Efficient Searching With MCTS and Imitation Learning: A Case Study in Pommerman

HAILAN YANG\textsuperscript{1}, SHENGZE LI\textsuperscript{2}, XINHAI XU\textsuperscript{2}, XUNYUN LIU\textsuperscript{2}, ZHUXUAN MENG\textsuperscript{2}, AND YONGJUN ZHANG\textsuperscript{2}

\textsuperscript{1}National University of Defense Technology, Changsha 410073, China
\textsuperscript{2}Academy of Military Science, Beijing 100091, China

Corresponding author: Yongjun Zhang (yjzhang@nudt.edu.cn)

This work was supported by the National Natural Science Foundation of China under Grant 11901578.

ABSTRACT Pommerman is a popular reinforcement learning environment because it imposes several challenges such as sparse and deceptive rewards and delayed action effects. In this paper, we propose an efficient reinforcement learning approach that uses a more efficient Monte Carlo tree search combined with action pruning and flexible imitation learning to accelerate the search performance, allowing the agent to avoid meaningless explorations and find some high-level strategies. Under the Pommerman benchmark, we evaluate the agent driven by the proposed approach against the heuristic and pure reinforcement learning baselines, and the results show that our method can yield a relatively high-level agent performance during combat, which demonstrates the efficiency of our method in this specific domain and its potential ability.

INDEX TERMS Reinforcement learning, Monte Carlo tree search, action pruning, imitation learning.

I. INTRODUCTION

Reinforcement Learning (RL) is a branch of machine learning (ML) that has achieved great success in recent years. The agents developed by the RL algorithm show the potential to reach a human-level intelligence in several classic computer games, such as the Atari [1], Go [2], and Starcraft [3]. The methodology behind the RL is to optimally represent the state transition process in a complex interactive system. In this system, the agent controlled by the RL algorithm can improve its strategy through several rounds of the training process to reach some acceptable future rewards. However, training an RL agent will take quite a long time to learn good policies via large numbers of trials, especially for the domains where rewards are delayed, sparse, or deceptive. During the interactive process, the agent will encounter difficulties in exploring or exploiting such an environment.

In an RL phase, the temporal difference (TD) approaches are commonly used, which relies on the Markov theory. Although this assumption can make the problem easier to solve, it always fails in the situation in which the future states are closely related to the current state. The main problem of TD learning is that the step updates are based on the initial conditions of the learning parameters. Thus, this bias will lead to a poor convergence property which is called the deadly triad [4]. There is an alternative approach named Monte Carlo tree search (MCTS) [5], which does not suffer from the problem of bias since each update is made using a true sample of the decision trace. However, the mean drawback of the MCTS is the high variance derived from the random rollout process, which leads to a considerable computational request.

In this paper, we focus on the improvement of the learning performance while using the MCTS approach. The idea comes from an observation that the conventional learning process always trains an agent from a lower level, while some high-level strategies are thrown away or diluted since an increased time cost is incurred. In a tree search or decision process, time is wasted in many unnecessary explorations that make the learning process a heavy burden. Therefore, we propose an efficient approach to accelerate the learning phase that contains three key elements of novelty: 1) We construct a hierarchical searching framework in which the top level is adapted to the final result that only indicates wins, ties, or losses, and the lower level is adapted to a sub-task to determine if the agent has captured the power-up item, evaded the bomb, or beat the enemy. 2) To find some high-level policies, the action space is pruned to avoid meaningless explorations, and the action filter is redesigned for different conditions.
search levels and tasks. 3) We use a convolutional neural network to distinguish the filtered actions by using imitation learning to bring prior knowledge into the tree policy. This idea is further evaluated through a popular RL benchmark named Pommerman [6], a simple but challenging environment, to show its potential ability. Our approach achieved significantly better performance than a baseline heuristic method and state-of-the-art RL techniques including vanilla MCTS, DQN(Deep-Q Network) [7], A2C(Advantage Actor-Critic), and A3C(Asynchronous Advantage Actor Critic) [8].

The remainder of this paper is organized as follows: in section 2 we discuss the background of Pommerman and related methods. Section 3 describes the presented learning methods and strategies. Section 4 presents our experimental details and results. Finally, we conclude with a discussion of the present work and future directions in Section 5.

II. BACKGROUND
A. POMMERMAN FRAMEWORK
Pommerman [6] is derived from the classic console game Bomberman (Hudson Soft, 1983). The game is played in a randomly generated 11 × 11 grid where four agents need to eliminate each other. During each round, the agents start from a separate corner with a single bomb equipped as shown in Fig. 1. They will step simultaneously and six actions are available to be chosen: STOP, UP, LEFT, DOWN, RIGHT, and BOMB. In addition to the agents, the board consists of passages, woods, and rigid walls. The board will always keep an accessible path between each other. Rigid walls are indestructible. Wooden walls can be destroyed by bombs and will become either a passage or a power-up item. Power-ups can increase the power and blast strength of the agent’s bomb or allow the bomb to be kickable.

There are three game settings in Pommerman: FFA (free for all, a single agent against 3 opponents), Team (2v2), and TeamRadio (team variant with the communication). Here, we only consider the FFA mode with no communication. The agent who survives to the end will receive a reward of 1, and the other will receive a reward of −1.

B. RELATED WORK IN POMMERMAN
The first Pommerman competition was organized in 2018 and focused on the FFA game mode. The first place was taken by Gong’s agent [9] which depends on a finite state machine tree search approach. As a part of NeurIPS 2018, the second competition received more attention which mainly focused on the TEAM mode. The agents ranked first and third are both MCTS-based approaches [10], and the second is another MCTS implementation with a search depth of 2. During NeurIPS 2019, the third competition was held and the first competition was developed by Gorog M. who further enhanced his previous MCTS agent.

The Pommerman environment involves several unique challenges: 1) Sparse and deceptive rewards. As mentioned before, only a non-zero reward is obtained at the end of the episode, moreover, the final rewards cannot indicate whether the agent is eliminated by other opponents or itself, and the utility of a BOMB action becomes hard to evaluate. 2) Delayed action effects. Both objects of killing the enemy, removing woods, and getting power-up are all dependent on executing a BOMB action, but it will last 10 time-steps from the bomb placed step to the bomb exploded step. Meanwhile, the kickable property will further lead the decision process to become complicated. 3) Non-zero-sum game. In the FFA mode, the independent agent will interact with others while competitive and non-competitive behaviour coexists. The existence of the optimal solution still lacks certification on a mathematical level at this time. 4) Partial observation. In some cases, the agent can only see its nearby areas, which makes the enemy’s action been unpredictable and intractable.

These unique challenges attract many researchers and some representative studies are summarized as follows. Multi-agent strategies [11] are a good review that presents and evaluates various strategies for solving the reinforcement learning problems in Pommerman. Backplay [12] is an approach that improves the sample efficiency of model-free RL by constructing a curriculum learning framework through several demonstrations. The agent starts from a state near the end of the game and searches backward to the initial state of an episode. MAGNet Agent [13] utilized a relevance graph representation of the state accompanied by self-attention and message generation, which involved the relation property of each element in learning. Continual learning [14] proposed a Continual Match Based Training (COMBAT) framework for training a group of Advantage Actor-Critic (A2C) agents in Pommerman that won the first place among the learning agents during the 2nd Pommerman Competition. Skynet Agent [15] used proximal policy optimization (PPO) with reward shaping, action pruning, and opponent curriculum learning. A3C-TP [16] extended the Asynchronous Advantage Actor-Critic (A3C) method with a novel auxiliary task of terminal prediction that predicts temporal closeness to terminal states. Later, they proposed
PI-A3C [17] which integrates MCTS as a demonstrator for A3C, to avoid meaningless suicides. In hybrid search agent [9], the author used the heuristics and tree search algorithms such as breadth first search (BFS), MCTS, and flat monte carlo search (FMCS) in Pommerman. The result shows that heuristic agents using depth-limited tree search can slightly outperform hand-made heuristics. The study in [18] proposed AMS-A3C and AMF-A3C architectures to perform agent modelling, which means that the network could perform policy improvement and opponent policy recognition simultaneously. The Pommerman agent [19] compared different methods such as Q-learning, snorkel, random forest, and MCTS. While testing these agents against the baseline agent called SimpleAgent, MCTS wins most of the time. The statistical agent [20] analysed several agents that depend on the statistical planning methods and indicated that the MCTS is outperformed.

**C. MONTE CARLO TREE SEARCH**

MCTS is a highly selective best-first tree search method. The first practical success was seen in the computer game Go [2]. MCTS has since been applied successfully to general game playing, real-time and continuous domains, multi-player games, single-player games, imperfect information games, computer games, and more [5].

In MCTS, a searching tree is generated where each node represents a state of the domain and links to one possible action. MCTS proceeds in four phases (Fig. 2): selection, expansion, simulation, and backpropagation. The MCTS algorithm proceeds by repeatedly adding one node at a time to the current tree until the terminated state. MCTS uses random actions, a.k.a. rollouts, to estimate state-action values. After the simulation, all collected rewards are back-propagated through all visited nodes.

**D. BACKPLAY**

Model-free reinforcement learning mostly requires a large number of trials to learn a good policy, especially in environments with sparse rewards, such as Pommerman. A new method called backplay [12] for reducing training time for such problems was proposed. The idea is to create a curriculum for the agent by reversing a single trajectory (i.e., state sequence) of reasonably good, but not necessarily optimal, behaviour. It starts the agent at the end of a demonstration and allow it to learn a policy in this easier setup. Then, the starting point is moved backward until the agent is training only on the initial state of the task.

**III. PROPOSED APPROACH**

Since we assume the standard reinforcement learning setting of the agent interacting in an environment over a discrete number of steps, for the sake of brevity, we will denote the symbols we have used in the following part. Here, at the time step \( t \), the agent in state \( s_t \) takes an action \( a_t \) and receives a reward \( r_t \). The discounted return is defined and simplified as \( R_t:∞ = \sum_t^{∞} γ^t r_t = r_∞ \). The state-value function, \( V^π(s) = \mathbb{E}[R_t:∞|s_t = s, π] \), is the expected return from state \( s \) following a policy \( π(a|s) \) and the action-value function, \( Q^π(s, a) = \mathbb{E}[R_t:∞|s_t = s, a_t = a, π] \), is the expected return following policy \( π \) after taking action \( a \) from state \( s \).

**A. STATE SPACE AND NETWORK ARCHITECTURE**

In each Pommerman environment time step, the agent will receive an observation in a dictionary that includes a board matrix of dimensions 11 $\times$ 11, the agent’s ammo, position, blast strength, the ability of bomb kick, etc. Continual learning and terminal prediction are mostly used in the CNN architecture [14], [21]. In our approach, to fit such information with a CNN model, we maintain a feature map of 11 $\times$ 11 $\times$ 20. The first two dimensions are used to fit the board size and the last is the number of channels. These channels include the item position on the board (passages, walls, teammate, enemies, etc.), the agent’s ability (ammo, blast strength, and bomb kick), and the bomb information (bomb life, bomb strength, bomb movement direction, and flame life) which are all encoded with the one-hot scheme. Moreover, as shown in Fig. 3, the neural network structure we used contains 2 convolutional layers, followed by policy and value heads. Each convolutional layer has 64 3 $\times$ 3 filters with a stride of 1, and the ReLU is used as the activation function, which reduces the interdependence of parameters, and alleviates the occurrence of overfitting problems. Moreover, there is no vanishing gradient problem in neural networks [22]. The input contains 20 feature planes, each of shape 11 $\times$ 11. It then convolves using 2 layers of convolution; the result thus has a shape of 11 $\times$ 11 $\times$ 64. Then, each head convolves using 64 3 $\times$ 3 kernels. Finally, the output is squashed into action probability and value estimation.

In this section, we propose a general hierarchical searching framework as shown in Algorithm 1. In this framework, we start at any state \( s \) before the decision will be made by the agent. This state can be treated as a root state \( s_{root} \) where the search is carried out. All the available actions can be composed as the action space \( \mathcal{A} \). To decide which action is the best under the state \( s \), a tree structure will be constructed with a specified action filter \( f_{\mathcal{A}_{main}} \). This filter will be related to the final game results of winning or losing.
The top-level searching process will be terminated if the root node reaches the maximum visit counts. Following the standard MCTS fashion, it will yield the utility of each action \( V_{\text{root}} \), which can be further normalized by the softmax function.

The sub-level searching process can be treated as a supplementary search to satisfy the auxiliary sub-tasks. First, the state is delineated into three separate states: evade, attack, and balance. These sub-tasks are mainly used to avoid negative or lazy behaviours. After the state has related to a specific sub-task, another search can be carried out to find the optimized solution. Then, the action utility will also be evaluated.

### B. HIERARCHICAL SEARCHING FRAMEWORK

The top searching process which using the MCTS estimate will be unreliable at the beginning of the search, but will eventually converge to a more reliable estimate after a given sufficient time, and the optimal estimate can be reached under infinite time. The sub searching process from a partial perspective determines probability distributions. Therefore, these two search results need to be further balanced. Here, we adopt the Kullback–Leibler (KL) divergence to evaluate the distribution of two sets of action utility by the following:

\[
k = D_{KL}(P_{\text{main}} || P_{\text{sub}}),
\]

where \( P_{\text{main}} \) is the distribution related to the top level search and the \( P_{\text{sub}} \) is related to a specific sub task. Then the action could be selected from the following rule:

\[
a = \begin{cases} 
\text{argmax}(P_{\text{main}}), & k < \epsilon, \\
\text{argmax}(P_{\text{main}} + w \cdot P_{\text{sub}}), & \text{otherwise}, 
\end{cases}
\]

where the \( w \) is a weight to determine the influence of a sub-task execution and \( \epsilon \) is a threshold value.

It is worth mentioning that, although from a design perspective, the two searching processes are subordinate relationship, from a realizable perspective, they can be fully paralleled, and each single tree search can also be paralleled.

### Algorithm 1 Hierarchical Searching Procedure

**Require:** root state \( s_{\text{root}} \), action space

**Ensure:** action \( a \)

Initialize search tree \( T_{\text{main}} \), global filter \( f^A_{\text{main}} \)

\[
\text{node}_{\text{root}} = T_{\text{main}}(s_{\text{root}}, f^A_{\text{main}})
\]

while \( N_{\text{root}} < N \) do

\[
\text{node}_{\text{expand}} = \text{tree}_\text{policy}(\text{node}_{\text{root}})
\]

\[
\text{reward} = \text{default}_\text{policy}(\text{node}_{\text{expand}}, f^A_{\text{main}})
\]

\[
\text{backup}(\text{node}_{\text{expand}}, \text{reward})
\]

end while

\[
Q_{\text{main}}(s_{\text{root}}, a) = 1/N(s_{\text{root}}, a) \sum_{s_{\text{child}}} V_{\text{root}}(a \rightarrow s_{\text{child}}) V(s_{\text{child}})
\]

\[
P_{\text{main}} = \text{softmax}(V(s_{\text{root}}))
\]

if \( \text{evade}_\text{condition}(s_{\text{root}}) \) then

\[\text{task}_{\text{sub}} = \text{“evade”}\]

else if \( \text{attack}_\text{condition}(s_{\text{root}}) \) then

\[\text{task}_{\text{sub}} = \text{“attack”}\]

else

\[\text{task}_{\text{sub}} = \text{“balance”}\]

end if

**Initialization:** search tree \( T_{\text{sub}} \), global filter \( f^A_{\text{sub}} \)

\[
\text{node}_{\text{root}} = T_{\text{sub}}(s_{\text{root}}, f^A_{\text{sub}})
\]

while \( N_{\text{root}} < M \) do

\[
\text{node}_{\text{expand}} = \text{tree}_\text{policy}(\text{node}_{\text{root}})
\]

\[
\text{sub}_\text{reward} = \text{default}_\text{policy}(\text{node}_{\text{expand}}, f^A_{\text{sub}})
\]

\[
\text{backup}(\text{node}_{\text{expand}}, \text{sub}_\text{reward})
\]

end while

\[
Q_{\text{sub}}(s_{\text{root}}, a) = 1/N(s_{\text{root}}, a) \sum_{s_{\text{child}}} V_{\text{root}}(a \rightarrow s_{\text{child}}) V(s_{\text{child}})
\]

\[
P_{\text{sub}} = \text{softmax}(V(s_{\text{root}}))
\]

\[
k = D_{KL}(P_{\text{main}} || P_{\text{sub}})
\]

if \( k < \epsilon \) then

\[a = \text{argmax}(P_{\text{main}})\]

else

\[a = \text{argmax}(P_{\text{main}} + wP_{\text{sub}})\]

end if

### C. ACTION FILTERS

The problem with the MCTS game tree for more than one enemy is that the branching factor is vast, it is 6 for our
agent and $6^3$ for the other agent(s). That made our agent run-time very slow. Therefore, we considered working with the action filters to reduce the runtime and accelerate the training time.

We needed to create an action filter that allows the agent to “predict” the future value of a current state. We began noticing different results and gameplays by choosing different action filters, and their weights. We started with some basic weighting vectors and features, such as how many bombs are on the board at the current moment, how many agents are left, how many breakable walls our agent destroyed, or the distance from our enemies, and similar other conjectures. Then, we added features that allowed the agent to avoid committing suicide, such as checking if the agent is on the flames (that marks the explosion area) or near them, following the same rationale for bombs or enemies. Then, we became offensive - we placed bombs near our enemies that would cause their death down that branch and cleared our way to other agents and so on.

There are three types of action filters using these features applied in our method as shown in Algorithm 2: evade, attack, and balance, accompanied with 3 individual score functions (different weights are assigned to these features). All the filters are designed by different rules and unique conditions of entries.

**Evade:** This will try to escape the bombs surrounding the agent and filter all the passable positions and avoid fights. We will assign punishment for being near enemies or bombs/flames.

**Attack:** This is used while an enemy approaches a Manhattan distance of 6, and all the available directions to the enemy are listed. The agent will prefer moving closer to the enemies and placing bombs near them/near breakable walls. However, in this situation, the agent will die in the process. We assign high weights to place bombs near enemies/walls.

**Balance:** This agent will take reasonable action in the current state. He knows to be offensive and defensive, balancing between the two. This is the best action filter.

**D. IMITATION LEARNING WITH BACKPLAY**

Although the action filters could accelerate the searching performance, we still think it is insufficient to be applied in a real-time running condition in Pommerman. The reward is only granted at the end of each game, and since the state space is large it takes a long time for the reward to propagate values of earlier states. In this sense, we use the more efficient imitation learning - **Backplay** - to give the filtered actions some biased probabilities, which make tree policy selection more efficient.

The key idea in Backplay is that we do not initialize the Markov Decision Process (MDP) in only a fixed $s_0$. Instead, we assume access to a demonstration which reaches a sequence of states $\{s^d_0, s^d_1, \ldots, s^d_T\}$. For each training episode, we uniformly sample starting states from the sub-sequence $\{s^d_{T-k}, s^d_{T-k+1}, \ldots, s^d_{T-j}\}$ for some window $[j, k]$. As training continues, advance the window according to a curriculum by increasing the values of $j$ and $k$ until we are training on the initial state in every episode ($j = k = T$).

We do this based on SimpleAgent playing against itself from the Pommerman repository. To implement imitation learning with neural networks, we implemented the key idea of backplay by saving full states for the entire game. Then, for every window of 5 states starting from the end of the game, we randomly selected one of the states to be an initial state from which we ran a learning episode. After each learning episode, we shifted the window 5 states backward and repeated the process until we reached the true initial state.
FIGURE 5. Results of different strategies.

TABLE 1. Reward shaping.

| Reward Shaping                        | Value   |
|---------------------------------------|---------|
| Bombing walls/enemies gets             | 0.02(0.05) |
| Picking up ammo gets                  | 0.02    |
| Not Moving into the walls gets        | -0.03   |
| Picking up kick gets                  | 0.02    |
| Picking up blast strength gets        | 0.02    |
| Making invalid actions gets           | -0.03   |

E. REWARD SHAPING

As is known, one of the challenges of Pommerman is that of sparse and delayed rewards. It only provides a reward when the game ends (with a maximum episode length of 800), either 1 or −1. To address the sparse reward problem, we add a reward function during the learning.

IV. EXPERIMENTS AND RESULTS

In this section, we will show the experimental results against the baseline (3 SimpleAgents) under the FFA mode. It should be noted that the SimpleAgent, has an average win rate of 23.4% compared to the other 3 SimpleAgents.

SimpleAgent serves as a purely rule-based agent that is developed by the author of Pommerman’s environment.

To illustrate our approach, we compare our proposed methods with some traditional reinforcement learning methods, such as Q-learning, DQN, A2C, and vanilla MCTS. We tested 200 games for each agent against SimpleAgent. The initial position of our agent was randomized. The expertise level of MCTS is 200 rollouts. We calculate the win ratio of our agent. The results are shown in Fig. 6. The Q-learning agent has the worst effect and the H-MCTS-NN (our hierarchical MCTS with neural network and imitation learning) performs best. For this reason the Q-learner can choose a “good” action only if it has been in the same state before. Since the Pommerman environment has a large state space and every battle starts on a random map, there is almost no chance to encounter the same scenario. The A2C win rate was 23.6% slightly better than that of SimpleAgent, converged faster than that of A3C, and the training time was 5 days. With limited time and hardware, we envision the possibility of further improving the results.
Algorithm 2 Action Filters

1: function GetFeature(EnvState)  
2:   Initialization: feature \( m = 0, S_\text{root} = \text{EnvState} \)  
3:   \( m[1] = \text{GetAgentAlive}(S_\text{root}) \)  
4:   \( m[2] = \text{GetNumberBombs}(S_\text{root}) \)  
5:   \( m[3] = \text{GetBrokenWalls}(S_\text{root}) \)  
6:   \( m[4] = \text{IsBombNearWall}(S_\text{root}) \)  
7:   \( m[5] = \text{IsOnFlame}(S_\text{root}) \)  
8:   \( m[6] = \text{GetClosestBombDist}(S_\text{root}) \)  
9:   \( m[7] = \text{GetClosestAgentDist}(S_\text{root}) \)  
10: \( m[8] = \text{IsBombNearEnem}(S_\text{root}) \)  
11: \( m[9] = \text{IsNearBomb}(S_\text{root}) \)  
12: \( m[10] = \text{IsStuck}(S_\text{root}) \)  
13: \( m[11] = \text{GetEnemActions}(S_\text{root}) \)  
14: return \( m \)  
15: end function

16: function GetActionScore(task_sub)  
17:   Initialization: action score \( \text{Score} = 0 \), weight \( w = 0 \)  
18:   if task_sub = evade then  
19:     \( w = \text{GetEvadeWeight()} \)  
20:   else if task_sub = attack then  
21:     \( w = \text{GetAttackWeight()} \)  
22:   else  
23:     \( w = \text{GetBalanceWeight()} \)  
24:   end if  
25:   for \( i, \text{action} \) in action space do  
26:     next_state = GetNextState(obs, action)  
27:     \( m = \text{GetFeature}(\text{next state}) \)  
28:     \( \text{Score}[i] = m \cdot w \)  
29:   end for  
30: return \( \text{Score} \)  
31: end function

The best performers were the vanilla MCTS and its variants. As we know that the main disadvantage of vanilla MCTSs is their runtime, the time required for them to act longer than the Pommerman environment time limit. Therefore, we use a neural network to lower its runtime, and the result shows an improvement.

Fig. 4 presents the learning against the rule-based opponent curve of H-MCTS-NN, MCTS, A2C, and A3C. Our method, H-MCTS-NN, outperforms the standard MCTS, A2C, and A3C in terms of both learning faster and converging to a better policy in learning against rule-based opponents. Our training models have an average win rate of more than 60% after 20,000,000 episodes and converge at 70%. However, the original MCTS, A2C, and A3C reach no more than 40% after 40,000,000 episodes.

To fully evaluate our approach, here we also include the conventional MCTS with the same setting, the hierarchical MCTS with action filters(H-MCTS), and the H-MCTS with neural network and imitation learning biased (H-MCTS-NN). We can vary the expertise level of MCTSs by changing the number of rollouts per action-selection. We experimented with 200 games against SimpleAgent with 100 (Fig. 7) and 200 (Fig. 8) rollouts per move.
The results show that the H-MCTS-NN agent perform better than others, and compared to the conventional MCTS, it will both improve the winning percentage and prevent the portion of ties. The same method with 200 rollouts learns better than those with 100 rollouts. Although more searches will achieve better results, the rollouts should be carefully balanced due to the limitation of response time per step. It should be noted that this is an original version with serial logic, and it is apparent that the MCTS scheme can be easily paralleled which will be included in our future research. In general, the presented approach achieves better performance than the baseline agent and conventional MCTS, which shows the potential ability to be applied in other real-time games.

After we obtained very good results by implementing the “balance” H-MCTS-NN approach for the heuristic, we started running it against our own attack agent and evade agent, and obtained the following results shown in Fig. 5. As can be seen, the balance agent managed to yield reasonably good results, and wins most of the games played against other agents.

V. CONCLUSION

This paper has proposed an efficient searching approach by using hierarchical searching, action pruning, and imitation learning, which is adopted and tested in a Pommern benchmark. The result shows that our method performs better than the provided baseline agent (SimpleAgent). There is still a considerable space for our agent to improve in the future. Future work may include the parallel algorithm for MCTS and more comparisons against other types of agents to analyse and measure the performance of our agents, further improvements, and evaluations for the multi-agent mode, such as the Team and TeamRadio modes.

REFERENCES

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. A. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, “Human-level control through deep reinforcement learning,” Nature, vol. 518, pp. 529–533, Feb. 2015.

[2] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, “Mastering the game of go without human knowledge,” Nature, vol. 550, no. 7676, pp. 354–359, Oct. 2017.

[3] O. Vinyals et al., “Starcraft II: A new challenge for reinforcement learning,” CoRR, vol. abs/1708.04782, pp. 1–20, Aug. 2017.

[4] R. S. Sutton and A. G. Barto, “Reinforcement learning: An introduction,” IEEE Trans. Neural Netw., vol. 9, no. 5, p. 1054, 2002.

[5] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, “A survey of Monte Carlo tree search methods,” IEEE Trans. Comput. Intell. AI in Games, vol. 4, no. 1, pp. 1–43, Mar. 2012.

[6] C. Resnick, W. Eldridge, D. Ha, D. Britz, J. Foerster, J. Togelius, K. Cho, and J. Bruna, “Pommern: A multi-agent playground,” Tech. Rep., 2018.

[7] V. Francois-Lavet, R. Fonteneau, and D. Ernst, “Playing Atari with deep reinforcement learning,” Comput. Sci., 2015.

[8] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” in Proc. Int. Conf. Mach. Learn., 2016, pp. 1928–1937.

[9] H. Zhou, Y. Gong, L. Mugrai, A. Khalifa, A. Nealen, and J. Togelius, “A hybrid search agent in Pommernman,” in Proc. 13th Int. Conf. Found. Digit. Games, Aug. 2018, pp. 1–4.

[10] T. Osogami and T. Takahashi, “Real-time tree search with pessimistic scenarios,” 2019, arXiv:1902.10870. [Online]. Available: http://arxiv.org/abs/1902.10870.

[11] D. Shah, N. Singh, and C. Talegaonkar, “Multi-agent strategies for Pommernman,” Tech. Rep.

[12] C. Resnick, R. Raileanu, S. Kapoor, A. Peysakhovich, K. Cho, and J. Bruna, “Backplay: ‘Man muss immer umkehren,’” 2018, arXiv:1807.06919. [Online]. Available: http://arxiv.org/abs/1807.06919.

[13] A. Malyshova, T. T. Sung, C.-B. Sohn, D. Kudenko, and A. Shipman, “Deep multi-agent reinforcement learning with relevance graphs,” 2018, arXiv:1811.12557. [Online]. Available: http://arxiv.org/abs/1811.12557.

[14] P. Peng, L. Pang, Y. Yuan, and C. Gao, “Continual match based training in Pommernman: Technical report,” 2018, arXiv:1812.07297. [Online]. Available: http://arxiv.org/abs/1812.07297.

[15] C. Gao, P. Hernandez-Leal, B. Kartal, and M. E. Taylor, “Skynet: A top deep RL agent in the inaugural Pommernman team competition,” 2019, arXiv:1905.01360. [Online]. Available: http://arxiv.org/abs/1905.01360.

[16] B. Kartal, P. Hernandez-Leal, and M. E. Taylor, “Using Monte Carlo tree search as a demonstrator within asynchronous deep RL,” 2018, arXiv:1812.00045. [Online]. Available: http://arxiv.org/abs/1812.00045.

[17] B. Kartal, P. Hernandez-Leal, C. Gao, and M. E. Taylor, “Safer deep RL with shallow MCTS: A case study in Pommernman,” 2019, arXiv:1904.05759. [Online]. Available: http://arxiv.org/abs/1904.05759.

[18] P. Hernandez-Leal, B. Kartal, and M. E. Taylor, “Agent modeling as auxiliary task for deep reinforcement learning,” in Proc. AAAI Conf. Artif. Intell. Interact. Digit. Entertainment, 2019, vol. 15, no. 1, pp. 31–37.

[19] O. B. D. M. Segal and G. K. V. Zilberman, “Pommernman agent,” Tech. Rep.

[20] D. Perez-Lebana, R. D. Gaina, O. Drageset, E. Ilhan, M. Balla, and S. M. Lucas, “Analysis of statistical forward planning methods in Pommernman,” in Proc. AAAI Conf. Artif. Intell. Interact. Digit. Entertainment, 2019, vol. 15, no. 1, pp. 66–72.

[21] B. Kartal, P. Hernandez-Leal, and M. E. Taylor, “Terminal prediction as an auxiliary task for deep reinforcement learning,” CoRR, vol. abs/1907.10827, pp. 1–8, Jul. 2019. [Online]. Available: http://arxiv.org/abs/1907.10827.

[22] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, G. Wang, J. Cai, and T. Chen, “Recent advances in convolutional neural networks,” Pattern Recognit., vol. 77, pp. 354–377, Aug. 2018.

HAILAN YANG received the B.S. degree in computer science and technology from the Ocean University of China, in 2018. She is currently pursuing the master’s degree with the National University of Defense Technology. Her research interests include multi-agent reinforcement learning algorithms and basic software.

SHENGE LI received the Ph.D. degree in aerospace science from the National University of Defense Technology, in 2018. He is currently an Assistant Research Fellow with the Academy of Military Science, China. His research interests include reinforcement learning, mechanics, and boundary element method.
XINHAI XU received the Ph.D. degree in computer science from the National University of Defense Technology, in 2012. He is currently an Associate Research Fellow with the Academy of Military Science, China. His research interests include artificial intelligence algorithm, simulation, and parallel computing.

ZHUXUAN MENG received the Ph.D. degree in aerospace science from the National University of Defense Technology, in 2018. She is currently an Assistant Professor with the Academy of Military Science, China. Her research interests include artificial intelligence algorithm and CFD.

XUNYUN LIU received the Ph.D. degree in computer science from the National University of Defense Technology, in 2018. He is currently an Assistant Research Fellow with the Academy of Military Science, China. His research interests include reinforcement learning, simulation, and parallel computing.

YONGJUN ZHANG received the Ph.D. degree in computer science from the National University of Defense Technology, in 2000. He is currently a Research Fellow with the Academy of Military Science, China. He has participated in the National High Technology Research and Development Program of China and the National Natural Science Foundation of China.