we use p-UCT to select actions:

$$\pi_{tree}(u^i) = \arg\max_{a \in A(u^i)} \left[ Q(u^i, a) + \pi'(u^i, a) \frac{\sqrt{N(u^i)}}{1 + N(u^i, a)} \right]$$

where $Q(u^i, a)$ is the average Q-value by choosing action $a$, $N(u^i)$ is the number of times $u^i$ has been visited, and $N(u^i, a)$ is the number of times $a$ has been taken. Values of decision nodes are estimated by a function $\pi'(u^i)$. To avoid the non-locality problem, we choose several deterministic worlds at the root node before the search begins by an inference algorithm, which we will describe in Section 4.3. During the expansion phase, traditional MCTS expands a node by one level, and FPMCTS expands a node by two levels so that new leaf nodes are decision nodes unless the game ends at a chance node at next level.

At chance nodes, other players’ moves are sampled from their policies. The network is updated after the self-play episode ends.

4.2 Policy-Value Network
Doudizhu has some specific hand categories that can make the action space very large. One example is Trio with single card, where any trio can be combined with an individual card as kicker, e.g.: 3334, 3335. This combination with kicker leads combinatorial explosion of the action space. To reduce the action space size, we only encode rank and category in the policy output. After the policy network produces answers, we use a kicker network and several heuristic rules to convert policy outputs to actual moves.

Since the neighboring relations between different card ranks are quite crucial in Doudizhu, we use 1D-convolutional layers. The network $f_\theta(s) = (p, v)$ contains 10 residual blocks with batch normalization and rectifier non-linearities. The parameters of $f_\theta$ are adjusted by the gradient of a loss function that sums the over mean-squared error and cross-entropy losses.

$$l = C_1(z - v)^2 - \pi^T \log p + C_2||\theta||_2$$

where $C_1$ is used to rescale the value error since $v$ is normalized, and $C_2$ is a parameter controlling the level of $L_2$ weight regularization.

The input of the network is a condensed $15 \times 7$ matrix of current information state, including current player’s cards, union of other players’ cards, recent three actions, and union of played cards. The output is a matrix of legal moves classified into $n$ separate buckets according to the hand categories.

Our input and output encoding can be viewed as a trade-off between precision and speed. In practice, the encoding has dramatically improved the training efficiency with few issues.

4.3 Inference
Players involved in imperfect information games can only observe a portion of the world, so making a good inference about the unseen portion can usually result in better action selections. A straightforward way to generate a deterministic world is random selection, but it fails to incorporate valuable knowledge from current game history. Therefore,
researchers have designed several ways to bias the cards’ distribution. Ponsen [Ponsen et al., 2010] has shown that the Bayesian Method equipped with a dependable policy regression model would be quite effective in Poker Games. Similarly, in Doudizhu, we take the output of policy-value network as a policy approximator to improve our inference.

Theoretically, the probability distribution for the real world could be computed precisely by the Bayesian Formula. We use $\mathcal{S}$ to denote all possible worlds of a specific information state. For any determinized situation $s \in \mathcal{S}$, its prior probability could be denoted as $p(s)$, and its history could be denoted by sequence $h(S) = h(s) = a_1, a_2, \cdots , a_k$. The policy could be denoted as $Pr(a|s) = Pr(a|I^p(s)(s))$. History situations $\{s_1, s_2, \cdots , s_k\}$ of this current situation $s$ would be derived by adding the played cards back. Therefore, the posterior probability $p(H|s)$ of the situation $s$ in the view of the current player could be computed as follows.

$$p(s|H) = \frac{p(s)p(H|s)}{\sum_{s' \in S} p(s')p(H|s')}$$

in which for any $H = a_1, a_2, \cdots , a_k$,

$$p(a_1, a_2, \cdots , a_k|s) = \prod_{s_j, p(s_j) \neq x} Pr(a_j|s_j)$$

However, in practice, there still exist several challenges [Buro et al., 2009]. First, calculating $p(s|H = h(s))$ for a large number of possible worlds is intractable. Second, a tight policy would make the prediction brittle in the face of players who do not play like us. Therefore, we designed a two-phase inference algorithm which can offer credible sampled worlds using modest hardware resources.

In Phase 1, a large number of deterministic worlds are randomly generated, then a small portion of the generated worlds would be retained based on filter scores. The scores are given by a 6-layer 1D-convolutional neural network trained from self-play games.

In Phase 2, the Bayesian method and the policy-value network are used to compute approximate posterior probabilities of samples generated in Phase 1. In detail, the policy $Pr(a|s)$ for a certain $s$ could be derived from $f_\vartheta(s) = (p(s), v)$ by applying a temperature operator. The temperature operator is used to make the inference algorithm more robust when the human opponents have different play styles.

Figure 1: DeltaDou’s search algorithm. This graph shows the D’s initial choice 3 was corrected by the FPMCTS to 44, a cooperating approach with his teammate E. This action led E whose cards was inferred as AA8 or AA9 with 99% confidence got rid of all his cards first, and thus, peasants won the game. Each node would be an information set, storing its value and visit number. Edges from hollowed nodes would be retained based on filter scores. The scores are given by a 6-layer 1D-convolutional neural network trained from self-play games.

Figure 2: Inference example. This scenario shows the change of peasant D’s guess on peasant E’s cards. After a round of play, the probability of peasant E having exactly two A’s increased from 40% to 93% from the point of view of D.
4.4 Self-play

Self-play is the key process to improve the performance of DeltaDou. Our procedure is similar to that of AlphaZero except that we maintain one network for each position (C, D, and E) respectively. We use randomly generated deck in each game for self-play. During self-play, we find that DeltaDou struggles to play correctly in some rare cases.

Strategies for each state during self-play and game results are stored in a reservoir that is used to train the policy-value network for each player. The algorithm then updates itself iteratively as what is done in AlphaZero.

5 Experiments

In this section, we first describe how to train the policy-value network in DeltaDou. Then we show that DeltaDou is much stronger than other existing programs and appears to have reached human expert strength using performance results.

5.1 Training Process

We used a hand-coded heuristic algorithm to bootstrap the training process. In the first phase, 200,000 games were self-played by the heuristic algorithm, then the game results were used to generate the initial policy-value network under supervised learning.

During self-play, each episode contains 8000 games, and FPMCTS contains 400 playouts. Inference is used when any player has fewer than 15 cards in hand. At the beginning of the self-play process, a high temperature is used to reduce the overfitting problem of policy, and the temperature cools down during self-play.

In order to check the progress of the self-play, we need to evaluate the performance of the agent. Our approach is inspired by teams tournament for duplicate bridge. When two teams A and B are compared with each other, the same hand is played twice: First, Team A plays as the landlord and Team B as the peasants, and then Team A plays as the peasants and Team B as the landlord. The following two metrics are used to compare the performances of the teams:

- **ADP (Average Difference in Points)**: The difference of the points scored per game of the two teams for the given hand set. For each game, the winning team gets 1 point and the losing team gets -1 point. Every bomb played during the game doubles the score.
- **WP (Winning Percentage)**: The fraction of games a team won, i.e. number of games won divided by number of games played. A team has a better performance than the opponent team if the former got a WP higher than 50%.

The performance gain of the learning process was measured against XDou, our strongest heuristic program. The learning curve of DeltaDou is shown in Figure 4.

5.2 Comparison to Doudizhu Baseline

We launched a tournament that include the two opponent side each play at landlord position and peasants position for 100 decks. WP and ADP will be used as playing strength metrics.

We are not aware of any AI competition where strong Doudizhu AI’s can compete with each other. Therefore, we chose two Doudizhu solvers to benchmark DeltaDou. The first one is RHCP, an open source heuristics-based algorithm available online (https://blog.csdn.net/sm9sun/article/details/70787814). It decides the hand to play purely by some handcrafted hand value estimation function. The second one is...
Table 1: Head-to-head tournament results between DeltaDou and baseline programs. DeltaDou and XDou are both much stronger than RHCP. In fact, XDou performed even better by exploiting the weak opponent more. However, DeltaDou showed statistically significant margin over XDou when they faced each other directly.

<table>
<thead>
<tr>
<th></th>
<th>WP</th>
<th>ADP</th>
</tr>
</thead>
<tbody>
<tr>
<td>XDou vs RHCP</td>
<td>79%</td>
<td>+0.88</td>
</tr>
<tr>
<td>DeltaDou vs RHCP</td>
<td>75%</td>
<td>+0.72</td>
</tr>
<tr>
<td>DeltaDou vs XDou</td>
<td>59%</td>
<td>+0.28</td>
</tr>
</tbody>
</table>

Table 2: DeltaDou vs Policy against baseline. The policy algorithm achieves comparable performance as XDou, and it is still left far behind by DeltaDou, which used both looking forward search and inference.

<table>
<thead>
<tr>
<th></th>
<th>WP</th>
<th>ADP</th>
</tr>
</thead>
<tbody>
<tr>
<td>RHCP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XDou</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>WP</th>
<th>ADP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Policy</td>
<td>58%</td>
<td>+0.19</td>
</tr>
<tr>
<td>DeltaDou</td>
<td>75%</td>
<td>+0.72</td>
</tr>
</tbody>
</table>

Table 3: DeltaDou’s tournament results vs human team. X-Y means X plays as the landlord and Y plays as the peasants. D and H denotes DeltaDou and the human team respectively. DeltaDou is stronger than human as the landlord (39% vs 36%, -0.44 vs -0.47) and is weaker than human as the peasants (61% vs 67%, 0.44 vs 0.61). The overall performance of DeltaDou is slightly worse than the human opponents, with 48.5% and -0.07 for WP and ADP respectively.

<table>
<thead>
<tr>
<th></th>
<th>WP</th>
<th>ADP</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeltaDou vs Human</td>
<td>48.5%</td>
<td>-0.07</td>
</tr>
</tbody>
</table>

Table 1: DeltaDou’s tournament results vs human team. X-Y means X plays as the landlord and Y plays as the peasants. D and H denotes DeltaDou and the human team respectively. DeltaDou is stronger than human as the landlord (39% vs 36%, -0.44 vs -0.47) and is weaker than human as the peasants (61% vs 67%, 0.44 vs 0.61). The overall performance of DeltaDou is slightly worse than the human opponents, with 48.5% and -0.07 for WP and ADP respectively.

6 Conclusion and Future Work

In this paper, we present our Doudizhu AI, DeltaDou, which dominates existing Doudizhu programs and plays roughly at human expert level. DeltaDou successfully integrates FPMC-CTS and inference algorithm into an AlphaZero-like framework. As far as we know, it is the first reinforcement learning algorithm that reaches expert level in a difficult multi-player imperfect information game with limited hardware resource and training time.

In future work, we would like to remove the heuristic-based neural network for bootstrapping so that the training process can start from scratch without any human knowledge. We have started experimenting with opponent modeling, which may help the AI to reach super-human level strength by exploiting the weakness of opponents more and by better understanding its partner. Finally, we will try to adapt our framework to other imperfect information card games such as bridge or skat.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments. Our colleagues, Guangxin Ren, Chang Wu, and Wen Xu, shared their know-how on the Doudizhu AI and contributed code libraries and other useful tools.
References


