Mastering Fighting Game Using Deep Reinforcement Learning With Self-play

Dae-Wook Kim
Electronics and Telecommunications Research Institute
Daejeon, South Korea
dooroomie@etri.re.kr

Sungyun Park
Electronics and Telecommunications Research Institute
Daejeon, South Korea
tjddbs5671@etri.re.kr

Seong-il Yang
Electronics and Telecommunications Research Institute
Daejeon, South Korea
siyang@etri.re.kr

***Abstract*—One-on-one fighting game has played a role as a bridge between board game and real-time simulation game in terms of research on game AI because it needs middle-level computation power with medium-size complexity. In this paper, we propose a method to create fighting game AI agent using deep reinforcement learning with self-play and Monte Carlo Tree Search (MCTS). We also analyze various reinforcement learning configuration such as changes on state vector, reward shaping, and opponent compositions with novel performance metric. Agent trained by the proposed method was evaluated against other AIs. The evaluation result shows that mixing MCTS and self-play in a 1:3 ratio makes it possible to overwhelm other AIs in the game with 94.4% win rate. The fully-trained agent understands the game mechanism so that it waits until being close to enemy and performs actions at the optimal timing.**

***Index Terms*—reinforcement learning, self-play, Monte Carlo Tree Search, fighting game AI, FightingICE**

I. Introduction

FightingICE is a 1v1 fighting game environment, whose name is derived from the fighting game platform organized by Intelligent Computer Entertainment lab in Ritsumeikan university. One-on-one fighting game has distinct characteristics for studying diverse AI algorithms. First of all, it is an intermediate step between the genre of board games and real-time strategy simulation games. Board game AIs such as Deep Blue [1] or AlphaGo [2] are given enough time to consider future actions of players while playing the game. One-on-one fighting game AI, however, has little time so that it has to react in real-time. This makes fighting game more difficult than board game. Meanwhile, fighting game has shorter playtime with less action complexity than real-time simulation game such as Starcraft or DOTA2. It has an advantage of designing and evaluating AI algorithms with less computation power.

The FightingICE environment offers unique traits that allow researchers to study general fighting game AI. First, it provides delayed game information in order to reflect human's physical limitation. The information occurred in real-time is delivered through simulator with a delay of approximately 0.2 seconds. For this reason, AI agent should act based on this inaccurate information. Second, it has two modes, a standard and a speed run. In the standard mode, players start with a certain amount of hit points (HP) and fight until one of them would knocked down or the playtime runs out. The speed run mode is to beat a competitor as quickly as possible. Finally, it provides a special character without revealing its specification of skills and actions. Many studies have been conducted upon these characteristics.

Competition on FightingICE [3] has held every year from 2013. It consists of 6 rounds, the composition of 2 leagues and 3 characters including the undefined character named LUD. The ranking of competitors will be determined by the total result of 6 rounds. From 2013 to 2015, rule-based bots were the mainstream. Some participant applied k-nearest neighbor or forward model to rule-based bots. Since Monte Carlo Tree Search (MCTS) emerged as a trend in 2016, a number of AIs that combines MCTS and handmade rules were submitted. Other submitted bots [4] treated opponent modeling, reinforcement learning, genetic algorithm, dynamic scripting, finite state machine, heuristic approach based on human experience, and rolling horizon evolutionary algorithm. MCTS and rule-based agent have dominated the competition until recently. On the contrary, AI agent generated by reinforcement learning has no outstanding results yet.

Although reinforcement learning is known as one of methods to create game AI, it has not been widely used because of its difficulty of applying on massive and complicate game environment. Deep neural network, however, has proven that it is able to handle the high complexity with approximation. As a result, the AI agent obtained high score than human in Atari games [5]. Moreover, it also won against pro-players in Go [2], [6] and Starcraft II [7] using deep reinforcement learning and self-play.

In this paper, we verify whether the reinforcement learning with self-play is also able to reach the high-level performance on a specific one-on-one battle game such as FightingICE. The contributions of this paper are as follows. We evaluate agents with a novel performance measure as well as ordinary win rate. In addition, we demonstrate that the proposed method makes it possible to overwhelm other entries of the past competitions. As far as we know, it is the first time parametric components are analyzed through various experiments related to self-play learning. This paper serves as a baseline for making general fighting game AI with even higher performance.

The rest of this paper is organized in the following order.

978-1-7281-4533-4/20/$31.00 ©2020 IEEE

Chapter 2 provides related works for applying deep learning to the FightingICE environment. In chapter 3, we detail our proposed method with respect to reinforcement learning and self-play. Subsequently, chapter 4 explains the evaluation methods and experiments. Experimental results and discussions are presented in chapter 5. We remark the conclusion in chapter 6.

II. Related Work

Many researchers have studied reinforcement learning to apply it to the FightingICE environment. In [8], [9], they collected gameplay data through simulation from the past bots. Convolutional neural network (CNN) model was trained to predict actions for given input state. The optimized state shape was obtained from prediction accuracy of models and also utilized to reinforcement learning. The agent was evaluated against Machete which is the winner bot of 2015 competition. It was unsuccessful to show significant achievements. Q-learning [10] and SARSA [11], one of well-known reinforcement learning algorithms, were applied in [12]. Its state included relevant game information such as HP, position and energy. Its reward was designed with numeric terms based on HP. These agents were evaluated by random action agent. SARSA agent did not exceed 50% win rate, and Q-learning agent showed a win rate of 50% or more depending on the hyper-parameter setting. Yoon and Kim [13] used visual data represented by 96 64 pixels image. The number of actions was simplified to 11, and the agent was implemented with DQN [5]. Even if it scored higher as the learning progressed, there is a clear limitation that it was tested against sandbag bot alone. Takano et al. [14] showed remarkable result among reinforcement learning approaches. They combined offensive reward and defensive reward. Those rewards are given when the opponent's HP or my HP is reduced. The agent, also implemented with DQN, ranked 4th in the standard league of 2017 competition. It beat MCTS agent for the first time using deep reinforcement learning. Meanwhile, there is another kind of study that mixes reinforcement learning and MCTS [15]. This method proposed selecting the optimal action by MCTS in search space partially reduced by reinforcement learning. The agent was trained by playing against GigaThunder and tested onto several other bots. Its average win rate indicated slightly higher than 50%.

In one-on-one games, self-play is used as a major method to improve performance through the fights against its own copy. This has an advantage of training even in situations where play data is not enough; however, there is a risk that the agent converges to specific strategy, and it does not always guarantee that it will work well. Hence, in recent studies [2], [7], [16], it is more preferred that self-play learning starts from certain baselines rather than scratch. All of them initially perform supervised learning through human's play data. After that, self-play boosts skill level of the agent. For the first work [2], the data obtained from self-play enhances value network accuracy as well as prevents it from over-fitting. Likewise, in the second work [7], supervised learning was conducted using approximately 970,000 human battle replays before self-play reinforcement learning. Due to the circular structure of strategy like a rock-paper-scissors, They divided training agents into three categories: main agents, main exploiters, and league exploiters. This composition avoids converging to the local optimum. It is notable that a mechanism called prioritized fictitious self-play (PFSP) is applied, which determines the matching opponents based on the win rates. Oh et al. [16] created three types of agent groups: aggressive, balanced, and defensive by reward shaping. Each version of agent was regularly stored in a shared pool and used for self-play learning. Agents from the shared pool was extracted by probabilities. The more latest the model is, the higher the probability gets. In [6], they used MCTS instead of human's play data. The role of MCTS is to search wide state space. Neural network narrows down the state space effectively and is trained from MCTS. By complementing each other's deficiencies, self-play reinforcement learning with MCTS enabled to obtain better performance without supervised learning.

III. Method

A. State and Action

A state `s` obtained from the game environment becomes an input of neural network. The input features of the state are based on [14] and we added additional features such as threshold and relative distance. Those details are described in Table I.

The character attributes are composed of HP, energy, position, speed, action, character state, and remaining action frame. Skills require different energy consumption depending on the skill type so that it is necessary to divide into several sections. We introduce energyT features. EnergyT5 represents normalized energy from 0 to 1 when energy is between 0 and 5. If the energy is 5 or higher, EnergyT5 clips it to 1. EnergyT30, EnergyT50, and EnergyT150 are applied in the same way. By making multiple features for the energy, agent can recognize states more clearly to select the skills. We set intervals of energy with reference to the skill table. Movement feature represents the direction. X movement gets 0 when agent is moving to the left and 1 to the right. The action and the character state are encoded as one hot vector. The character state has one of four types: ground, crouch, on air, and recovery. The size of the character attribute becomes 74, and the entire size doubled to 148 for two players.

The projectile attribute contains information on long-range attacks. For data efficiency, projectile features with maximum two for each player were allocated to the state vector. Hence, if three or more projectiles are created by one player, feature vector describes only for the first two. If there is no projectile on the screen, their values become all zero. Similar to character attribute, the size of the projectile attribute is 12 in total.

The distance attribute represents the relative distance between two players. To act basic attacks such as punch or kick, the character needs to be located closer than a certain distance. Therefore, one hot encoding was added to distinguish distance more clearly. DistanceT is encoded by three sections, less than

TABLE I
FEATURES FOR STATE

	Feature Name	Value	Size	Feature Name	Value	Size	Feature Name	Value	Size
Character Attribute (for each player)	HP	0 1	1	Energy	0 1	1	EnergyT5	0 1	1
	EnergyT30	0 1	1	EnergyT50	0 1	1	EnergyT150	0 1	1
	X position	0 1	1	Y position	0 1	1	X movement	0 or 1	1
	Y movement	0 or 1	1	X velocity	0 1	1	Y velocity	0 1	1
	Action	0 or 1	56	Character state	0 or 1	4	Remaining frame	0 1	1
	Controllable	0 or 1	1						
Projectile Attribute (for each player)	X position	0 1	2	Y position	0 1	2	Damage	0 1	2
Distance Attribute	Distance	0 1	1	DistanceT	0 or 1	3			
Time Attribute	Remaining time	0 1	1						

0.15, 0.15 0.3, and more than 0.3. The time attribute for the current round was added to the last input feature.

165 features are extracted for one frame by aforementioned way. We designed our state vector using two frames, delayed frame and simulated frame. The delayed frame is basically provided by the game simulator. Taking this frame alone hinders estimation on the value function of reinforcement learning. On the other hand, the simulated frame contains the estimated real-time information elapsed 0.2 seconds from the delayed frame. It takes an effect of prediction. Even if it may not match the actual observation, it is helpful in certain situation such as when player is moving along a specific trajectory in the air. This is similar to human inferring the actions of the opponent.

The number of output is set to 56. This is same as the number of actions provided by the game. Instead, we adopted action masking mechanism which filters the actions that are not possible in the current state. For example, air projectile attack action is only valid when the player is in the air. When this action occurs from the output of network, it is transferred to the game simulator as a no-op action. Through the action masking, the agent randomly selects 56 actions at the beginning of learning. It is induced to reduce the frequency of invalid actions itself at the end.

B. Reward

Similar to the previous studies [14], [16], reward is composed of three components. The first one is for differences in HP. When the agent reduces opponent's HP, it gets a positive reward. On the contrary, if the agent is damaged by the opponent, it gets a negative reward. This HP reward is normalized between 10 and 10 for entire HP 400. The second one is obtained at the end of the round depending on the result of win or lose. It gives 10 for win, 0 for draw and 10 for lose. The last one gives constant negative values to the agent until the round ends. If the time is over, the agent obtain 10 reward for a round. This makes it possible to defeat the opponent quickly as well as to win the game. Ideally, it can get a reward up to 20 if it completely knock down the opponent as soon as game starts. For the reward scale, we had trial and error with values of 1, 10, and 100. The value of 10 showed the best stability on training. To reflect true goal of winning games, the discounting factor is set to 1.

C. Reinforcement Learning Algorithm

Proximal policy optimization (PPO) algorithms [17], one of policy gradient methods, is applied for the proposed deep reinforcement learning. PPO has an advantage for efficient learning by inferring the value for each state and relative action value. Policy clipping also helps to adjust the policy stably without sudden shift. Besides, since the state s described above contains continuous values such as the character's position or energy, PPO is regarded to be more suitable than Q-learning based algorithms.

For network parameters , state s, action a, reward r, and actual reward G_t from time step t to T acquired by the current policy, training process is to find optimal policy $(a|s)$. Loss L^{PG} of policy gradient method at time t is

$$L^{PG} = \hat{E}[\log (a_t|s_t) G_t] \quad \text{where } G_t = \sum_{k=t+1} {}^{k\ t\ 1} r_k \tag{1}$$

Estimated advantage function $\hat{A}$ is expressed as (2) with the value function V , reward r_t, and discount factor . Actor-critic loss $L^{PG\ A}$ is

$$\hat{A} = r_t + + {}^{T\ 1} r_{t+T\ 1} + {}^{T} V(s_{t+T}) \quad V(s_t) \tag{2}$$

$$L^{PG\ A} = \hat{E}[\log (a_t|s_t) \hat{A}_t] \tag{3}$$

The final loss L^{CLIP} with clipping range becomes

$$L^{CLIP} = \hat{E} \left[\frac{(a_t|s_t)}{-(a_t|s_t)} \hat{A}_t \right] = \hat{E}[r \ \hat{A}_t] \quad \text{subject to } 1 < r < 1 + \tag{4}$$

The agent was trained on 2-stage learning schedule, as shown in Fig. 1. In the first stage, the agent plays against SampleMctsAi (also called MCTS AI) provided by FightingICE. When we adopt a simple random action agent, it happens to generate a biased experience because it is not able to intentionally approach to the player. On the other hand, MCTS searches wide breadth through simulations to find which action is advantageous in a particular situation in real time. It provides more general experience. Through this stage, the agent learns how to react to the most situations, how to filter specific actions caused by action masking, and how to acquire precise value function. In the second stage, self-play

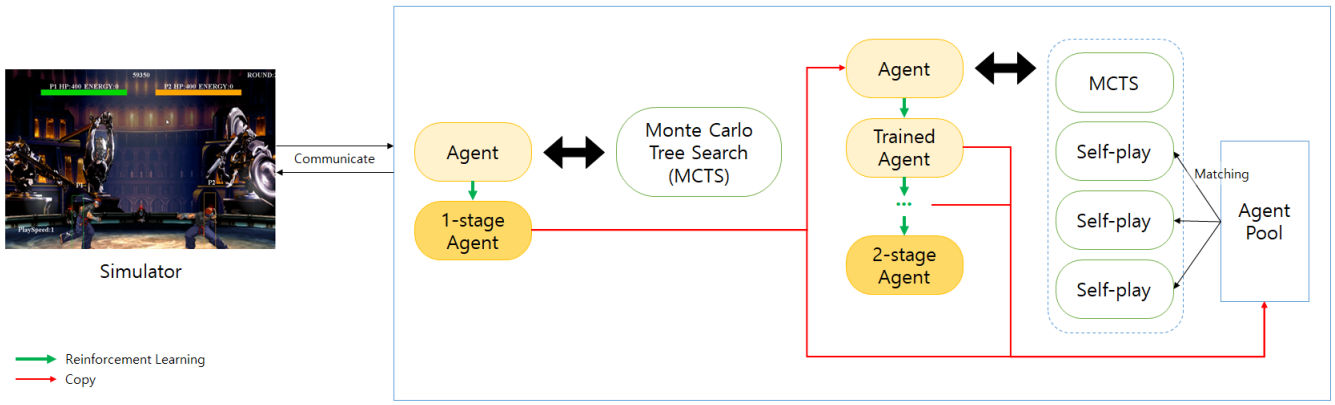

Fig. 1. An overview of proposed two stage reinforcement learning. In the first stage, agent is trained against MCTS AI alone. After performance saturation in the first stage, agent is trained against itself as well as MCTS AI. During training in second stage, agent is duplicated into agent pool to find its weakness.

TABLE II
HYPER-PARAMETERS FOR REINFORCEMENT LEARNING

name	value
step size	4
environment size	16
batch size (step size environment size)	64
learning rate	1e-4
value function coefficient	0.25
entropy coefficient	1e-3
clipping range for policy	0.02
clipping for value function	none
clipping for gradient norm	0.5
trade-off factor for generalized advantage estimator	0.95
minibatch size	4
number of epoch for optimizing surrogate	4

learning is added. While it helps to find the current agent's weakness and remove vulnerability, MCTS keeps generality and prevents sticking to a specific strategy. Other hyper-parameters for reinforcement learning is described in Table II.

D. Self-play Learning

To match a desired opponent, prioritized self-play algorithm is applied on the agent pool. After competing against all rivals in the agent pool, the average win rates of the last 30 games for each rival are calculated. We designed matching probability P_{match} as Equation (5) where W_B is the win rate against rival B. Based on Equation (5), the opponents difficult to win are matched more frequently.

$$P_{match}(B) = \frac{1 \quad W_B}{\sum_{C2Pool} (1 \quad W_C)} \tag{5}$$

Training agent is copied to the agent pool every 500,000 time steps. In addition, it is also duplicated whenever it wins all the other agents more than 80%. In an initial agent pool, there exists only one agent trained on stage 1. However, the number of agents in the pool increases through duplication process.

E. Network Architecture

The neural network of agent is composed of multi-layer perceptron (MLP). All layers are fully connected layers, and rectified linear unit (ReLU) is used for activation function. The numbers of nodes from the first hidden layer to the third hidden layer are 330, 330, and 165 each. At the end, This network is divided into two branches with 165 and 80 hidden units, respectively for policy and value estimation. The final output size for policy network equals 56 and value network equals 1.

IV. EXPERIMENT

The agent is trained with 32 Core CPU, 64 GB RAM, and RTX 2080 GPU. The hardware specification is enough to train simultaneously with the separated 16 multiple environments. Proposed algorithm was implemented in Python with TensorFlow, and PPO was provided from stable-baselines library. The version of FightingICE we used is FTG 4.40.

A. Evaluation

We evaluated our agent by playing against other AIs. The average win rate as well as the average HP difference and elapsed time were carefully selected for measurement. The opponent bots for evaluation were chosen from top rankers submitted in the 2017 and 2019 competitions. We ignored 2018 competition bots because the unique combo system was applied in 2018 competition alone. The names of 10 evaluation bots are FooAI, GigaThunder, JayBot_2017, Mutagen, LGIST_Bot, HaibuAI, DiceAI, Toothless, FalzAI, and SampleMctsAi. Several other bots including ReiwaThunder which won in 2019 were excluded due to its version compatibility. We played 9 games for each AI and changed the side to play another 9 games. All evaluations were performed only with ZEN character, expecting the results would not be affected by characters if specifications are provided.

B. Experiment Details

First, we compared agents in terms of the state vector configuration on whether delayed frame or simulated frame is used. If the state vector is composed of either delayed or simulated, the number of network input requires half of the original. This causes modification of network structure. Therefore, the numbers of units of the first two layers were adjusted to 165 each and the rest of the network remained as same as the original. All experiments were evaluated under playing against MCTS AI.

In the second experiment, we tested reward shaping. For the three types of aforementioned reward, agent is trained separately with all the types, two of three types, or one type alone. Training with time penalty reward alone had been excluded from the experiment in advance because we expected that it would not be trained well. For comprehensive assessment, we defined additional measurement called SDR score, a stable damage rate. It is calculated with the win rate, HP difference, and elapsed time. Equation (6) represents the SDR score.

$$\text{SDR Score} = \frac{(\text{Win Rate}) \quad (\text{HP Di erence})}{(\text{Elapsed Time})} \tag{6}$$

For example, assuming same win rate, the agent who makes a lot of HP difference against opponent gets higher SDR score. Even if some agents are able to make the same HP difference, the agent who knocks down the opponent in a shorter time receives more SDR score. Similar to the first experiment, MCTS AI was matched for evaluation.

The third is an experiment on self-play composition for each learning stage. MCTS AI was selected for the opponent of training in the first stage, then we mixed MCTS AI and self-play in the second stage. The purpose of this experiment is to analyze how the performance changes for each stage when tuning the mixing ratio of MCTS AI and self-play. On the self-play learning, agent starts training against its own copy. Therefore, at the beginning of the self-play learning, the opponent of the first stage becomes random action agent while that of the second stage corresponds to the agent trained by MCTS AI.

Finally, the cases of other opponent combination were investigated. Agents were trained by random agents only, or by the combination of competitors in 2016 and 2017. The competitors used here are Machete, Ranezi, Thunder01, JayBot2016, Triump, Ichibanchan, and paranahueBot. We strictly chose them not to overlap the evaluation bots except MCTS AI.

V. Result and Discussion

Fig. 2 shows the training result of state configuration and reward shaping. When we adopt both simulated and delayed frame, the win rate reaches almost 1.0 at 2M steps. However, if the agents are trained with single frame, It goes less than 0.5, indicating poor learning capability. It means that use of both frames is crucial factor in aspect of working complementary to each other.

In the reward shaping, win rate converges to 0.9 at the early stage of learning when taking either HP reward or round reward. On the other hand, in the both cases of applying all reward and all but time reward, those graphs show convergence after 0.4M steps. It is remarkable that they takes time a little bit more than the prior two cases. The reason is that both HP and round rewards are deeply related to winning the game. Therefore, they affect the direction of gradient pointing to the optimal. If the rewards are not blended together, the gradient clearly points out the optimal so that the agent is able to be trained quickly. However, if rewards are mixed, the direction of the gradient is slightly twisted sideways. To test variations in network structure, etc., it is suitable to give the agent either HP or round reward, which is straightforward to implement and costs relatively shorter training time.

Even though taking single reward among HP and round looks better than using all reward, it changes by adding time reward. Excluding round reward indicated by the green line presents unstable performance. Besides, the agent did not be trained at all in the case of removing HP reward represented as the blue line. The agent with all rewards shows slower learning than the agent without time reward. This significantly means that time reward hinders the agent learning unlike HP and round rewards. The importance of reward would be aligned in order of HP, round, and time.

The middle bottom of Fig. 2 shows the average HP difference with the opponent as the training progresses. The curves depict the tendency similar to the win rate graph. The agent with either HP or round reward was trained rapidly, making a difference of more than 150 HPs in 0.2M steps. It converges to 250 after a sudden leap in score around 1.1M steps. After 1.5M steps, except ignoring either round or HP reward, all of them reached close to 250.

The significant role of time reward is proved on the top right graph in Fig. 2. The game duration of the 'All' and 'except Round' cases are approximately 7 seconds shorter than the other cases with exception of the blue line which is not able to win the game at all. According to the aforementioned result of win rate, we proved that the time reward slows the agent's training. However the time reward leads the agent to defeat the opponent quickly under a sufficient training time. For the SDR score, the case of using all rewards demonstrates the best. As a result, the best performance needs to take all rewards.

The evaluation results with respect to self-play configuration are illustrated in Fig. 3. All the graphs were depicted under the assumption of that agents reach the maximum performance in 3 million training step. In the first stage as shown in the left graph, the case of mixing MCTS and self-play in a 1:3 ratio was the best even if we had proposed playing only against MCTS. It recorded 0.939 of win rate which was close to 0.944 of the proposed method. For the second stage in the middle graph, combining MCTS and self-play in 1:3 ratio also achieved the first place while the win rate tends to decrease as the ratio of MCTS increases. It means that self-play learning is dominant factor to get high-skilled agent.

With self-play learning only, the training result did not get

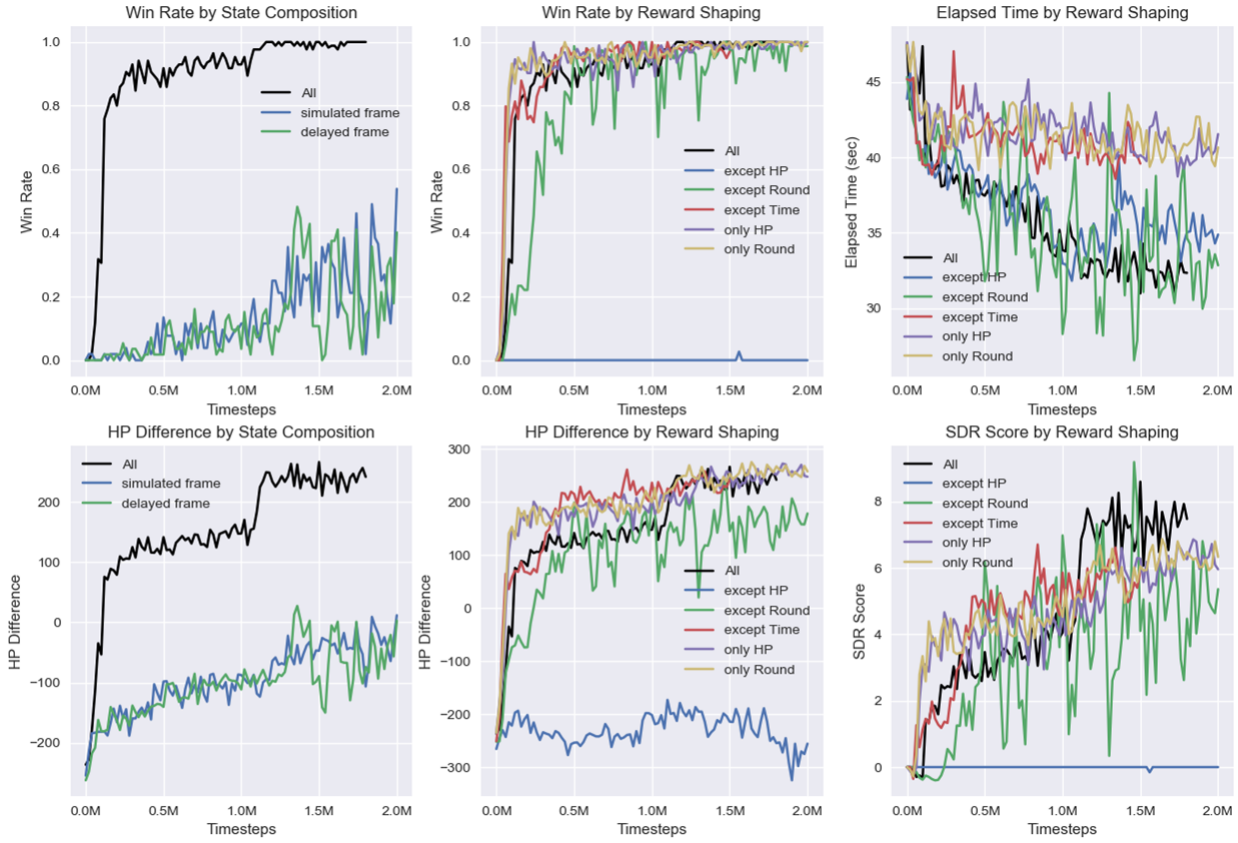

Fig. 2. Performance for state configuration and reward shaping. Performance was evaluated through win rate, HP difference, elapsed time and SDR score. Left two graphs shows performance with respect to state composition. Middle and right graphs represent performance by reward shaping.

to win much against evaluation bots in both stage 1 and stage 2. Especially in stage 2, the performance of agent declined despite of the warm start on the first stage with the win rate of 0.786. As expected in the chapter 3, it reflects the main role of MCTS that provides broad and essential experience to the agent. Note that self-play learning can makes the agent biased into a local optimum.

For the 3:1 ratio, it is unusual that the win rate in stage 2 was increased than stage 1 (MCTS only) while the win rate in stage 1 was decreased. It is caused by the opponent of self-play learning for each stage. The opponent on stage 1 was just random action agent. Since the training agent was barely taken helpful information from the self-play, it relied on MCTS more. On the other hand, the opponent on stage 2 was strong enough to defeat MCTS perfectly. Therefore, the training agent can be improved by overcoming its own weaknesses.

The right chart of Fig. 3 represents the performance according to the opponent group. The agent trained by MCTS is stronger than any others. It is interesting to note that the agent trained by mixed group of MCTS and past entries of 2016 and 2017 competition shows the weakest skill. This means that when playing against either random agent or enemies made up of past competition bots, the agent quickly learn a specific tactic to win them. Conversely, since MCTS provides the most demanding attacks in the various situations through simulation, the agent learns a universal tactic. If these two different styles are combined into one, the agent runs about in confusion. It occurs learning delay despite of sufficient time steps. Consequently, taking 1:3 ratio helps creating powerful agent. Using MCTS alone is also not a bad option when lacking self-play implementation.

At the early timestep of learning, actions of the agent should have uniform distribution. Indeed, NEUTRAL and AIR are operated more frequently in actual play thanks to the action masking, as shown in Fig. 4. After 0.3M steps, action distribution is narrowed down to around 3 actions of tackle attack STAND_D_DB_BB, middle kick STAND_B, and kick on lower posture CROUCH_FB. The model finally identified effective attacks in the first stage such as STAND_B, NEUTRAL, CROUCH_FB, and CROUCH_GUARD. At the beginning of the second stage, the model performs diverse actions again in order to overcome its weakness. Actions like STAND_D_DF_FC are tend to be presented more frequently. This tendency is changed along training. NEUTRAL is operated more than STAND_B at the end of the second stage. It is remarkable result that the fully-trained agent waited for the enemy's action by doing NEUTRAL and performed attacks at the optimal timing. On the other hand, the first stage agent depended on accidental attacks by doing STAND_B and

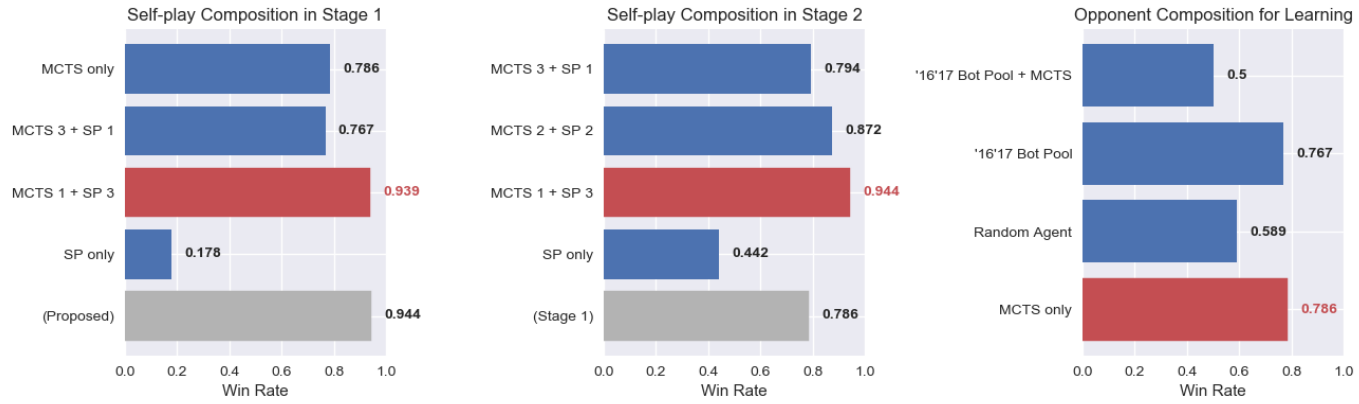

Fig. 3. Evaluation result by mixing ratio of MCTS and self-play, and opponent composition. SP means self-play. '16'17 Bot Pool is composed of competitors in 2016 and 2017.

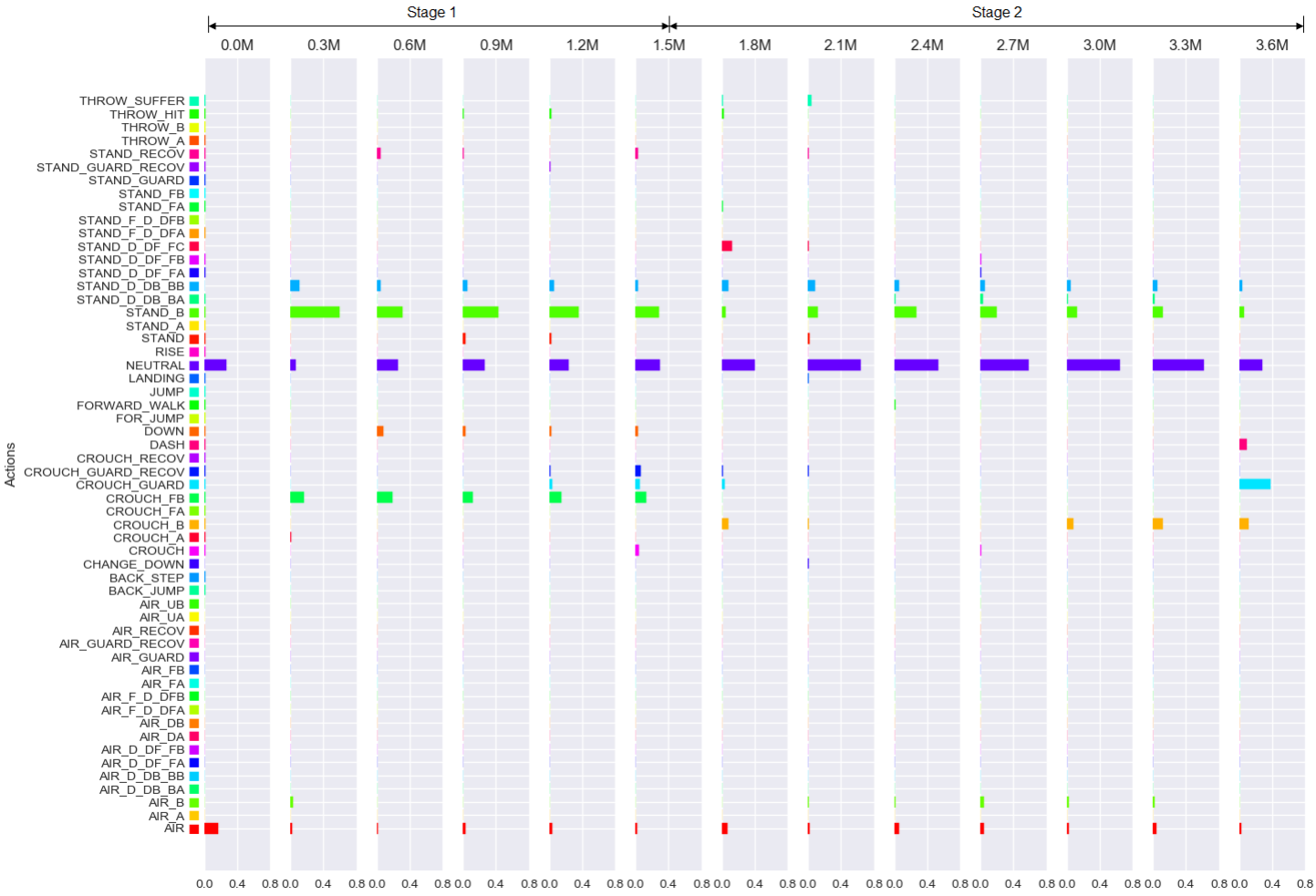

Fig. 4. Action distribution with respect to timesteps. The first stage learning is from 0 steps to 1.5 million steps. The second stage learning is from 1.5 million steps to 3.6 million steps.