WagerWin: An Efficient Reinforcement Learning Framework for Gambling Games

Haoli Wang, Hejun Wu, *Member, IEEE*, and Guoming Lai

***Abstract*—Although reinforcement learning (RL) has achieved great success in diverse scenarios, complex gambling games still pose great challenges for RL. Common deep RL methods have difficulties maintaining stability and efficiency in such games. By theoretical analysis, we find that the return distribution of a gambling game is an intrinsic factor of this problem. Such return distribution of gambling games is partitioned into two parts, depending on the win/lose outcome. These two parts represent the gain and loss. They repel each other because the player keeps "raising," i.e., making a wager. However, common deep RL methods directly approximate the expectation of the return, without considering the particularity of the distribution. This way causes a redundant loss term in the objective function and a subsequent high variance. In this work, we propose WagerWin, a new framework for gambling games. WagerWin introduces probability and value factorization to construct a more effective value function. Our framework removes the redundant loss term of the objective function in training. In addition, WagerWin supports customized policy adaptation, which can tune the pretrained policy for different inclinations. We conduct extensive experiments on *DouDizhu* and SmallDou, a reduced version of *DouDizhu*. The results demonstrate that WagerWin outperforms the original state-of-the-art RL model in both training efficiency and stability.**

***Index Terms*—Gambling games, game AI, reinforcement learning (RL).**

I. Introduction

REINFORCEMENT learning (RL) is a powerful approach to building game AIs. The fast-developing deep RL has led to great success in various types of games, e.g., *AlphaGo* [1] and *AlphaZero* [2] in board games; *AlphaStar* [3], *OpenAI Five* [4], and *JueWu* [5] in large video games. Recent RL studies have also made notable contributions to some gambling games, such as *Suphx* [6], *DouZero* [7], and *AlphaHoldem* [8]. We give a formal definition of our discussed gambling games in Section IV-A.

Manuscript received 14 June 2022; revised 8 October 2022; accepted 17 November 2022. Date of publication 5 December 2022; date of current version 15 September 2023. This work was supported in part by the National Natural Science Foundation of China under Grant 62272497, in part by the Science and Technology Program of Guangzhou, China, under Grant 202002020045, and in part by the Professorial and Doctoral Initiating Project of Huizhou University under Grant 2017JB005. *(Corresponding authors: Hejun Wu; Guoming Lai.)*

Haoli Wang and Hejun Wu are with the Department of Computer Science and Engineering, Sun Yat-Sen University, Guangzhou 510275, China (e-mail: wanghli6@mail3.sysu.edu.cn; wuhejun@mail.sysu.edu.cn).

Guoming Lai is with the School of Computer Science and Engineering, Huizhou University, Guangdong 516007, China (e-mail: laigm@hzu.edu.cn).

This article has supplementary material provided by the authors and color versions of one or more figures available at https://doi.org/10.1109/TG.2022.3226526.

Digital Object Identifier 10.1109/TG.2022.3226526

Despite the superhuman performance achieved by the above-mentioned game AIs, complex gambling games are still very challenging for RL. It is difficult for common deep RL methods to maintain stability and efficiency, especially in large imperfect-information games with complex rules. According to the work in [6], [7], and [8], special methods are necessary for promising results, including sophisticated modeling, massive data, strict constraints, and auxiliary techniques. Meanwhile, the intrinsic properties of gambling games have not drawn the researchers' attention. To our knowledge, this is the first work considering this issue.

In this work, by analyzing the return distribution in gambling games, we find a crucial problem that causes difficulties for deep RL. Our targeted gambling games are featured by the following points: First, for each game, the player ends up either winning or losing the wager. Second, the player can choose to "raise," so that they will win or lose more. Many popular games satisfy these features, such as *stud*, *hold'em*, *Mahjong*, and *DouDizhu*. There is only a single-terminal reward in every game episode for RL. Thus, the return distribution can be naturally partitioned into two disjoint parts depending on the win/lose outcome. These two parts repel each other when the player "raises." Besides, for gambling games, the winning and losing factors are not closely related. However, in common deep RL methods, the value function is directly targeted at the expectation of the return. The parametric value function is default updated by the sampled discounted accumulated rewards using mean-squared error (MSE) loss. This way causes a redundant loss term in the objective function and brings an extra high variance. Consequently, the training process is limited by low efficiency and instability. So, the model requires more training data or special techniques.

To solve this problem, we propose WagerWin, an efficient RL framework that introduces probability and value factorization. Inspired by the idea of distributional RL [9], [10], [11] and value function factorization [12], [13] in MARL, instead of estimating the single Q-value, WagerWin models learn the winning probability, losing Q-value, and winning Q-value, respectively. These three values compose the estimated Q-value for policies. We modify the training data so that each value is individually updated by different targets. This way, WagerWin removes the redundant loss term and makes the model more concentrated on valid information. WagerWin is then featured by a more stable and efficient training process. In addition, by learning these new values, WagerWin becomes more informative and enables more flexible policies. Our proposed customized policy adaptation

for value-based WagerWin algorithms can manually tune the pretrained policy for different inclinations. For example, we can make the policy focus more on win rates instead of average returns.

Our experiments on SmallDou show that WagerWin significantly improves the performance from the original RL method. SmallDou is a reduced version of *DouDizhu*. *DouDizhu* is one of the most popular card games in China and is a typical gambling game. We use *DouZero* [7], the state-of-the-art AI system on *DouDizhu*, as the baseline. We apply WagerWin to the original *DouZero* model and train both models using identical settings. The results show the WagerWin model outperforms the original by attaining the equivalent scores using less than 60% training data, and about 0.142 higher averaged scores than the original *DouZero* with the same 10^9 timesteps. We then conduct extensive experiments to evaluate WagerWin in various aspects, and the results are positive. We also directly compare WagerWin with the released pretrained *DouZero* policies. Ours achieves better performance using less than half of the training data and computation. The source codes of the environment and the algorithm have been released on GitHub.[1]

The article is structured as follows. Section II summarizes the representative AI studies on gambling games and return distributions. Section III introduces the background of our environments. Section IV analyzes the problem of common RL in gambling games. Section V describes the WagerWin framework and algorithm. Section VI shows the experiment results. Finaly, Section VII concludes the article.

II. Related Work

Game AI constantly serves as the representative of artificial intelligence, demonstrating the growth of AI through remarkable achievements in various environments. Recent studies have gradually evolved from relatively simple perfect-information games to more complicated imperfect-information multiplayer games. Breakthroughs have been made in some popular gambling games such as *Mahjong*, Texas *hold'em*, and *DouDizhu*.

One major approach to solving gambling games is searching. Counterfactual regret minimization (CFR) [14] is the leading searching algorithm that solves two-player zero-sum games. CFR iteratively minimizes the regrets of both players so that the strategies approximate a Nash equilibrium [15]. Many CFR variants are proposed to improve the computational efficiency. These CFR-based methods dominate the design of Texas *hold'em* AIs. Cepheus [16], [17] proposes CFR+, an enhanced CFR algorithm, and solves Heads-up limit *hold'em*. DeepStack [18] integrates CFR with deep neural network (DNN) to solve the more complex Heads-up no-limit *hold'em* (HUNL). Libratus [19], another superhuman HUNL AI, computes a blueprint strategy by Monte Carlo (MC) CFR [20]. Based on Libratus, Pluribus [21] further beats top professionals in six-player no-limit Texas *hold'em*.

RL is another major approach that is based on sampling. Recent studies show that deep RL methods can achieve competitive performance in many gambling games. Unlike CFR, RL methods require less computation and can be more easily generalized to complicated game environments. *Suphx* [6] demonstrates strong performance in *Mahjong*. *Suphx* uses a mixture of supervised learning (SL) and RL. During RL training, *Suphx* introduces global reward prediction and oracle guiding to boost efficiency. In *DouDizhu*, while the complex action space prohibits the use of DQN [22] and Actor-Critic [23] algorithms, *DouZero* [7] enhances traditional MC methods with DNN to tackle the environment effectively. In Texas *hold'em*, *AlphaHoldem* [8] adopts Proximal Policy Optimization (PPO) [24]. *AlphaHoldem* introduces the following crucial techniques to speed up the learning process: 1) the fine-designed information encoding and network architecture; 2) additional constrictions of the loss function; 3) novel self-play strategy evolution algorithm. With these techniques, *AlphaHoldem* successfully beats DeepStack [18] in HUNL with much fewer computing resources. Combining RL with search has also been studied in gambling games. Many studies follow the paradigm of *AlphaZero* [2], which implements MC tree search (MCTS) [25] with DNNs. For example, WRPM-MCTS [26] manages to apply this structure to *DouDizhu*. DeltaDou [27] models the hidden information with Bayesian inference and utilizes MCTS onto *DouDizhu*. *Suphx* can conduct parametric MC policy adaptation in run-time to fine tune the policies. Recently, ReBel [28] offers a general RL+search framework that converges to a Nash equilibrium in two-player zero-sum games. The structure of ReBel resembles *AlphaZero* while the state is altered to the public belief state.

Analyzing the distribution over returns is important in RL since statistics other than the expectation are also useful for some scenarios, such as risk management. However, the distribution has not been much considered in previous game AIs. Both theory and algorithms have been established for distributional RL. Bellemare et al. [9] introduce the distributional Bellman operator for policy evaluation. Their algorithm, C51, parameterize the return distribution as a categorical distribution over a fixed set of equidistant points and minimize the KL divergence to the projected distributional Bellman target. QR-DQN [10] proposed the use of quantile regression for distributional RL. It computes the return quantiles on fixed, uniform quantile fractions using quantile regression and minimizes the quantile Huber loss between the Bellman updated distribution and the current return distribution. IQN [11] proposes learning the quantile values for sampled quantile fractions rather than fixed ones with an implicit quantile value network that maps from quantile fractions to quantile values.

Our work, however, handles the return distribution differently from the algorithms mentioned above. The main reasons we do not adopt classic distributional RL algorithms are as follows. First, the exact value of the return/payoff is crucial to gambling games, but distributional RL only learns an approximation of the value, which may not be very accurate. Second, distributional RLs are often based on the DQN [22] framework while we do not presuppose the RL algorithms. Third, the reward mechanisms of gambling games are significantly characterized, so one need not consider the overall distribution of returns. Instead, learning the winning probability brings more advantages for game AIs.

[1] [Online]. Available: https://github.com/zmz2011/DouZero_Small_WagerWin

III. Background

A. Reinforcement Learning

Our framework is based on deep RL. We consider the Markov decision process (MDP), the standard problem model of RL. An MDP is defined by a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, where $\mathcal{S}$ is the *state space*; $\mathcal{A}$ the *action space*; $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$ denotes the transition probability from one state $s \in \mathcal{S}$ to the next state $s' \in \mathcal{S}$ given an action $a \in \mathcal{A}$; $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the *reward function* that determines the immediate reward received by the agent for a transition from (s, a) to s'; $\gamma \in [0, 1]$ is the *discount factor* that trades off the instantaneous and future rewards. At each timestep t, the agent observes the environmental state s_t and chooses to execute an action a_t. Then, the state transitions to $s_{t+1} \sim \mathcal{P}(\cdot|s_t, a_t)$ and returns the immediate reward $r_t = \mathcal{R}(s_t, a_t, s_{t+1})$ to the agent. The objective is to solve the MDP by finding a policy $\pi : \mathcal{S} \to \Delta(\mathcal{A})$, by which the expected discounted cumulative reward

$$\mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t \middle| a_t \sim \pi(\cdot|s_t), s_0\right] \tag{1}$$

is maximized during sequential actions. The *return* $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$ denotes the discounted cumulative reward at timestep t, where T is the terminal step of an episode. Based on (1), value functions are introduced to express the expected return over policy. The action-value function/Q-function is defined as $Q^\pi(s, a) = \mathbb{E}[R_t|s_t = s, a_t = a; \pi]$, and the state-value function is defined as $V^\pi(s) = \mathbb{E}[R_t|s_t = s; \pi]$. In deep RL, the value function is approximated by DNN.

B. DouDizhu *and* DouZero

We choose *DouDizhu* for our experiments and *DouZero* [7] as our base model. *DouDizhu* is a three-player card game that is extremely popular in China and is a typical gambling game. Among the three players, two of them are the Peasants. They need to cooperate as a team to compete against the other player named the Landlord. The game-playing consists of two phases: 1) bidding and 2) card-playing. The bidding phase designates the roles to the players and deals leftover cards to the Landlord. In the card-playing phase, the three players play cards in turn. Within a game episode, there are several rounds, and each begins with one player playing a legal combination of cards (solo, pair, etc.). The subsequent players must either choose to pass or beat the previous hand by playing a superior combination of cards. The round continues until two consecutive players choose to pass. Then, the player who played the last hand initiates the next round. The player wins by emptying his/her hand. Each Bomb and Rocket doubles the reward of one game, which can be regarded as the "raising" action. In this article, we only study the card-playing phase that can be formulated as an imperfect-information multiplayer game. A more detailed introduction about this game is provided in Appendix A (supplementary material). Readers may also refer to Wikipedia[2] for more information.

[2][Online]. Available: https://en.wikipedia.org/wiki/Dou_dizhu

DouZero [7] is regarded as the state-of-the-art AI of *DouDizhu*. *DouDizhu* is particularly challenging to solve. Its card-playing phase is a three-player multiagent imperfect-information game with both competition and cooperation. Meanwhile, the possible actions have the number of 27 472 and are hard to abstract according to the work in [7]. *DouZero* applies the Deep MC (DMC) algorithm, which generalizes MC with DNN for function approximation. MC methods are classic RL algorithms based on averaging sample returns [29]. DMC is a straightforward implementation of every-visit MC. Specifically, a Q-network is used to estimate $Q(s, a)$, where the observation encoding s and action encoding a are concatenated together as the input. The MSE loss is used to update the parameters. In every iteration, DMC samples a batch of episodes and optimizes its Q-network using all occurred instances (s, a, R). As there is only one nonzero reward at the terminal step of all episodes, DMC equals directly predicting the discounted final reward.

IV. Problem Formulation

A. Problem Model of Gambling Games

We begin with a formal description of our target game environments. Here, we do not presuppose the observability or the number of players. Although the multiplayer sequential-action game is supposed to be modeled as an extensive-form game (EFG) [30] theoretically, in this work, we do not focus on the solution in game theory. We assume the following discussions are based on common deep RL where the game environments are fixed. First, the dynamic policy of other players, if any, is omitted. Second, an encoding function is given to automatically convert the game information to the input state s for the RL model.

A gambling game possesses the following three properties.

1) Every game episode ends with a final reward r_T, and a specific win/lose outcome for the player. We denote the final outcome $\{\text{lose}, \text{win}\}$ as $u \in \{0, 1\}$. No nonzero reward occurs except r_T.
2) In every decision-making step, the range of r_T is constrained by two functions $R^{(w)}_{\min}(s)$ and $R^{(l)}_{\max}(s)$. If winning, then $r_T \geq R^{(w)}_{\min}(s) > 0$; if losing, then $r_T \leq R^{(l)}_{\max}(s) < 0$. $R^{(w)}_{\min}(s)$ and $R^{(l)}_{\max}(s)$ are derived from the game rules and can be inferred by the current information. For convenience, let $R^{(w)}$ and $R^{(l)}$ express r_T under different outcomes:

$$r_T = \begin{cases} R^{(w)}, & \text{if } u = 1 \\ R^{(l)}, & \text{if } u = 0. \end{cases} \tag{2}$$

3) The player can choose to "raise" by taking specific actions, named raising actions. By doing so, $R^{(w)}_{\min}(s)$ increases and $R^{(l)}_{\max}(s)$ decreases. In other words, the more the player wager, the more they will win or lose in the end. Raising actions cannot be withdrawn, so $R^{(w)}_{\min}(s)$ and $R^{(l)}_{\max}(s)$ are monotonic to the game episode. Raising actions are publicly observed.

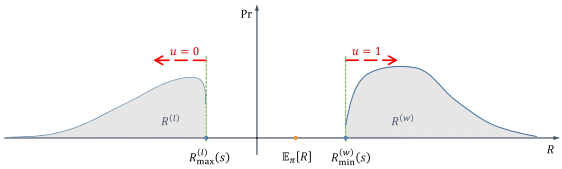

Fig. 1. Illustration of the return distribution $Z^\pi(s,a)$ in gambling games.

Most popular gambling games satisfy the above three properties, such as Texas *hold'em* and *DouDizhu*. Take Heads-up Texas *hold'em* as an example. If the player loses, it means they fold, or they have a smaller hand at the showdown. Then, $R^{(l)}$ is the amount they bet. Otherwise, the player wins, and $R^{(w)}$ is the amount the opponent bets (ignoring the cases of ties). In every step, $R^{(w)}_{\min}(s)$ is the current amount the opponent has bet, and $R^{(l)}_{\max}(s)$ is the current amount the player has bet. If the player calls or raises, then $R^{(l)}_{\max}(s)$ consequently becomes smaller. If the opponent calls or raises, then $R^{(w)}_{\min}(s)$ becomes larger. During the game, $R^{(l)}_{\max}$ does not increase, and $R^{(w)}_{\min}$ does not decrease, and the player can either win by not less than $R^{(w)}_{\min}(s)$ or lose by at least $R^{(l)}_{\max}(s)$.

The goal of RL is to maximize the expected returns $\mathbb{E}_\pi[R]$ in every game. Our targeted gambling games are naturally undiscounted, sparse reward, finite-horizon environments for common RL methods. Hence, the discount factor $\gamma = 1.0$ and $R_t = r_T$ for every steps t in the episode. We can omit the timestep subscript and use R to express both the final reward and the return hereafter.

The particularity of gambling games is that the return distribution always consists of two disjoint parts. The Q-value function for gambling games is $Q(s,a) = \mathbb{E}_\pi[R|s,a]$. We introduce the value distribution Z^π from distributional RL [9] to facilitate the formulation. Here, $Z^\pi(s,a)$ represents the distribution of the return R under s,a given $\mathcal{P}$ and π. An illustration of $Z^\pi(s,a)$ is shown in Fig. 1, assuming R is continuous for convenience. The left part is formed by $R^{(l)}$, and the right part is formed by $R^{(w)}$. These two parts are separated by the outcome u. There is no possibility that the return R lies within $(R^{(l)}_{\max}(s), R^{(w)}_{\min}(s))$.

B. Redundant Loss Term

We now expatiate the problem of the redundant loss term in gambling games. This problem derives from that the value function in common deep RL methods uses the expectation to generalize the return distribution with two separated parts. Although the ideal deep RL model can theoretically approximate the expectation of arbitrary distributions using MSE loss given sufficient sampled data, in practice, this way is inappropriate in gambling games. As shown in Fig. 1, $R^{(l)}_{\max}(s)$ and $R^{(w)}_{\min}(s)$ create a gap between the distribution of $R^{(l)}$ and $R^{(w)}$. Every time a raising action is taken, the two parts repel each other, and the gap gets widened. As a result, the sampled returns $R \sim Z^\pi(s,a)$ tend to get away from $\mathbb{E}_\pi[R|s,a]$, which is the target of the value function. In brief, the training loss automatically increases as the player takes raising actions. Moreover, different raising amounts create different gaps, and different gaps produce different scales of losses. This problem is disadvantageous for deep RL.

We give an analytical demonstration of the above-mentioned problem. Assume that the environment $\mathcal{P}$ is fixed. Let p^π denote the winning probability of the game under policy π, $p^\pi(s,a) = \Pr\{R > 0|s,a;\pi\} = \int_0^{+\infty} Z^\pi(s,a)dR$. The expectation of the return can then be restructured as $\mathbb{E}_\pi[R|s,a] = p^\pi(s,a)\mathbb{E}_\pi[R^{(w)}|s,a] + (1-p^\pi(s,a))\mathbb{E}_\pi[R^{(l)}|s,a]$. Given an MC Q-value model $Q^\theta : \mathcal{S}\times\mathcal{A}\to\mathbb{R}$ with parameters θ; a batch of n instances sampled under π for training: $\mathcal{D} = \{(s_1,a_1,R_1),\ldots,(s_n,a_n,R_n)\}$, where all $R_i \sim Z^\pi(s_i,a_i)$. Now the expectation of the Q-value loss can be calculated as

$$\begin{aligned}\mathbb{E}_{\pi,\mathcal{D}}[\mathcal{L}_Q] &= \mathbb{E}_\pi\left[\frac{1}{n}\sum_{i=1}^n \left(Q^\theta(s_i,a_i) - R_i\right)^2 \middle| R_i \sim Z^\pi(s_i,a_i)\right] \\ &= \frac{1}{n}\sum_{i=1}^n \left(Q^\theta(s_i,a_i) - \mathbb{E}_{R_i\sim Z^\pi(s_i,a_i)}[R_i]\right)^2 \\ &\quad + \frac{1}{n}\sum_{i=1}^n \mathrm{Var}_{R_i\sim Z^\pi(s_i,a_i)}(R_i) \end{aligned} \tag{3}$$

which consists of both the bias and the variance. For simplicity, let $\mathbb{E}_Z[R_i]$ denote $\mathbb{E}_{R_i\sim Z^\pi(s_i,a_i)}[R_i]$, $\mathrm{Var}_Z(R_i)$ denote $\mathrm{Var}_{R_i\sim Z^\pi(s_i,a_i)}(R_i)$, and p_i^π denote $p^\pi(s_i,a_i)$. Then

$$\begin{aligned}\mathrm{Var}_Z(R_i) &= \mathbb{E}_Z\left[R_i^2\right] - \mathbb{E}_Z^2[R_i] \\ &= p_i^\pi \mathbb{E}_Z\left[\left(R_i^{(w)}\right)^2\right] + (1-p_i^\pi)\mathbb{E}_Z\left[\left(R_i^{(l)}\right)^2\right] \\ &\quad + \left(p_i^\pi \mathbb{E}_Z\left[R_i^{(w)}\right] + (1-p_i^\pi)\mathbb{E}_Z\left[R_i^{(l)}\right]\right)^2 \\ &= p_i^\pi \mathrm{Var}_Z\left(R_i^{(w)}\right) + (1-p_i^\pi)\mathrm{Var}_Z\left(R_i^{(l)}\right) \\ &\quad + p_i^\pi(1-p_i^\pi)\left(\mathbb{E}_Z\left[R_i^{(w)}\right] - \mathbb{E}_Z\left[R_i^{(l)}\right]\right)^2.\end{aligned}$$

Therefore

$$\begin{aligned}\mathbb{E}_{\pi,\mathcal{D}}[\mathcal{L}_Q] &= \frac{1}{n}\sum_{i=1}^n \left(Q^\theta(s_i,a_i) - \mathbb{E}_Z[R_i]\right)^2 \\ &\quad + \frac{1}{n}\sum_{i=1}^n \left(p_i^\pi \mathrm{Var}_Z(R_i^{(w)}) + (1-p_i^\pi)\mathrm{Var}_Z(R_i^{(l)})\right) \\ &\quad + \frac{1}{n}\sum_{i=1}^n p_i^\pi(1-p_i^\pi)\left(\mathbb{E}_Z\left[R_i^{(w)}\right] - \mathbb{E}_Z\left[R_i^{(l)}\right]\right)^2.\end{aligned} \tag{4}$$

Let

$$T_1 = \frac{1}{n}\sum_{i=1}^n p_i^\pi(1-p_i^\pi)\left(\mathbb{E}_Z\left[R_i^{(w)}\right] - \mathbb{E}_Z\left[R_i^{(l)}\right]\right)^2$$

then obviously $\mathbb{E}_{\pi,\mathcal{D}}[\mathcal{L}_Q] \geq T_1$. Although in (4), p_i^π, $\mathrm{Var}_Z(R_i^{(w)})$ and $\mathrm{Var}_Z(R_i^{(l)})$ are intrinsic, T_1 is the redundant loss term. Because it is irrelevant to the target and is caused by the gap between $R^{(w)}$ and $R^{(l)}$, rather than the randomness

of the environment or policy. However, T_1 is usually of significance. Let

$$T_2 = \frac{1}{n}\sum_{i=1}^{n} p_i^{\pi}(1-p_i^{\pi})\left(R_{\min}^{(w)}(s_i) - R_{\max}^{(l)}(s_i)\right)^2$$

then T_2 is the lower bound of T_1, and T_2 strongly depends on the raising amounts. For example, in *DouDizhu*, if one Bomb is played now, then the final reward doubles. So $\mathbb{E}_Z[R^{(w)}] \geq R_{\min}^{(w)}(s) = 2$, $\mathbb{E}_Z[R^{(l)}] \leq R_{\max}^{(l)}(s) = -2$. Suppose the winning probability $p^{\pi} = 0.7$, then by T_2, the data points from this state have $T_1 \geq 0.7 \times 0.3 \times (2-(-2))^2 = 3.36$. Similarly, when two Bombs are played, the redundant loss term rises to 13.44, which can consume most of the loss and the gradient.

Term T_1 exaggerates the training loss and worsens the high-variance problem in deep RL. Moreover, raising actions like Bombs in *DouDizhu* is crucial for gambling games. This significant redundant loss term from value function is a distracter for efficient learning of valid information. It brings extra high variance, which makes the training process relatively inefficient and unstable. Besides the numerical value, (4) also indicates that both win rate and raising amount strongly affect the RL training in gambling games.

V. WagerWin Framework

A. Architecture

To solve the problem of the redundant loss term, we need to remove the correlation between the two separated parts $R^{(w)}$ and $R^{(l)}$ in the distribution of R. Instead of estimating the single Q-value $Q(s,a) = \mathbb{E}_{\pi}[R|s,a]$, our proposed framework for gambling games explicitly learns three new values: 1) the winning probability $p^{\pi}(s,a) = \mathbb{E}[u|s,a;\pi]$, 2) the winning Q-value $Q^{(w)}(s,a) = \mathbb{E}[R^{(w)}|s,a;\pi]$, 3) and the losing Q-value $Q^{(l)}(s,a) = \mathbb{E}[R^{(l)}|s,a;\pi]$. Our estimated Q-value $\hat{Q}^{\pi}$ for the policy is then calculated as $\hat{Q}^{\pi}(s,a) = p^{\pi}(s,a)Q^{(w)}(s,a) + (1-p^{\pi}(s,a))Q^{(l)}(s,a)$.

To update the three new values, we do not calculate the error between $\hat{Q}^{\pi}$ and R. Instead, we restructure the training data and the loss functions to directly optimize each value. The original minibatch training data $\mathcal{D}$ are partitioned into winning/losing subsets according to u: $\mathcal{D} = \mathcal{D}^{(l)} \cup \mathcal{D}^{(w)}$, where $\mathcal{D}^{(l)} = \{(s,a,R^{(l)},u)|u=0\}$ and $\mathcal{D}^{(w)} = \{(s,a,R^{(w)},u)|u=1\}$. $Q^{(w)}$ is trained by $\mathcal{D}^{(w)}$ and $Q^{(l)}$ is trained by $\mathcal{D}^{(l)}$ using MSE loss. In other words, $Q^{(w)}$ approximates the right part and $Q^{(l)}$ approximates the left part of Z^{π} in Fig. 1. p^{π} is trained by $\{u\}$ using binary-cross-entropy (BCE) loss. The loss function of WagerWin

$$\mathcal{L}_{\mathrm{W}} = \alpha\mathcal{L}_p + \beta\mathcal{L}_{Q'} \tag{5}$$

where α and β are hyperparameters that control the weights. For $\mathcal{D}$ with n instances

$$\begin{aligned}\mathcal{L}_p &= \mathrm{BCE}_{\mathcal{D}}\left(p^{\pi}(s,a),u\right)\\ &= -\frac{1}{n}\sum_{i=1}^{n}\left(u_i\log\left(p^{\pi}(s_i,a_i)\right)\right.\\ &\quad\left.+(1-u_i)\log(1-p^{\pi}(s_i,a_i))\right)\end{aligned} \tag{6}$$

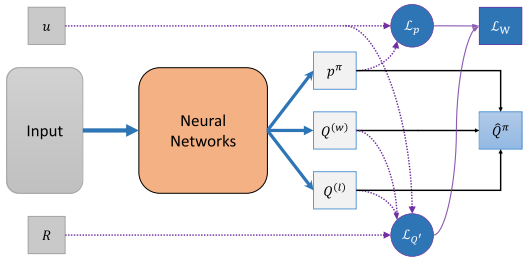

Fig. 2. Architecture of the WagerWin framework.

and

$$\begin{aligned}\mathcal{L}_{Q'} &= \frac{|\mathcal{D}^{(w)}|}{n}\mathrm{MSE}_{\mathcal{D}^{(w)}}\left(Q^{(w)}(s,a),R^{(w)}\right)\\ &\quad+\frac{|\mathcal{D}^{(l)}|}{n}\mathrm{MSE}_{\mathcal{D}^{(l)}}\left(Q^{(l)}(s,a),R^{(l)}\right) \qquad (7a)\\ &= \frac{1}{n}\sum_{i=1}^{n}\left(u_i\cdot\left(Q^{(w)}(s_i,a_i)-R_i\right)^2\right.\\ &\quad\left.+(1-u_i)\cdot\left(Q^{(l)}(s_i,a_i)-R_i\right)^2\right). \qquad (7b)\end{aligned}$$

In practice, we use u to mask every instances for $\mathcal{L}_{Q'}$, as shown in (7b). We name this framework WagerWin, as it improves the AI performance to win more wagers. Fig. 2 shows the overall architecture of the WagerWin framework. The architecture of the neural networks has no restrictions. For simplicity, we omit s and a hereafter. $\hat{Q}^{\pi} = p^{\pi}Q^{(w)} + (1-p^{\pi})Q^{(l)}$ is for rollout policy, but $\hat{Q}^{\pi}$ is not involved in training. WagerWin has two main advantages for gambling games. First, it removes the redundant loss term T_1 and the extra variance in common value functions. The separation between $R^{(l)}$ and $R^{(w)}$ does not affect the training process. p^{π}, $Q^{(l)}$, and $Q^{(w)}$ independently focus on $\mathbb{E}_{\pi}[u]$, $\mathbb{E}_{\pi}[R^{(l)}]$, and $\mathbb{E}_{\pi}[R^{(w)}]$. This strengthens the stability and efficiency during the network training. Second, it constructs explicit decoupled information by winning probability, which makes our model more informative. This enables customized policy adaptation in Section V-B.

To factorize the Q-value by the winning probability is empirically reasonable. In gambling games, the winning factors are often intrinsic. The player needs to take good actions based on the current condition to win a game. While the losing factors can be extrinsic, which means losing a game may not be due to the policy. Even an optimal policy in gambling games cannot guarantee winning in many cases. So, the winning Q-value and the losing Q-value measure two distinguishable informations, and the winning probability synthesizes winning and losing factors.

Our framework does not specify the RL algorithm as long as there is a DNN value function approximator. Since the problem of the redundant loss term also occurs in state value function $V(s)$, the criticism in policy-based RL methods [23] can also theoretically be combined with WagerWin. In this article, we apply WagerWin to DMC in *DouZero* [7] to form a *WagerWin-DMC* algorithm, which is used in our experiments in Section VI.

The pseudocode of WagerWin-DMC is given in Appendix B (supplementary material).

B. Customized Policy Adaptation

Expert players can often change their strategies in different situations. For example, they can actively choose to be more aggressive or conservative in special cases. However, common RL approaches can only get a static policy. Its policy is fixed after the training. This feature may become a weakness as the policy lacks flexibility and may be exploited when the opponents get familiar with it. Some methods are proposed for this problem. For example, *Suphx* [6] employs simple rules to decide whether to declare a winning hand at the different stage of a tournament. *Suphx* also proposes run-time policy adaptation that fine tunes its policy for the current game based on simulations. However, this process is very time consuming and thus cannot be applied to online testing.

By learning the additional information, it is easy for the WagerWin framework to generate adapted policies. Our proposed customized policy adaptation, by manually adapting some hyperparameters, can tune the rollout policy based on the winning probability. We introduce four hyperparameters, $a^{(w)}, b^{(w)}, a^{(l)}$, and $b^{(l)}$ to linearly control $Q^{(w)}$ and $Q^{(l)}$, making the adapted Q-value

$$\hat{Q}^{\pi'} = p^{\pi}\left(a^{(w)}Q^{(w)} + b^{(w)}\right) + (1-p^{\pi})\left(a^{(l)}Q^{(l)} + b^{(l)}\right). \tag{8}$$

By default, $(a^{(w)}, b^{(w)}, a^{(l)}, b^{(l)}) = (1, 0, 1, 0)$, and $\hat{Q}^{\pi'}$ is the same as $\hat{Q}^{\pi}$. Set $(a^{(w)}, b^{(w)}, a^{(l)}, b^{(l)}) = (0, 1, 0, 1)$, then $\hat{Q}^{\pi'} = 2p^{\pi} - 1$. Now, the adapted policy is forced to only care about the win rate instead of the final reward. We will discuss this adaptation in Section VI-C. To further tune these four hyperparameters, we can get various inclinations of policies, such as more aggressive or conservative ones. For example, $(a^{(w)}, b^{(w)}, a^{(l)}, b^{(l)}) = (1, 0, 0, R^{(l)}_{\max})$, then the adapted policy only considers the minimum losses if losing.

Another practice is to constrain Q-values by current $R^{(w)}_{\min}(s)$ and $R^{(l)}_{\max}(s)$ in decision-making stage. We let

$$\hat{Q}^{\pi} = p^{\pi}\max\left(R^{(w)}_{\min}, Q^{(w)}\right) + (1-p^{\pi})\min\left(R^{(l)}_{\max}, Q^{(l)}\right). \tag{9}$$

For example, in *DouDizhu*, the boundaries of the final reward are defined by the Bomb number, which is predictable at run-time. This practice is helpful in early training as it revises $Q^{(w)}$ and $Q^{(l)}$ when the randomly initialized DNN is not yet stable.

VI. Experiments

We use *DouDizhu* to test our framework. We straightforwardly implement WagerWin to the original model of *DouZero* [7] project, forming the WagerWin-DMC algorithm. We conduct comprehensive experiments to investigate the effectiveness of WagerWin in the following sections. More experiments are provided in the appendix (supplementary material).

A. Experimental Setups

Due to the limitation of our time and resources, most of our experiments and analyses are on a reduced version of *DouDizhu*, named SmallDou. Specifically, we delete the card with rank 2−5 so that the deck has 38 cards now. In the SmallDou environment, the Landlord has 14 cards and the Peasants have 12 cards. We then delete two rare kinds of actions (Plane with Pairs and Quad with two Pairs) to limit the number of cards in a single action. The new environment preserves the characteristic of *DouDizhu* and is still challenging. We also train and compare our framework on the original *DouDizhu* environment in Section VI-E.

In SmallDou, we compare the following methods.

1) *Original DMC:* The original DMC algorithm from the *DouZero* project. We use the open-sourced code[3] and keep almost all the settings unchanged.
2) *WagerWin-DMC:* We implement the WagerWin framework to the original DMC model. All hyperparameters remain the same with Original DMC for a fair comparison. The implementation details are provided later.
3) *RLCard:* A simple rule-based policy in RLCard package[4] [31], which serves as a weak baseline.

Our focus is to study how the WagerWin framework improves the performance from the original deep RL method with the minimum adjustments.

Metrics: Following the work in [27] and [7], there are two metrics to measure the performance of *DouDizhu* AI programs. Given two algorithms A and B, one can evaluate the following.

1) WP (Winning Percentage): The number of the games wins by A divided by the total number of games played between A and B.
2) ADP (Average Difference in Points): The average difference of points scored per game in games of A against B. In each game, the base score is 1; every Bomb and Rocket doubles the score.

Generally, WP measures the win rate, and ADP measures the expected payoff. ADP is a more accepted metric. *DouZero* sets different objectives to let its DMC models be oriented to WP or ADP metrics, respectively. Under the WP objective, the agent is given a $+1/-1$ reward at the terminal step based on u in each game. Under the ADP objective, the agent receives the actual point it scores. WagerWin-DMC does not use the WP objective because under the WP objective $R^{(w)} = 1$ and $R^{(l)} = -1$ constantly, which makes $Q^{(w)}$ and $Q^{(l)}$ unnecessary.

Specifically, we train the following agents in SmallDou: WagerWin-DMC under the ADP objective for 1e9 timesteps (*WD 1e9*), Original DMC under the ADP objective for 1e9 timesteps (*OD-ADP 1e9*), and Original DMC under the WP objective for 1e9 timesteps (*OD-WP 1e9*). We save all trained models every 5e7 timesteps for evaluation. Following the work in [7], we skip the bidding phase and randomly sample 10 000 decks of SmallDou and use them to conduct all the evaluations. We adopt round-robin tournaments where every model competes with all others as the Landlord and the Peasants, respectively.

[3][Online]. Available: https://github.com/kwai/DouZero
[4][Online]. Available: https://github.com/datamllab/rlcard

TABLE I
PROGRAM SETTINGS AND HYPERPARAMETERS

Learning rate ψ	1e-4
Optimizer	RMSProp
Max norm of gradients	40.0
Discount factor γ	1.0
Number of actors	8
Number of learner	1
Batch size	3072
Exploration ratio of ϵ-greedy	0.01
LSTM hidden sizes	[128]
MLP hidden sizes	[512, 512, 512, 512, 512]
Activation function	ReLU
Sequence length for LSTM	12

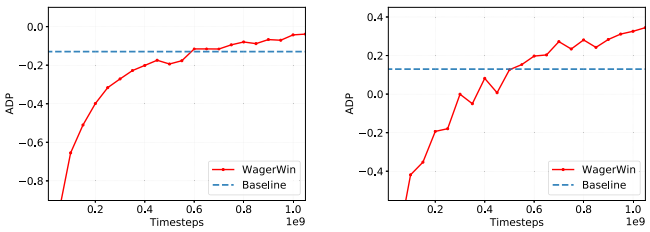

Fig. 3. Performances of WagerWin-DMC models against OD-ADP 1e9 as the baseline w.r.t. timesteps. (a) WD Landlord versus OD-ADP 1e9 Peasants (Baseline). (b) WD Peasants versus OD-ADP 1e9 Landlord (Baseline).

Implementation Details: We complete all experiments on a small server with an Intel Core i9-10900X CPU and two NVIDIA RTX 3090 GPUs. Each model training in SmallDou uses only a multicore CPU and a single GPU, which is reproducible for even personal computers. The DNN architecture consists of a one-layer LSTM and a six-layer MLP. For both Original DMC and WagerWin-DMC, the program settings and hyperparameters are all identical, listed in Table I. Compared to the source code of *DouZero* project, our implementation changes the batch size from 3200 to 3072 and the sequence length from 15 to 12 because SmallDou has a smaller episode length than *DouDizhu*. For the WagerWin-DMC model, the additional hyperparameters of the loss function are $\alpha = 1, \beta = 0.5$. The training speeds for Original DMC and WagerWin-DMC models on our server are both roughly 2.5e8 timesteps per day.

B. Performance Evaluation

To evaluate the effectiveness of our proposed framework, we record the performance of WagerWin-DMC w.r.t. training timesteps against OD-ADP 1e9 as the baseline. Fig. 3 shows the results. The dotted line is the score that OD-ADP 1e9 plays against itself. We observe that WagerWin-DMC achieves better performance than Original DMC at the same training timesteps (1e9). The improvement for the Peasants is more significant. To be exact, WD 1e9 Landlord and WD 1e9 Peasants respectively achieve 0.087 and 0.196 higher ADP than OD-ADP 1e9. On average, our models bring about 0.142 more scores in every random game. Furthermore, WagerWin-DMC attains the same scores as Original DMC with significantly less training data. WD 5.5e8 Peasants already have higher scores than OD-ADP 1e9 Peasants while WD 6e8 Landlord has higher scores than OD-ADP 1e9 Landlord. It indicates that WagerWin-DMC gets equivalent performance using less than 60% of sampled data. To conclude, the WagerWin framework learns more efficiently than common RL in gambling games and ends up with better performance.

We also summarize the ADP results of head-to-head completions among some representative models in Table II. WD 1e9 ranked the first model by outperforming all others. Both WD models surpass OD-ADP models with the same training samples (shown with underlines). Moreover, WD 6e8 outperforms OD-ADP 1e9 and OD-WP 1e9, proving that WagerWin-DMC can get equivalent performance using less than 60% training data. An interesting point is that OD-ADP 1e9 does not have much advantage against OD-WP 1e9, and OD-ADP 6e8 is even weaker than OD-WP 1e9. This may be because, under the WP objective, there are no raising actions. The problem of the redundant loss term does not occur. OD-WP manages to learn more efficiently and beats inefficient OD-ADP using WP-based policy.

C. Results of Customized Policy Adaptation

As is mentioned in Section V-B, to tune the weights of the outputs, we can get adapted policies from pretrained WagerWin-DMC models. Here, we show how our models improve the WP score by simple customized policy adaptation. We set $(a^{(w)}, b^{(w)}, a^{(l)}, b^{(l)}) = (0, 1, 0, 1)$, then $\hat{Q}^{\pi'} = 2p^{\pi} - 1$. It forces the model to predict the Q-value under the WP objective instead of the ADP objective. We name this adapted model *WD-Adapt*.

Table III summarizes the WP results of head-to-head completions among WD-Adapt 1e9, WD 1e9, OD-WP 1e9, and OD-ADP 1e9, and RLCard. Not surprisingly, OD-WP 1e9 outperforms WD 1e9 and OD-ADP 1e9, as it directly learns its policy under the WP objective. The performance of WD 1e9 is 2.76% higher than OD 1e9, probably because WagerWin additionally learns the information of winning probabilities p^{π}. However, by adopting customized policy adaptation, WD-Adapt 1e9 attains a 1.3% increment of WP score from WD 1e9 (shown with underline), and is ranked the first WP model. After the adaptation, the WP score of WD-Adapt 1e9 is slightly better than OD-WP 1e9, even though OD-WP 1e9 is trained using the WP objective while WD-Adapt 1e9 is not. The results show that WagerWin-DMC can efficiently and effectively learn adequate WP information under the ADP objective and maintain both WP and ADP scores. This feature may be very useful in some circumstances.

D. Case Study

We conduct some case studies to examine what WagerWin-DMC models have learned and what their output values mean. We pick up a representative case from the self-play data of WD 1e9 models and visualize the outputs w.r.t. the available actions. More case studies are provided in Appendix C-B (supplementary material). The listed outputs are, respectively, winning probability p^{π}, winning Q-value $Q^{(w)}$, losing Q-value $Q^{(l)}$, estimated Q-value $\hat{Q}^{\pi} = p^{\pi} \max(R^{(w)}_{\min}, Q^{(w)}) +$

TABLE II
ADP RESULTS IN SMALLDOU

Rank	A \ B	WD 1e9	WD 6e8	OD-ADP 1e9	OD-WP 1e9	OD-ADP 6e8	RLCard
1	WD 1e9	-	**0.1283**	**0.1415**	**0.2016**	**0.2546**	**1.9059**
2	WD 6e8	-0.1283	-	**0.0409**	**0.0883**	**0.1640**	**1.8775**
3	OD-ADP 1e9	-0.1415	-0.0409	-	**0.0577**	**0.1481**	**1.8242**
4	OD-WP 1e9	-0.2016	-0.0883	-0.0577	-	**0.0480**	**1.7418**
5	OD-ADP 6e8	-0.2546	-0.1640	-0.1481	-0.0480	-	**1.7719**
6	RLCard	-1.9059	-1.8775	-1.8242	-1.7418	-1.7719	-

Note: This table shows the ADP results of models against each other by playing 10,000 randomly sampled decks. Model A outperforms model B if ADP is larger than 0 (highlighted in boldface). The models are ranked according to the number of other models that they beat.

TABLE III
WP RESULTS IN SMALLDOU

Rank	A \ B	WD-Adapt 1e9	OD-WP 1e9	WD 1e9	OD-ADP 1e9	RLCard
1	WD-Adapt 1e9	-	**0.5004**	**0.5130**	**0.5418**	**0.8247**
2	OD-WP 1e9	0.4996	-	**0.5088**	**0.5354**	**0.8276**
3	WD 1e9	0.4870	0.4912	-	**0.5276**	**0.8148**
4	OD-ADP 1e9	0.4582	0.4646	0.4724	-	**0.8045**
5	RLCard	0.1753	0.1724	0.1852	0.1955	-

Note: This table shows the WP results of models against each other by playing 10,000 randomly sampled decks. Model A outperforms model B if WP is larger than 0.5 (highlighted in boldface).

TABLE IV
ADP AND WP OF WAGERWIN AGAINST PRETRAINED *DOUZERO*

A \ B	DouZero-ADP		DouZero-WP		DouZero-SL	
	ADP	WP	ADP	WP	ADP	WP
WagerWin 8e9	**0.009**	0.502	0.101	0.453	0.802	0.616

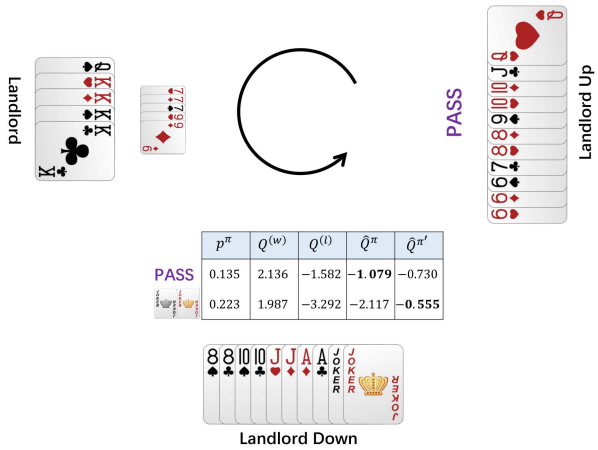

Fig. 4. Case study: A difference between ADP policy and WP policy. The WP policy tends to play the Rocket while the ADP policy chooses to pass.

$(1-p^{\pi})\min(R_{\max}^{(l)}, Q^{(l)})$, and WP adaptation Q-value $\hat{Q}^{\pi'} = 2p^{\pi} - 1$. We bold the top value for both ADP and WP actions.

Fig. 4 shows the difference between the ADP policy $\arg\max_a \hat{Q}^{\pi}(s,a)$ and the WP policy $\arg\max_a \hat{Q}^{\pi'}(s,a)$. Landlord Down now has two available actions. Under the ADP objective WD 1e9 chooses to pass while WD-Adapt 1e9 plays the Rocket. This case matches the report in *DouZero* [7] that agents under the WP objective play more aggressively about Bombs even if they will lose. But under the ADP objective, the player should not easily play Bombs or Rocket as these actions will double $R^{(l)}$.

Another interesting observation is as follows: From the output values, the WagerWin-DMC model learns that to play the Rocket will double the wager, as $Q^{(w)}$ is close to 2. Moreover, here the model is aware that the Landlord probably has a Bomb, because its $Q^{(l)}$ for Rocket is close to -4. If Landlord Down cannot win this game after playing the Rocket, it may lose 2^2 points. To stop loss, in the later round, Landlord Down splits its Rocket and gets -2 points.

E. *WagerWin in* DouDizhu

In the end, we conduct a brief experiment to compare WagerWin-DMC with *DouZero* [7] on *DouDizhu*. The released pretrained *DouZero* models on cloud storage[5] are directly used for the comparison. The involved models are as follows.

1) DouZero-*ADP:* The pretrained *DouZero* agents with ADP as an objective.
2) DouZero-*WP:* The pretrained *DouZero* agents with WP as an objective.
3) DouZero-*SL:* An SL baseline. It uses the same state representation and neural architecture as *DouZero*-ADP and *DouZero*-WP, and trains with 49 990 075 samples collected from human experts.
4) *WagerWin:* Our WagerWin-DMC implementation of *DouZero* models. All settings remain the same as *DouZero*-ADP for a fair comparison.

[5][Online]. Available: https://pan.baidu.com/s/18g-JUKad6D8rmBONXUDuOQ

According to *DouZero* paper [7], both *DouZero*-ADP and *DouZero*-WP are trained for 30 days with four GPUs and roughly 1.7e10 timesteps sampled data. Meanwhile, we train WagerWin for only 18 days with two RTX 3090 GPUs and 8e9 timesteps.

The ADP and WP of WagerWin 8e9 against pretrained *DouZero* agents are summarized in Table IV. We follow the same evaluation pattern, where two agents play the same 10 000 randomly sampled decks against each other. The boldface value in Table IV indicates that WagerWin achieves better performance than *DouZero*-ADP using less than half the sampled data. It can be concluded that the WagerWin framework doubles the training efficiency from the original state-of-the-art algorithm in *DouDizhu*.

VII. Conclusion

This work analyzes the problem of common deep RL methods in gambling games and proposes WagerWin, a simple yet effective framework that suits environments such as *DouDizhu*. The reward distribution of gambling games is separated into two parts depending on the win/lose outcome, and raising actions further widen the gap between these two parts. However, the value function of common RL targeted at the expected returns, which causes a redundant loss term. The redundant loss term is significant and can make the training process less stable and efficient. To address the problem, WagerWin learns the decoupled winning probability, losing Q-value, and winning Q-value. These three values are independently updated. To explicitly introduce new information enables WagerWin to apply customized policy adaptation, where one can tune the inclined policy for different objectives. We conduct extensive experiments on SmallDou, a reduced version of *DouDizhu*, and with *DouZero*, the state-of-the-art *DouDizhu* AI. The results show that WagerWin significantly improves the performance and efficiency from the original model. These benefits hardly bring extra computational costs.

In the future, we plan to implement WagerWin in other popular gambling games like Texas *hold'em* and *Mahjong*. We would also like to apply WagerWin to other classic RL algorithms, such as PPO, to further test its effectiveness and generalization. Moreover, we are aware that, in imperfect-information EFGs, common deep RL algorithms do not guarantee the convergence to the solution in game theory. It would be of significance to extend our study onto CFRs [14], [32] or NFSP [33] for further study.

References

[1] D. Silver et al., "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[2] D. Silver et al., "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[3] O. Vinyals et al., "Grandmaster level in Starcraft II using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[4] C. Berner et al., "Dota 2 with large scale deep reinforcement learning," 2019, *arXiv:1912.06680*.

[5] D. Ye et al., "Mastering complex control in MOBA games with deep reinforcement learning," in *Proc. 34th AAAI Conf. Artif. Intell.*, 2020, pp. 6672–6679.

[6] J. Li et al., "Suphx: Mastering Mahjong with deep reinforcement learning," 2020, *arXiv:2003.13590*.

[7] D. Zha et al., "DouZero: Mastering DouDizhu with self-play deep reinforcement learning," in *Proc. 38th Int. Conf. Mach. Learn.*, 2021, pp. 12333–12344.

[8] E. Zhao et al., "AlphaHoldem: High-performance artificial intelligence for heads-up no-limit poker via end-to-end reinforcement learning," in *Proc. 36th AAAI Conf. Artif. Intell.*, 2022, pp. 4689–4697.

[9] M. G. Bellemare, W. Dabney, and R. Munos, "A distributional perspective on reinforcement learning," in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 449–458.

[10] W. Dabney, M. Rowland, M. G. Bellemare, and R. Munos, "Distributional reinforcement learning with quantile regression," in *Proc. 32nd AAAI Conf. Artif. Intell.*, 2018, pp. 2892–2901.

[11] W. Dabney, G. Ostrovski, D. Silver, and R. Munos, "Implicit quantile networks for distributional reinforcement learning," in *Proc. 35th Int. Conf. Mach. Learn.*, 2018, pp. 1104–1113.

[12] P. Sunehag et al., "Value-decomposition networks for cooperative multi-agent learning based on team reward," in *Proc. 17th Int. Conf. Auton. Agents MultiAgent Syst.*, 2018, pp. 2085–2087.

[13] T. Rashid et al., "QMIX: Monotonic value function factorisation for deep multi-agent reinforcement learning," in *Proc. 35th Int. Conf. Mach. Learn.*, 2018, pp. 4292–4301.

[14] M. Zinkevich, M. Johanson, M. H. Bowling, and C. Piccione, "Regret minimization in games with incomplete information," in *Proc. Adv. Neural Inf. Process. Syst.*, 2007, pp. 1729–1736.

[15] J. F. Nash, "Non-cooperative games," *Ann. Math.*, vol. 54, pp. 286–295, 1951.

[16] M. Bowling, N. Burch, M. Johanson, and O. Tammelin, "Heads-up limit Hold'em Poker is solved," *Science*, vol. 347, no. 6218, pp. 145–149, 2015.

[17] O. Tammelin, N. Burch, M. Johanson, and M. Bowling, "Solving heads-up limit Texas Hold'em," in *Proc. 24th Int. Joint Conf. Artif. Intell.*, 2015, pp. 645–652.

[18] M. Moravcík et al., "DeepStack: Expert-level artificial intelligence in no-limit poker," 2017, *arXiv:1701.01724*.

[19] N. Brown and T. Sandholm, "Libratus: The superhuman AI for no-limit poker," in *Proc. 26th Int. Joint Conf. Artif. Intell.*, 2017, pp. 5226–5228.

[20] M. Lanctot, K. Waugh, M. Zinkevich, and M. H. Bowling, "Monte Carlo sampling for regret minimization in extensive games," in *Proc. Adv. Neural Inf. Process. Syst.*, 2009, pp. 1078–1086.

[21] N. Brown and T. Sandholm, "Superhuman AI for multiplayer poker," *Science*, vol. 365, no. 6456, pp. 885–890, 2019.

[22] V. Mnih et al., "Playing Atari with deep reinforcement learning," 2013, *arXiv:1312.5602*.

[23] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proc. Adv. Neural Inf. Process. Syst.*, 1999, pp. 1057–1063.

[24] J. Schulman et al., "Proximal policy optimization algorithms," 2017, *arXiv:1707.06347*.

[25] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *Proc. Mach. Learn.: ECML 2006, 17th Eur. Conf. Mach. Learn.*, 2006, pp. 282–293.

[26] G. Tan et al., "Winning rate prediction model based on Monte Carlo tree search for computer Dou Dizhu," *IEEE Trans. Games*, vol. 13, no. 2, pp. 123–137, Jun. 2021.

[27] Q. Jiang et al., "DeltaDou: Expert-level DouDizhu AI through self-play," in *Proc. 28th Int. Joint Conf. Artif. Intell.*, 2019, pp. 1265–1271.

[28] N. Brown, A. Bakhtin, A. Lerer, and Q. Gong, "Combining deep reinforcement learning and search for imperfect-information games," *Adv. Neural Inf. Process. Syst.*, vol. 33, pp. 17057–17069, 2020.

[29] R. S. Sutton and A. G. Barto, *Reinforcement Learning - An Introduction* (Adaptive Computation and Machine Learning Series). Cambridge, MA, USA: MIT Press, 1998.

[30] M. J. Osborne and A. Rubinstein, *A Course in Game Theory*. Cambridge, MA, USA: MIT Press, 1994.

[31] D. Zha et al., "RLCard: A toolkit for reinforcement learning in card games," 2019, *arXiv:1910.04376*.

[32] N. Brown, A. Lerer, S. Gross, and T. Sandholm, "Deep counterfactual regret minimization," in *Proc. 36th Int. Conf. Mach. Learn.*, 2019, pp. 793–802.

[33] J. Heinrich and D. Silver, "Deep reinforcement learning from self-play in imperfect-information games," 2016, *arXiv:1603.01121*.