RLCard: A Toolkit for Reinforcement Learning in Card Games

Daochen Zha[1], **Kwei-Herng Lai**[1], **Yuanpu Cao**[1*], **Songyi Huang**[2], **Ruzhe Wei**[1*], **Junyu Guo**[1*], **Xia Hu**[1]
[1] Department of Computer Science and Engineering, Texas A&M University, College Station, USA
[2] Simon Fraser University, BC, Canada
{daochen.zha, khlai037}@tamu.edu, yuanpucao@gmail.com, songyih@sfu.ca,
ruzhe.wei@outlook.com, {guojunyu, xiahu}@tamu.edu

Abstract

We present RLCard, an open-source toolkit for reinforcement learning research in card games. It supports various card environments with easy-to-use interfaces, including Blackjack, Leduc Hold'em, Texas Hold'em, UNO, Dou Dizhu and Mahjong. The goal of RLCard is to bridge reinforcement learning and imperfect information games, and push forward the research of reinforcement learning in domains with multiple agents, large state and action space, and sparse reward. In this paper, we provide an overview of the key components in RLCard, a discussion of the design principles, a brief introduction of the interfaces, and comprehensive evaluations of the environments. The codes and documents are available at `https://github.com/datamllab/rlcard`.

Introduction

Reinforcement learning (RL) is a promising paradigm in Artificial Intelligence for learning goal-oriented tasks. Through interactions with the environments, reinforcement learning agents learn to make decisions at each state in a trial-and-error fashion. With neural networks as function approximators, deep reinforcement learning has recently achieved breakthroughs in various domains: Atari games (Mnih et al. 2015), Go game (Silver et al. 2017), continuous control (Lillicrap et al. 2015), and neural architecture search (Zoph and Le 2016), just to name a few. Out of these achievements, however, reinforcement learning is still immature and unstable in applications with multiple agents, large decision space or sparse reward.

In this paper, we introduce various styles of card environments for reinforcement learning research. Card games are ideal testbeds with several challenges. **First**, card games are played by multiple agents who must learn to compete or collaborate with each other. For example, in Dou Dizhu, peasants need to work together to fight against the landlord in order to win the game. **Second**, card games have huge state space. For instance, the number of states in UNO can reach 10^{163}. The cards of each player are hidden from the other players. A player not only needs to consider her own hand, but also has to reason about the other players' cards from the signals of their actions. **Third**, card games may have large action space. For example, the possible number of actions in Dou Dizhu can reach 10^4 with an explosion of card combinations. **Last**, card games may suffer from sparse reward. For example, in Mahjong, winning hands are scarce. We observe one winning hand every five hundreds of games if playing randomly. Moreover, card games are easy to understand with huge popularity. Games such as Texas Hold'em, UNO and Dou Dizhu are played by hundreds of millions of people. We usually do not need to spend efforts on learning the rules before we can dive into algorithm development.

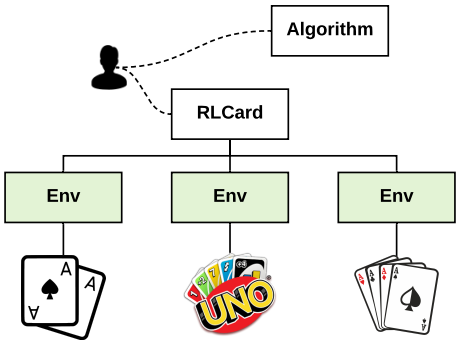


Figure 1: An overview of RLCard. It supports various styles of card games, such as betting games, Chinese Poker, and boarding games, wrapped by easy-to-use interfaces.

To develop card environments with easy-to-use interfaces is a challenging task. First, the interfaces must be accessible to RL researchers who may or may not have a game theory background. In the extensive form games, the player will not observe her next state immediately after taking an action. The next state is exposed to the player only after all other players have chosen their actions. This makes it difficult to design environment interfaces. Second, the environments need to be configurable. The state representation, action abstraction, reward design, or even the game rules should be easily adjusted for research purposes.

We present RLCard, an opensource toolkit designed for reinforcement learning in card games. It supports various card environments, as summarized in Table 1. The interfaces are straightforward for reinforcement learning. The transitions of each player and collected and well organized after a complete game in the multi-agent setting. We also provide a single-agent interface, where the other players are simulated

*Authors contribute during the visit at Texas A&M University.

Environment	InfoSet Number	Avg. InfoSet Size	Action Size
Blackjack	10^3	10^1	10^0
Leduc Hold'em	10^2	10^2	10^0
Limit Texas Hold'em	10^{14}	10^3	10^0
Dou Dizhu	$10^{53} \sim 10^{83}$	10^{23}	10^4
Mahjong	10^{121}	10^{48}	10^2
No-limit Texas Hold'em	10^{162}	10^3	10^4
UNO	10^{163}	10^{10}	10^1

Table 1: A summary of the games in RLCard. **InfoSet Number:** the number of the information sets; **Avg. InforSet Size:** the average number of states in a single information set; **Action Size:** the size of the action space (without abstraction). Note that for some games, we can only provide a range of the complexity estimation. For example, Dou Dizhu allows a large number of legal combinations of cards, which makes it challenging to estimate the size of the state space.

using pre-trained models. The state and action encoding can be easily configured. The games are implemented under the same structure with clear logic. The evaluation tools are provided to measure the performance by winning rates of tournaments. Future versions will extend the toolkit to include more environments. The goal of RLCard is to bridge reinforcement learning and imperfect information games, and push forward the research of reinforcement learning in domains with multiple agents, large state space, large action space, and sparse reward.

Overview

In this section, we give an overview of RLCard, and introduce the interfaces. More introductions can be found in Appendix. Figure 1 shows an overview of RLCard. Each game is wrapped by an environment class with easy-to-use interfaces. With RLCard, we can focus on algorithm development instead of engineering efforts on games. When developing the toolkit, we adopt the following design principles:

- **Reproducible.** Results on the environments can be reproduced. The same result should be obtained with the same random seed in different runs.
- **Accessible.** Experiences are collected and well organized after each game with straightforward interfaces. State representation, action encoding, reward design, or even the game rules, can be conveniently configured.
- **Scalable.** New card environments can be added conveniently into the toolkit with the above design principles. We try to minimize the dependencies in the toolkit so that the codes can be easily maintained.

Available Environments

The toolkit provides various styles of card games that are popular among hundreds of millions of people, including betting games, Chinese Poker, and some boarding games. Table 1 summarizes the card games in RLCard and estimates the complexity of each game. The game size can be measured by the number of information sets, which are the observed states from the view of one player. The average size of the information set is defined as the average number of possible game states in each information set. For example, given the observation from the view of one player in Texas Hold'em, the other players could have many possible hands. Each possible hand corresponds to one game state in this information set. The size of the action space is also provided since large action space will greatly increase the difficulty.

Since most of games have very large state space, it is challenging to immediately solve these human-size games. Thus, we have also implemented some smaller versions of some large games. For example, RLCard also implements a smaller version of Dou Dizhu, where we only keep cards 8, 9, 10, J, Q, K, and A. This variant keeps key features of Dou Dizhu but with much smaller state space.

Basic Interface

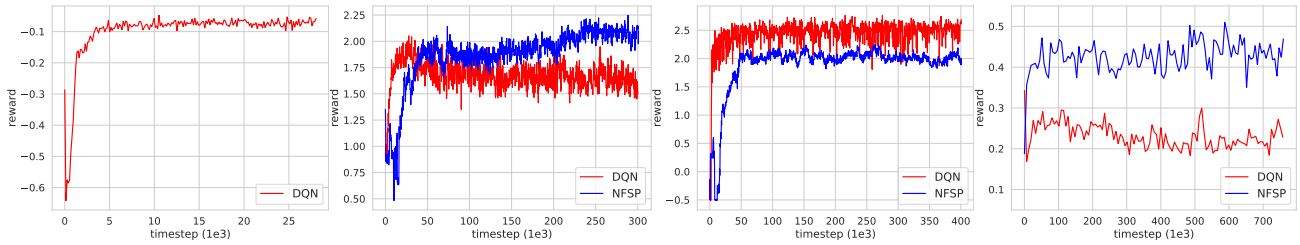
We provide a `run` function for quickly getting started. It directly generates payoffs and game data, which are organized as transitions, i.e., (state, action, reward, next_state, done). This interface is designed for algorithms that do not need to traverse the game tree. An example of running Dou Dizhu with three random agents is as follows:

```
import rlcard
import RandomAgent

# Initialize the environment
env = rlcard.make('doudizhu')

# Initialize random agents
agent = RandomAgent()
env.set_agents([agent, agent, agent])

while True:
        # Generate data from the environment
        trajectories, payoffs = env.run()
        # Train agent here
```

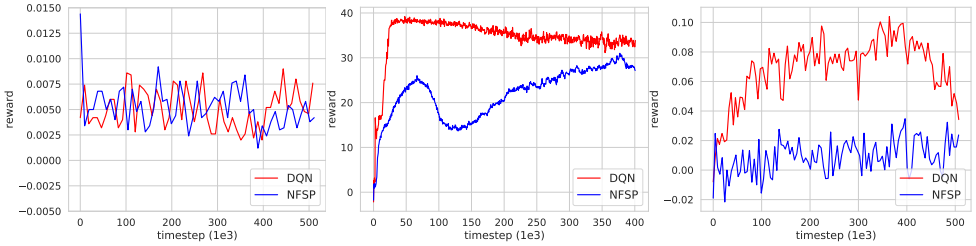
For sampling based algorithms that do not require traversing backward in the game tree (Heinrich, Lanctot, and Silver 2015; Heinrich and Silver 2016; Lanctot et al. 2017), the basic interface could be preferred since we do not need to care about the details of the traversing.

Advanced Interfaces

We also provide some advanced interfaces that operate upon the game tree. Following other RL toolkits (Brockman et al. 2016; Lanctot et al. 2019), we define a `step` function which moves the environment to the next state given the current action. To enable traversing backward, we provide a `step_back` function, which traverses back to the previous



(a) Blackjack (b) Leduc Hold'em (c) Limit Texas Hold'em (d) Dou Dizhu



(e) Mahjong (f) No-limit Texas Hold'em (g) UNO

Figure 2: Learning curves on the card environments in terms of the performance against random agents. X-axis represents the total steps taken in the environment. Y-axis is the reward achieved by playing against random agents.

state. Note that the action of the current player will lead to the observation of the next player, and the current player can observe her next state only after all other players have chosen their actions. Thus, users need to be careful with `step` and `step_back` since the next state is "delayed".

Note that the deign of `step` and `step_back` is similar to traditional tree-based interface. Specifically, `step` is corresponding to accessing child node, and `step` would access the parent node. This design enables flexible node visiting strategies of the game tree, such as external sampling MCCFR (Lanctot et al. 2009).

Evaluation

This section conducts experiments to evaluate the toolkit. We mainly focus on the following two questions: (1) How the state-of-the-art reinforcement learning algorithms perform on the introduced environments? (2) How many computation resources are required to generate game data?

Training Agents on Environments

We apply Deep Q-Network (DQN) (Silver et al. 2016), Neural Fictitious Self-Play (NFSP) (Heinrich and Silver 2016), and Counterfactual Regret Minimization (CFR) (Zinkevich et al. 2008) to the environments. These algorithms belong to different categories. DQN is a standard single-agent reinforcement learning algorithm, NFSP is a deep reinforcement learning approach for extensive games with imperfect information, and CFR is a standard regret minimization method for extensive imperfect information games. For DQN, we fix other players as random agents so that DQN agent can be trained in single-agent setting. We only test CFR on Leduc Hold'em since it is computationally expensive, requiring complete traversal of the game tree. Blackjack is only tested by DQN because it is a single-agent environment.

To evaluate the agents is non-trivial. The performance of imperfect information game is usually measured by exploitablity (Zinkevich et al. 2008; Johanson et al. 2011), which searches for the best response against the trained policy. However, it is computationally expensive to obtain the best response for the large environments in the toolkit since it relies on traversal of the game tree. Thus, we evaluate the performance based on winning rates. In this paper, we adopt two methods to empirically evaluate the performance. First, we report the winning rates of the agents against random agents. Second, we compare the agents with tournaments. In our experiments, the hyperparameters are lightly tuned. For DQN, the memory size is selected in $\{2000, 100000\}$, the discount factor is set to 0.99, Adam optimizer is applied with learning rate 0.00005, and the network structure is MLP with size 10-10 128-128, 512-512 or 512-1024-2048-1024-512 based on the size of the state and action space. For NFSP, the anticipatory parameter is chosen from $\{0.1, 0.5\}$. Memory size for supervised learning and reinforcement learning are 10^6 and 3×10^4, respectively.

Results against random agents. The learning curves in terms of the performance against random agents are shown in Figure 2. The rewards of the agents are obtained through competitions against random agents. Specifically, the rewards of betting games (Leduc Hold'em, Limit Texas Hold'em, No-limit Texas Holdem) are defined as the average winning big blinds per hand. The rewards of the other games are obtained directly from the winning rates. In Dou Dizhu, players are in different roles (landlord and peasants). We fix the role of the agent as the landlord in evaluation.

We make two observations. First, all the algorithms have similar results against random agents. DQN is slightly better than NFSP on Texas Hold'em and UNO, while NFSP is

	×1		×4		×8		×16	
	total	per step	total	per step	total	per step	total	per step
Blackjack	156.1	1.1×10^{-4}	45.6	3.3×10^{-5}	23.0	1.7×10^{-5}	12.8	9.3×10^{-6}
Leduc Hold'em	204.6	8.0×10^{-5}	58.7	2.3×10^{-5}	28.7	1.1×10^{-5}	17.9	7.0×10^{-6}
Limit Texas Hold'em	324.6	1.1×10^{-4}	83.3	2.8×10^{-5}	45.9	1.6×10^{-5}	24.6	8.3×10^{-6}
Dou Dizhu	87270.8	1.4×10^{-3}	22894.3	3.6×10^{-4}	12753.9	2.0×10^{-4}	7275.9	1.1×10^{-4}
Mahjong	74786.0	8.1×10^{-4}	20825.9	2.3×10^{-4}	11059.7	1.2×10^{-4}	6169.6	6.7×10^{-5}
No-limit Texas Hold'em	597.7	1.4×10^{-4}	160.6	3.7×10^{-5}	81.9	1.9×10^{-5}	48.4	1.1×10^{-5}
UNO	4952.9	1.1×10^{-4}	1366.0	3.0×10^{-5}	696.5	1.5×10^{-5}	440.7	9.5×10^{-6}

Table 2: Running time in seconds of $1,000,000$ games with random agents, under one process and multiple processes. Per step: the running time divided by the number of performed timesteps.

Tournament	NFSP	DQN
Leduc Hold'em	1.0691	-1.0691
Limit Texas Hold'em	-0.0308	0.0308
Dou Dizhu with NFSP landlord	0.7049	0.2951
Dou Dizhu with DQN landlord	0.7303	0.2697
Mahjong	-0.0090	-0.0104
No-limit Texas Hold'em	9.5610	-9.5610
UNO	-0.0428	0.0428

Table 3: Average payoffs of NFSP and DQN by playing $10,000$ games. For Dou Dizhu, we switch roles of landlord and peasants and report the results separately. For Mahjong, two DQN agents and two NFSP agents randomly choose seats in each game, and the averaged results are reported.

slightly better than DQN on Leduc Hold'em and Dou Dizhu. It is reasonable for DQN to achieve this result since DQN is trained to exploit the random agents. Second, NFSP and DQN are highly unstable in large games. Specifically, they only achieve minor improvements during the learning process on UNO, Mahjong and Dou Dizhu. These games are challenging due to their large state/action space and sparse reward. We believe there is a lot of room for improvement. More efforts are needed to study how we can stably train reinforcement learning agents in these large environments.

Tournament results. We report the average payoffs the agents achieved when playing against each other. The results between NFSP and DQN are shown in Table 3. We observe that NFSP is stronger than DQN on most of the environments. We also compare CFR with NFSP and DQN on Leduc Hold'em. CRF achieves better performance, winning 0.0776 and 1.2493 against NFSP and DQN, respectively.

Discussion We further analyze the trained agents. We find that the DQN agents play very aggressively in betting games. For example, in Leduc Hold'em environment, DQN agent tend to choose "raise" or "call" in almost every decision. Interestingly, this naive strategy works well when the opponent is a random agent since the random agent may easily choose "fold" so that that the DQN policy can win. However, DQN policy may be highly exploitable since one can easily find its weaknesses. Thus, performance against random agents can only be a way to get a sense of whether the agent is improving on the environment, but is **NOT** enough to be used to evaluate algorithms. For large games, we recommend evaluating algorithms by playing against existing models. To benchmark the evaluation, we will develop rule-based agents and stronger pre-trained models in the future.

Running Time Analysis

We evaluate the efficiency of the implemented environments by running self-play on the games with random agents. Specifically, we report the running time in seconds of $1,000,000$ games using a single process and multiple processes. Since Dou Dizhu, UNO and Mahjong have long sequences in one game, we additionally report a normalized version of running time, i.e., the average running time per timestep. Our experiments are conducted on a server with 24 Intel(R) Xeon(R) Silver 4116 CPU @2.10GHz processors and 64.0 GB memory. Each experiment is run 3 times with different random seeds. The average running time in seconds is reported in Table 2. We observe that all the environments achieve higher throughputs with more processors.

Related work

There are a few open-source reinforcement learning libraries, most of which focus on single-agent environments (Brockman et al. 2016; Duan et al. 2016; Shi et al. 2019). Recently, there have been some projects that support multi-agent environments (Vinyals et al. 2017; Zheng et al. 2018; Juliani et al. 2018; Suarez et al. 2019). However, they do not support card game environments. A contemporary framework OpenSpiel (Lanctot et al. 2019) provides a large collection of games, including several simple card games. Our toolkit is specifically designed for card games with straightforward interfaces, supporting various styles of card games that are not included in existing toolkits.

The most popular techniques for solving poker games in literature are Counterfactual Regret Minimization (CFR) (Zinkevich et al. 2008) and its variants (Brown et al. 2018). Achievements have been made on betting games such as Texas Hold'em (Moravčík et al. 2017; Brown and Sandholm 2017). However, CFR is computationally expensive, since it relies on complete traversal of the game tree, and is infeasible for games with large state space such as Dou Dizhu (Jiang et al. 2019).

Recent studies show that reinforcement learning strategies can perform well in betting games (Heinrich, Lanctot, and Silver 2015; Heinrich and Silver 2016; Lanctot et al. 2017), and achieve satisfactory performance in Dou Dizhu (Jiang et al. 2019). The inspiring results and the flexibility of RL offer the opportunity to explore deep reinforcement learning in more difficult card games with large state and action space.

Conclusions and Future Directions

In this paper, we introduce RLCard, an open-source toolkit for reinforcement learning research in card games. RLCard supports multiple challenging card environments wrapped with common and easy-to-use interfaces. In the future, we plan to enhance the toolkit in several aspects. First, in order to benchmark the evaluation, we would like to design rule-based agents and provide more pre-trained models for evaluation. Second, we plan to develop visualization and analysis tools for the environments. Third, we will further accelerate the environments with more efficient implementations. Last, we will include more interesting games and more algorithms to enrich the toolkit.

Acknowledgements

We would like to thank JJ World Network Technology Co., LTD for the generous support.

References

[Brockman et al. 2016] Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; and Zaremba, W. 2016. Openai gym. *arXiv preprint arXiv:1606.01540*.

[Brown and Sandholm 2017] Brown, N., and Sandholm, T. 2017. Safe and nested subgame solving for imperfect-information games. In *Advances in neural information processing systems*.

[Brown et al. 2018] Brown, N.; Lerer, A.; Gross, S.; and Sandholm, T. 2018. Deep counterfactual regret minimization. *arXiv preprint arXiv:1811.00164*.

[Duan et al. 2016] Duan, Y.; Chen, X.; Houthooft, R.; Schulman, J.; and Abbeel, P. 2016. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*.

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

[Heinrich, Lanctot, and Silver 2015] Heinrich, J.; Lanctot, M.; and Silver, D. 2015. Fictitious self-play in extensive-form games. In *International Conference on Machine Learning*.

[Jiang et al. 2019] Jiang, Q.; Li, K.; Du, B.; Chen, H.; and Fang, H. 2019. Deltadou: Expert-level doudizhu ai through self-play. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*.

[Johanson et al. 2011] Johanson, M.; Waugh, K.; Bowling, M.; and Zinkevich, M. 2011. Accelerating best response calculation in large extensive games. In *Twenty-Second International Joint Conference on Artificial Intelligence*.

[Juliani et al. 2018] Juliani, A.; Berges, V.-P.; Vckay, E.; Gao, Y.; Henry, H.; Mattar, M.; and Lange, D. 2018. Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*.

[Lanctot et al. 2009] Lanctot, M.; Waugh, K.; Zinkevich, M.; and Bowling, M. 2009. Monte carlo sampling for regret minimization in extensive games. In *Advances in neural information processing systems*.

[Lanctot et al. 2017] Lanctot, M.; Zambaldi, V.; Gruslys, A.; Lazaridou, A.; Tuyls, K.; Pérolat, J.; Silver, D.; and Graepel, T. 2017. A unified game-theoretic approach to multiagent reinforcement learning. In *Advances in Neural Information Processing Systems*.

[Lanctot et al. 2019] Lanctot, M.; Lockhart, E.; Lespiau, J.-B.; Zambaldi, V.; Upadhyay, S.; Pérolat, J.; Srinivasan, S.; Timbers, F.; Tuyls, K.; Omidshafiei, S.; Hennes, D.; Morrill, D.; Muller, P.; Ewalds, T.; Faulkner, R.; Kramár, J.; Vylder, B. D.; Saeta, B.; Bradbury, J.; Ding, D.; Borgeaud, S.; Lai, M.; Schrittwieser, J.; Anthony, T.; Hughes, E.; Danihelka, I.; and Ryan-Davis, J. 2019. Openspiel: A framework for reinforcement learning in games.

[Lillicrap et al. 2015] Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

[Mnih et al. 2015] Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529.

[Moravčík et al. 2017] Moravčík, M.; Schmid, M.; Burch, N.; Lisỳ, V.; Morrill, D.; Bard, N.; Davis, T.; Waugh, K.; Johanson, M.; and Bowling, M. 2017. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science* 356(6337):508–513.

[Shi et al. 2019] Shi, J.-C.; Yu, Y.; Da, Q.; Chen, S.-Y.; and Zeng, A.-X. 2019. Virtual-taobao: Virtualizing real-world online retail environment for reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

[Silver et al. 2016] Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of go with deep neural networks and tree search. *Nature* 529(7587):484.

[Silver et al. 2017] Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. 2017. Mastering the game of go without human knowledge. *Nature* 550(7676):354.

[Suarez et al. 2019] Suarez, J.; Du, Y.; Isola, P.; and Mordatch, I. 2019. Neural mmo: A massively multiagent game environment for training and evaluating intelligent agents. *arXiv preprint arXiv:1903.00784*.

[Vinyals et al. 2017] Vinyals, O.; Ewalds, T.; Bartunov, S.; Georgiev, P.; Vezhnevets, A. S.; Yeo, M.; Makhzani, A.; Küttler, H.; Agapiou, J.; Schrittwieser, J.; et al. 2017. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*.

[Zheng et al. 2018] Zheng, L.; Yang, J.; Cai, H.; Zhou, M.; Zhang, W.; Wang, J.; and Yu, Y. 2018. Magent: A many-agent reinforcement learning platform for artificial collective intelligence. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

[Zinkevich et al. 2008] Zinkevich, M.; Johanson, M.; Bowling, M.; and Piccione, C. 2008. Regret minimization in

games with incomplete information. In *Advances in neural information processing systems*.

[Zoph and Le 2016] Zoph, B., and Le, Q. V. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*.

Appendix

Game Design

Card games are usually played following similar procedures. We design several abstract base classes which are implemented in the specific games. In the toolkit, some common concepts are abstracted and defined as follows:

- **Player.** The person who plays the game. Each game is usually played by multiple players.
- **Game.** A game is a complete sequence starting from one of the non-terminal states to a terminal state. At the end of a game, each player will receive a payoff.
- **Round.** A round is a part of the sequence of a game. Most card games can be naturally divided into multiple rounds. For instance, Texas Hold'em consists of four rounds of betting. In Doudi Zhu a round is finished when two consecutive players pass.
- **Dealer.** Card games usually require shuffling and allocating a deck of cards to players. Dealer is responsible for deck management.
- **Judger.** A judger is responsible for making major decisions in a round or at the end of a game. For example, the next player in UNO is decided based on the type of the last card. In Texas Hold'em, the payoff is determined in the end of a game.

The games in the toolkit are implemented by associating a class with each of the above concepts. The common design principle makes the game logic easy to follow and understand. Other card games are usually compatible with the above structure so that can be easily added to the toolkit.

Environment Interfaces

This section briefly introduces the interfaces of the toolkit. We describe state representation, action encoding, and how we can modify them to customize the environments. After that, we show how to generate data with multiple processes. Finally, we introduce a single-agent interface, where the other players are simulated by pre-trained or rule-based models. More documents and examples can be found at the Github repository.

State Representation State is defined as all the information that can be observed from the view of one player in a specific timestep of the game. In the toolkit, each state is a dictionary consisting of two values. The first value is a list of legal actions. The second value is observation. There are various ways to encode the observation. For Blackjack, we directly use the player's score and the dealer's score as a representation. For other games in the toolkit, we encode the observed cards into several card planes. For example, in Dou Dizhu, the input of the policy is a matrix of 6 card planes, including the current hand, the union of the other two players' hands, the recent three actions, and the union of all the played cards.

Action Encoding The specific actions in a game are all encoded into action indices, which are positive integers starting from 0. Each action index corresponds to exactly one action in the game. The legal actions are also represented as a list of action indices. At each step, an agent should choose one of the action indices (i.e., integer values) among the legal actions instead of specific actions (such as "hit" or "stand" in Blackjack).

For some large games, action abstraction is adopted to reduce the action space. For example, Dou Dizhu suffers from the combinatorial explosion of the action space with more than 3×10^4 actions, where any trio, plane or quad can be combined with any individual card or pair (kicker). To reduce the action space, we only encode the major part of a combination and use rules to decide the kicker. In this way, the action space of Dou Dizhu is reduced to 309.

Customization In addition to the default state and action encoding, our design enables customization of state representation, action encoding, reward design, and even the game rules.

Each game is wrapped by an `Env` class, in which we can rewrite some key functions to customize the environments. The function of `extract_state` is to convert the original game state into representation. The function of `decode_action` is to map action indices to actions. One can implement his own abstraction of the actions by modifying this function. The function of `get_payoffs` will return the payoffs of the players in the end of the game. For each game, we provide a default setting for each of the above components. Users are encouraged to customize these settings to achieve better performance.

The parameters of each game can also be adjusted. For example, one can change the number of players or the fixed raise in Limit Texas Hold'em by modifying `__init__` function of the `LimitholdemGame` class. In this way, we may adjust the difficulty of the games for our purposes and design algorithms step by step.

Parallel Training The toolkit supports generating game data with multiple processes. Running in parallel will greatly accelerate the training in large environments. Specifically, we create duplicate environments in initialization. Then each worker will copy the model parameters from the main process, generate game data in a duplicate of the environment, and send the data to the main process. The main process will collect all the data to train the agent on either CPU or GPU. Example implementations of training agents with multiple processes can be found at the Github repository.

Single-Agent Interfaces We provide interfaces to explore training single-agent reinforcement learning agents in card games. Specifically, we develop pre-trained or rule-based models to simulate other players so that the games essentially become single-agent environments from the view of one player. These single-agent environments are also challenging since they have large state and action space, and sparse reward. In the future, we plan to use different levels of simulating models to create environments with vari-

ous difficulties. The single-agent interfaces follow OpenAI Gym (Brockman et al. 2016). Specifically, in the single-agent mode, given an action, `step` function will return the next state, reward, and whether the game is done. The `reset` function will reset the game and return the initial state. Standard single-agent RL algorithms can be easily applied to the environments.