Accelerating Training in Pommerman with Imitation and Reinforcement Learning

Hardik Meisheri
TCS Research
Mumbai, India
hardik.meisheri@tcs.com

Omkar Shelke
TCS Research
Mumbai, India
shelke.omkar@tcs.com

Richa Verma
TCS Research
Delhi, India
richa.verma4@tcs.com

Harshad Khadilkar
TCS Research
Mumbai, India
harshad.khadilkar@tcs.com

Abstract

The Pommerman simulation was recently developed to mimic the classic Japanese game Bomberman, and focuses on competitive gameplay in a multi-agent setting. We focus on the 2×2 team version of Pommerman, developed for a competition at NeurIPS 2018[1]. Our methodology involves training an agent initially through imitation learning on a noisy expert policy, followed by a proximal-policy optimization (PPO) reinforcement learning algorithm. The basic PPO approach is modified for stable transition from the imitation learning phase through reward shaping, action filters based on heuristics, and curriculum learning. The proposed methodology is able to beat heuristic and pure reinforcement learning baselines with a combined 100,000 training games, significantly faster than other non-tree-search methods in literature. We present results against multiple agents provided by the developers of the simulation, including some that we have enhanced. We include a sensitivity analysis over different parameters, and highlight undesirable effects of some strategies that initially appear promising. Since Pommerman is a complex multi-agent competitive environment, the strategies developed here provide insights into several real-world problems with characteristics such as partial observability, decentralized execution (without communication), and very sparse and delayed rewards.

Keywords: Deep Reinforcement Learning; Imitation Learning; Multi-Agent Deep Reinforcement Learning; Pommerman

1 Introduction

Reinforcement learning has achieved success in solving several complex problems, ranging from game playing [1, 2] to robotics [3] and autonomous driving [4]. Many algorithms originally developed for gameplay have been subsequently adapted for real-world applications, highlighting the importance of the former from both theoretical and practical perspectives. However, many of the current algorithms in RL have been designed for single-agent domains, where the environment is either stationary [5], or else is subject to a fixed set of rules or policies [1]. In addition, RL algorithms are prone to sample inefficiency, due to which it takes vast amount of training to reach to desirabele performance [6]. Relatively few studies [2] have considered situations with human or AI-driven opponents. Building RL algorithms for *mixed* cooperative and competitive environments with complex dynamics is

---

[1]https://nips.cc/Conferences/2018/CompetitionTrack

difficult, because of the challenge of separating the true reward signal from noise. At the same time, many real-world applications such as multi-robot exploration [7] and auctions [8] make this problem interesting from a practical standpoint in addition to its theoretical depth.

The key challenges in multi-agent scenarios are as follows. First, non-stationarity of the environment from the perspective of any single agent means that not all rewards are explainable by changes in the agent's own policy [9]. This also leads to another problem of credit assignment among the agents when there are sparse and common rewards [10]. Second, environments such as Pommerman can impose restrictions on communication[2], which disqualifies multi-agent RL approaches with centralised critics. Restricted communication is not peculiar to Pommerman, but can be found in several practical situations such as drone swarms as well. Third, constraints such as partial observability and sparse rewards further increase the complexity of the problem, leading to the possibility of policy degeneration.

Two approaches from prior literature that address these issues are to either roll out the environment through tree search [11, 12] or to undertake extensive training [13, 14], both of which require significant computational resources. In this paper, we aim to strike a balance between the purity of from-scratch RL policy search, with the limitations of imitating a noisy expert policy. We do so by initially imitating the noisy expert policy (a simple heuristic provided by the game developers) in order to learn the basic functionality of Pommerman [15], and follow this by training using a stochastic on-policy algorithm. The key contributions of this paper are, (i) a stable learning paradigm for imitation followed by RL-driven improvements without allowing policy forgetting, (ii) a significant reduction in training duration compared to prior literature, and (iii) extensive evaluation of the proposed method in terms of behaviour as well as performance against baseline agents.

About Pommerman: The basic Pommerman environment contains three variants: FFA (free for all, a fully observable mode with a single player against 3 opponents), Team (the partially observable 2×2 mode that we consider in this paper), and TeamRadio (team variant with communication). The Team environment contains an 11×11 board with agents spawning at each corner, with teammates starting in opposite corners as shown in Figure 1. At any given time, Agent can only see 5 cells from its position in any direction. The objective of the game is to survive and to kill the opponents by placing bombs. Bombs explode 10 time ticks after placement. Flames from the bomb last for 2 time ticks. Initially, the bomb blast range is 3 in horizontal as well as vertical direction. There are 6 discrete actions, 4 for cardinal movement and 1 each for placing

Figure 1: Sample initial board layout. Visibility for each agent is shown in the panel on the right.

a bomb and doing nothing. In addition, there are power-ups which can increase the blast radius of the bomb, increase ammo capacity to place more than one bomb simultaneously, and the capability to kick bombs away. There are two types of walls, wooden and stone. Wooden walls can be destroyed by the bombs and might reveal power-ups, whereas stone walls are unaffected. Agents can only move where there are passages. Each game starts with a random generation of stone walls and wooden walls, which are symmetric along the diagonal. If neither of the team is able to win after 800 time ticks, the game is said to be tied. Each agent has partial visibility of 5 cells in each direction.

2 Related Work

The unique challenges in Pommerman have attracted many researchers to this environment. Their approaches can be broadly categorized into model-free RL [16, 17, 13, 14, 18] and tree-search-based-RL [19, 20, 11, 21, 12]. In addition, [22] is an excellent review of Pommerman, its practical implications, and its limitations. A comparison of search techniques including MCTS, breadth-first,

---

[2]There was no inter-agent communication in the NeurIPS 2018 competition, while two bits of information can be exchanged in each time step for the NeurIPS 2019 version.

and flat Monte Carlo [20] shows that in the fully observable FFA mode, MCTS is able to beat simpler and hand-crafted solutions. An extension of this study [19] called Rolling Horizon Evolutionary Algorithm (RHEA) concludes that the more offensive strategies (like RHEA with a high rate of bomb placing) are normally also riskier, due to inadvertent suicides[3]. One way around this is to perform tree search using pessimistic scenarios [21], and to choose actions that minimise the risk. Since the worst scenario can be deterministic, it can be rolled out efficiently. However, unrealistic or illegal scenarios can be generated and these have a detrimental effect on learning.

Studies that propose prediction of the movements of the other agents in addition to learning self policy [16] are based on the hypothesis that this would improve coordination in multi-agent scenarios. Continual learning [13] was used to train a population of advantage-actor-critic (A2C) agents in Pommerman, beating all other learning agents in the 2018 Competition. A Deep Neural Network (DNN) is updated using A2C in a process that allows the agent to progressively learn new skills, such as picking items and hiding from bomb explosions. Another Deep Learning approach is proposed by [14], which uses Relevance Graphs obtained by a self-attention mechanism. This agent, enhanced with a message generation system, analyses the relevance of other agents and items observed in the environment. Backplay [17] speeds up training by backtracking from the terminal states to the initial states of episodes, improving sample-efficiency. *Skynet* [18] trains deep neural networks using Proximal Policy Optimization (PPO). They have also implemented reward shaping and trained using curriculum learning paradigm. This the closest to our work, however, they do not employ imitation learning and train the network using PPO from scratch, which requires tremendous amount of training and compute. In [11], also later expanded in [12], the authors train a DNN using Asynchronous Advantage Actor-Critic (A3C) enhanced with temporal distance to goal states. They also integrate MCTS as a demonstrator for A3C, which helps reduce agent suicides during training via imitation.

3 Proposed approach

The problem can be modeled as a markov decision process, $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$, where $\mathcal{S}$ represents the partially observable state, $\mathcal{A}$ denotes the six actions, $\mathcal{T}$ represents transition probabilities and $\mathcal{R}$ denotes reward. Our focus in this paper is on a model-free approach and hence the transition probabilities are not modeled. A potential way of reducing the computational effort for training is to use off-policy sample-efficient algorithms such as DQN [1]. However, the partial observability, sparse reward structure, and long episode length (up to 800 steps) make it difficult to use experience replay for stabilising deep Q-learning. At the other end of the spectrum, simpler on-policy methods such as policy gradient are susceptible to high variance. Therefore we turn to methods based on the actor-critic architecture. Trust Region Policy Optimization (TRPO) [23] maximizes an objective function similar to vanilla policy gradient method, subject to a constraint on the size of policy update. However, TRPO needs a second order derivative to compute gradients and hence, is very computationally expensive. Proximal Policy Optimization (PPO) [24] achieves similar performance while relying on first order derivative and hence is more efficient. We use PPO in our approach.

3.1 State Space and Network Architecture

Pommerman environment provides observation in a dictionary in which, along with a board matrix of dimensions 11×11, we get other information such as the agents' bomb kicking capability, ammo, blast strength, IDs of two enemies and of the teammate at each time step. For our approach, we represent every feature as a separate 11×11 matrix which can be easily fed to a CNN. Categorical features such as items on the board are represented using a one-hot encoded matrix, whereas scalar features are populated as a full matrix. Apart from the raw information available in the input dictionary, we create one additional 11×11 input matrix representing the scalar *desirability* of each observable tile on the board (for example, an open passage tile is more desirable than a bomb). This matrix is intended to encourage the agent to move towards desirable and safe positions on the board. In total, we get 19 channels in the input, details of which are given in Table 1. Our network comprises of three convolution layers, each with max pooling and dropouts followed by two fully connected layers. The output consists of six units with softmax activation, one for each action (Fig. 2).

---

[3]This is also visible in one of our agents when trained in a raw manner without curriculum learning

Board representation	One channel each for one hot encoding of passage, rigid wall, wooden wall, bomb, flames, fog, extra bomb powerup, increase range powerup, kick powerup.
Position encoding	One channel each for the agent's own position, teammate's position and those of its enemies.
Powerup representation	A channel to broadcast the values of ammo, blast strength and binary kick capability.
Bomb life and strength	A channel to denote the blast strength and leftover lives of bombs placed on the board.
Safe/desired cells	The values of such cells are encoded as follows: Powerups=0, wooden wall=1, passage =2, Fog = 3, Enemies = 4, Rigid walls = 5, Teammate = 6, Bombs = 7, Flames = 8.

Table 1: State space representation

3.2 Training Setup

We train first using imitation learning, followed by reinforcement learning, as mentioned in the introduction. The details are provided below, in addition to other modifications to the reward function and action selection.

Curriculum: The total training effort was equivalent to 150,000 games. Of these, the first 50,000 games were played purely using SimpleAgent (a default heuristic provided with the environment, described in detail later in Table 2). State and action samples were saved for all four agents participating in each game. The network from Fig. 2 is trained in supervised fashion with cross-entropy loss with states and actions as data instances and labels respectively.

The imitation learning model acts as a policy network during the next training phase which uses PPO. A replica of the same network is created for value function estimation, with the output layer of size 1 instead of 6. We refrain from using the same CNN layers to approximate the value function as this creates aberration during the initial phase of learning and often leads to policy forgetting and degradation. We also avoid using any regularization technique such as dropout while training using PPO, as this leads to significant increase in KL divergence between trained policies.

The total effort with PPO is 100,000 games, played against agents of increasing sophistication (explained in Sec 3.3). We observed that training directly with the SimpleAgent or any other fully functional agent leads to forgetfulness of basic skills such as blasting wooden walls, picking powerups etc. In addtion, most drastic effect that leads to degradation of policies is learning to place bombs. This has been also observed in other studies [12, 15]. Training against agents with increasing difficulty helps retain the skills acquired in the imitation phase.

Reward Shaping: The credit assignment problem can be broken down into two aspects. Assigning credit between the agents in a team, and for a single agent, distribution of rewards for different actions. The latter can be solved using generalized advantage estimates with a normalizing factor. Although the method is noisy, we observed stabilisation over the course of training.

At the end of episode, we get only single reward for the team and it may not be clear how to assign credit to individual agents. For example, consider an episode where an agent eliminates an opponent but then commits suicide, and its teammate eliminates the remaining opponent. Under this scenario, both team members get a positive reward from the environment, but this could reinforce the suicidal behaviour of the first agent. Similarly, one agent could eliminate both opponents whereas its teammate
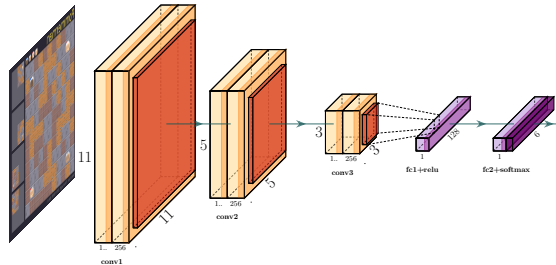

Figure 2: CNN Architecture for Policy

just camps; both agents would get positive rewards, reinforcing a lazy agent [25]. To solve the credit assignment problem within the team, we force the teammate to commit suicide at the start of each game (by placing a bomb and staying put until it explodes). Essentially, the game becomes 1-versus-2 and assigning the credit becomes easier. We provide a reshaped terminal reward signal as follows (at the risk of unintended changes in the policies [26]). Note that we do not infer whether enemies died due to the PPO agent killing them, or through inadvertent suicides.

- Reward is -1 if the game ends with both enemies alive (no success, whether tied or lost)

- Reward is 0.5, if at the end of episode there is only enemy agent alive (could be a loss or a tie, but at least one enemy was killed)

- Reward is 1 if the agent wins (both enemies dead)

Post-processing of selected actions: We veto the actions chosen by PPO in two cases, in order to improve the training efficiency. We call these post-processing rules as *jitter correction* and *action filter*, and their motivation and definition is given below.

Jitter Correction: A peculiarity of the policies trained through imitation on SimpleAgent is a tendency to alternate between the same two actions in successive time steps (for example, right and left). This jittery behaviour is also observed in SimpleAgent itself, and the imitation learnt policy attaches very high confidence (nearly 1) to these actions. Therefore, the jitter is also inherited by PPO during the initial RL phase even though PPO is a stochastic algorithm. The behaviour is particularly noticeable when no enemies are visible to the agent, leading to there being no obvious objectives to achieve. A possible solution would be to use momentum-based approaches such as n-step predictions. However, they are not tested with partial observations and dynamic state spaces [27, 28]. Instead, we include a mechanism of jitter correction to break the agent out of its loop (Algorithm 1).

Algorithm 1: Jitter Correction
---
xposition, yposition = empty list, empty list
while *not done* **do**
    append xpos, ypos to xposition, yposition
    static_cond1 = true if len(xposition[-15:]) ==1 else false
    static_cond2 = true if len(yposition[-15:]) ==1 else false
    x_cond_odd = true if len(set(xposition[-10::2])) == 1 else false
    x_cond_even = true if len(set(xposition[-11::2])) == 1 else false
    x_cond_uneq = false if len(set(xposition[-11::2]) - (set(xposition[-10::2]))) == 0 else true
    x_cond_long = true if len(set(xposition[-35:])) == 2 else false
    x_y_cond_long = true if len(set(xposition[-35:])) == 1 else false
    similar for y coordinate
    if *static_cond1 and static_cond2* **then** Take next 3 steps from expert policy ;
    else if *(x_cond_odd and x_cond_even and x_cond_uneq) or (x_cond_long and x_y_cond_long)*
      then Take next 2 steps from expert policy ;
    else if *(y_cond_odd and y_cond_even and y_cond_uneq) or (y_cond_long and y_x_cond_long)*
      then Take next 2 steps from expert policy ;
end

Action Filter: As observed in [12], there is a significant probability of an agent committing suicides at some point in the game, even with training. Avoiding this is particularly difficult because there are situations where the only way to avoid dying is to follow a long sequence of steps. We use a post-processing filter on the PPO actions in order to train efficiently (since the agent's death terminates the episode otherwise). This allows the agent to focus on higher level strategies. The PPO action is rejected if it is determined that the action would lead to death (for example, stepping into a bomb's path in the last time tick). Instead, any action apart from the PPO action and the bomb is chosen uniformly randomly. Given that the new action itself may be suicidal, the filter is applied until a safe action is found. A subtle difference between this approach and that of specifying the 'correct' action,

*_jitter	Removing jitter, where agent is either stuck on a single cell or is alternating between two cells.
*_action	Preventing suicidal actions, for example, whether the next action leads into a bomb path.
StaticAgent	Agent which does not move from initial position
SimpleAgent	Heuristic agent provided by the competition organizers
Imitation	Agent learned from the observations collected from SimpleAgent in a supervised setting
PPO	Agent trained using Curriculum learning with PPO , with warmup weights from imitation.
PPOAgent_Cautious	Agent trained with PPO with initial weights from imitation

Table 2: Nomenclature of agents

is that random choice allows for greater policy exploration. The Action Filter is implemented by rules shown in algorithm 2.

Algorithm 2: ActionFilter

---
act = agent.act(obs)
next state = get_next_state(obs, action)
while *next_state in flames or blast radius with bomb life remaining as 2* **do**
   | Restrict that action, take any random action from {right, left, top, bottom} - {act}
end

3.3 Experimentation

As outlined earlier, we begin the reinforcement learning portion of training with a policy network trained using imitation learning. However, the value network required for PPO does not reuse these weights. Instead, we freeze the policy network and train only the value network for 10,000 games against SimpleAgent (default heuristic provided by the developers). Following this, we train our model against three types of opponent teams with increasing sophistication. They are explained in Table 2. We start with 10,000 games against StaticAgent, which makes no moves whatsoever. This portion of training is used to learn how to approach and kill opponents by placing bombs near them. Next we train for 20,000 games against SimpleAgent, but without allowing it to place bombs. This helps the PPO agent learn how to follow and trap opponents, but restricting their bomb capability allows it to learn this skill quickly (by prolonging the games). Finally, we train for 60,000 games against the default SimpleAgent. The total training after imitation thus lasts 100,000 games. We have provided more detail about the rationale behind the curriculum in Sec. 4 and Fig. 5.

As the probabilities of the actions drawn from deterministic policies trained using imitation are very skewed, the entropy coefficient in the PPO surrogate objective has been kept to zero. Keeping it to the default value as mentioned in the original paper, leads to catastrophic forgetting and degradation of the learned skills. The PPO algorithm, like TRPO works on incremental updates in the policies and theoretical improves with respect to its previous policy. This provides a challenge while training, as Jitter Removal and Action Filter deviate from the pure PPO policy, and the resulting KL divergence between policies can be high. Keeping the higher threshold for KL divergence would also lead to degradation in policies, although that is also a function of batch size. Instead, we reduced the policy deviations with a probabilistic intervention: for each batch, only 10% trajectories had Jitter Correction active and 30% had Action Filter active. This provided stable learning and consistency in the observed KL divergence. We use 128 batch size and clip ratio of 0.01 during training.

We train two separate agents starting from the same imitation-learned policy,
1. PPO with curriculum, reward shaping, jitter correction and Action Filter termed as PPOAgent
2. Vanilla PPO without any intervention termed as PPOAgent_Cautious (for reasons explained later)
In the next section, we test our learned agents against various agents discussed above, to gauge the improvements over the initial imitation learning and the default heuristics.
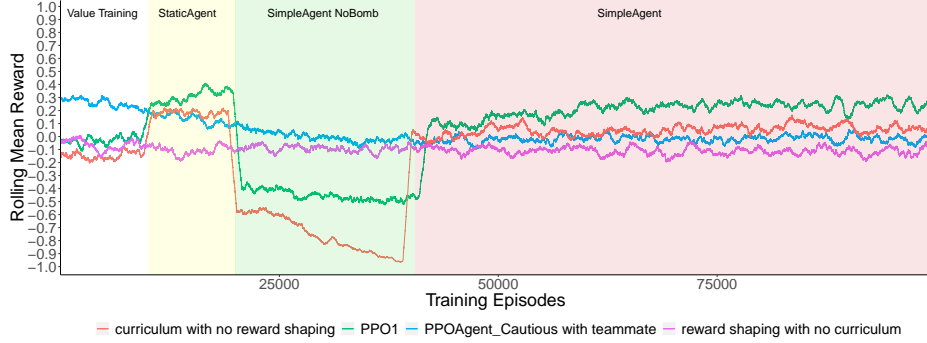
Figure 3: Reward during different training phases

4 Results and Discussion

We trained two agents for 100,000 games each: one (PPOAgent) with a curriculum of opponents, reward shaping, and post-processing of actions, and the other (PPOAgent_Cautious) without any interventions. Fig. 3 plots the evolution of training rewards for PPOAgent (green) and PPOAgent_Cautious (blue). Two more agents are also shown: one that plays against the same curriculum of opponents without reward shaping (orange), and another that uses reward shaping but no curriculum of opponents (pink). Note that (i) the y-axis is a 1000-episode rolling mean of the rewards, and (ii) PPOAgent_Cautious trains with its teammate (a SimpleAgent) active, so its initial reward is higher. Since the policy is invariant during the value training phase for PPOAgent, we know that this is the average reward for the imitation-learned policy against SimpleAgent. It is clear that the reward towards the end of training is higher than that for the imitation-learned policy, even though the plot only shows results with 10% jitter correction and 30% action filtering. The reward for PPOAgent_Cautious reduces before stabilizing, probably due to credit assignment issues (its teammate is also active). Learning is slow even when the teammate is terminated (as in the pink curve), while both curriculum-based agents (blue and orange) show significantly faster progress. The agent with no reward shaping sees a -1 reward for both ties and losses, which makes it difficult to learn (see deterioration against SimpleAgent_NoBomb). A visual rendering of both agents during gameplay shows that PPOAgent exhibits less jitter compared to its initial imitation-learned policy. Where it does enter a repetitive loop, it tends to do so in a finite area rather than just two neighbouring cells. This increases the probability of observing the enemy by accident, which breaks the loop. PPOAgent_Cautious learns to avoid placing the bomb at all (hence the nomenclature), even for breaking wooden walls. This restricts its movement to the initial quadrant. Most wins for PPOAgent_Cautious either due to an opponent committing suicide, or its teammate killing the opponents. Although it has a lower chance of accidentally dying than SimpleAgent, the learned policy returns very few wins.

Table 3 shows the performance of the agent trained using imitation learning on 50,000 games of SimpleAgent (initial policy used for PPO). The vanilla version uses the policy directly. Since we know that the policy is prone to jitter and to inadvertent suicides, we also test the policy augmented by one or both post-processing rules. These results act as a baseline for comparing the PPO results, which are given in Table 4 (including those between PPO and imitation). The PPO results also include performance against Skynet[4], which was the second best performing agent in the learning category in the 2018 NeurIPS competition. There are significant improvements over imitation, especially in the ratio of wins to losses. Furthermore, the PPO agent appears to win or tie 7 out of 8 games against Skynet. Fig. 4 explores the sensitivity of performance to the inclusion of jitter and action filters in each agent type. Specifically, we plot the change in wins, ties, losses (as a percentage of 1000 games) for the PPO agent against different opponents, when one or both filters are included. Jitter correction leads to fewer ties against all opponents, but increases both wins and losses. Action filter reduces losses against all opponents, but some of those losses are converted to ties. Using both jitter and action filters decreases losses as well as ties in all but one case, with more wins in all cases. In Sec. 3, we indicated that the curriculum of playing against gradually more difficult opponents leads to faster training than otherwise. In Fig. 5, we provide some intuition behind this claim. The plots on

---

[4]https://github.com/MultiAgentLearning/playground/tree/NeurIPS-2018-Docker-Agents

Opponents	Imitation_Vanilla			Imitation_jitter			Imitation_action			Imitation_jitter_action		
	Win	Lost	Tie	Win	Lost	Tie	Win	Lost	Tie	Win	Lost	Tie
StaticAgent	0.111	0.156	0.733	0.458	0.34	0.201	0.271	0.009	0.72	0.824	0.032	0.144
SimpleAgent_NoBomb	0.355	0.416	0.299	0.379	0.507	0.114	0.615	0.109	0.275	0.753	0.115	0.131
SimpleAgent	0.331	0.418	0.251	0.361	0.498	0.141	0.603	0.099	0.297	0.756	0.126	0.118
SimpleAgent_NoBomb_action	0.218	0.607	0.175	0.242	0.663	0.095	0.493	0.176	0.331	0.63	0.227	0.143
SimpleAgent_action	0.2	0.618	0.181	0.243	0.665	0.091	0.503	0.167	0.33	0.64	0.206	0.154
PPO_agent_Cautious	0.011	0.149	0.84	0.048	0.709	0.242	0.024	0.019	0.957	0.268	0.226	0.506

Table 3: Results in 1000 games for Imitation team (some games discarded due to fault after 12 steps).

Opponents	PPO_Vanilla			PPO_jitter			PPO_action			PPO_jitter_action		
	Win	Lost	Tie	Win	Lost	Tie	Win	Lost	Tie	Win	Lost	Tie
StaticAgent	0.179	0.138	0.681	0.614	0.253	0.132	0.347	0.003	0.65	0.904	0.023	0.073
SimpleAgent_NoBomb	0.373	0.366	0.260	0.425	0.446	0.128	0.583	0.081	0.335	0.778	0.111	0.111
SimpleAgent	0.347	0.379	0.273	0.426	0.45	0.123	0.622	0.079	0.298	0.778	0.088	0.135
SimpleAgent_NoBomb_action	0.230	0.586	0.183	0.271	0.641	0.086	0.507	016	0.333	0.681	0.19	0.129
SimpleAgent_action	0.260	0.537	0.202	0.282	0.615	0.101	0.521	0.167	0.312	0.672	0.186	0.142
PPO_agent_Cautious	0.007	0.138	0.855	0.067	0.69	0.239	0.026	0.011	0.963	0.313	0.126	0.423
Imitation_Vanilla	0.145	0.108	0.747	0.411	0.388	0.201	0.243	0.011	0.746	0.713	0.099	0.188
Skynet	-	-	-	-	-	-	-	-	-	0.451	0.126	0.423

Table 4: Results in 1000 games for PPO team (some games discarded due to fault after 12 steps).

the left are heatmaps of our agent's position while playing against different opponent types, and the plots on the right are heatmaps of bomb placement locations by our agent. All plots are aggregated over 50 games each, with our agent starting in the top left corner.

From Fig. 5a, we observe that most extensive exploration happens when playing against StaticAgent and SimpleAgent_NoBomb. This is because these two opponents are unable to leave their quadrants (cannot break wooden walls), which forces the PPO agent to hunt them down. On the other hand, the PPO agent can afford to be more conservative and wait for SimpleAgent or SimpleAgent_action to engage it, requiring lower exploration.

Fig. 5b shows similar behaviour, where bombs are placed in farther locations against StaticAgent and SimpleAgent_NoBomb (bombs in its own quadrant are used to break wooden walls). The two more sophisticated opponents require more nuanced strategy, including (as seen from graphically rendered games) multiple bomb placement to create traps. However, were the PPO agent not trained against the simpler agents first, the exploration and bomb placement tendencies learnt through imitation would be forgotten very quickly (as seen in PPO_Cautious).
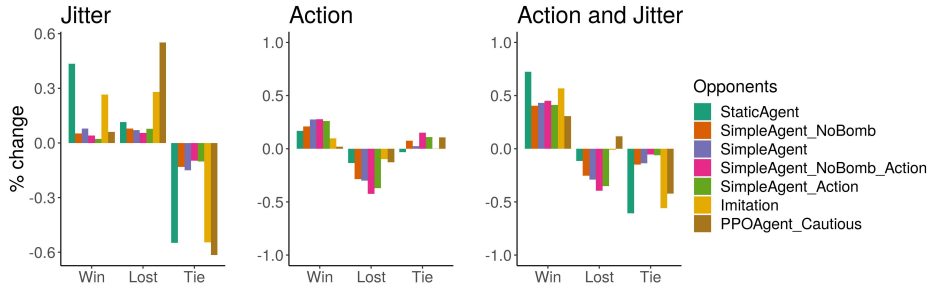

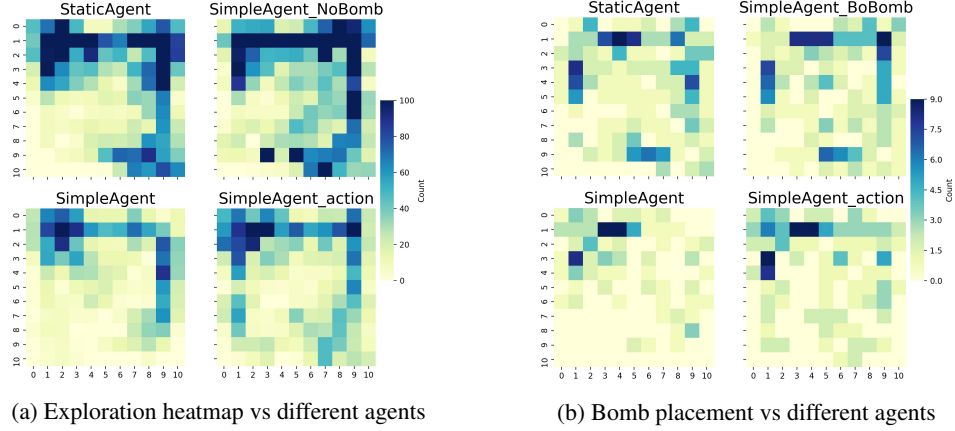Figure 4: Effect of including Jitter and actionfilter

(a) Exploration heatmap vs different agents (b) Bomb placement vs different agents

Figure 5: Comparison of characteristics against different opponent strategies.

5 Conclusion

We posit that the use of imitation followed by reinforcement learning is an effective way to reduce the training effort in Pommerman. Even if the expert policy for imitation is flawed, the agent is able to learn basic skills from it. Following this, reinforcement learning needs to be introduced gently (by training against simple opponents first) in order to retain the basic skills, while learning higher level skills against increasingly sophisticated opponents.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[2] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.

[3] J. Matas, S. James, and A. J. Davison, "Sim-to-real reinforcement learning for deformable object manipulation," in *Proceedings of The 2nd Conference on Robot Learning*, ser. Proceedings of Machine Learning Research, A. Billard, A. Dragan, J. Peters, and J. Morimoto, Eds., vol. 87. PMLR, 29–31 Oct 2018, pp. 734–743.

[4] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "Safe, multi-agent, reinforcement learning for autonomous driving," *arXiv preprint arXiv:1610.03295*, 2016.

[5] P. Mirowski, R. Pascanu, F. Viola, H. Soyer, A. J. Ballard, A. Banino, M. Denil, R. Goroshin, L. Sifre, K. Kavukcuoglu *et al.*, "Learning to navigate in complex environments," *arXiv preprint arXiv:1611.03673*, 2016.

[6] Y. Yu, "Towards sample efficient reinforcement learning." in *IJCAI*, 2018, pp. 5739–5743.

[7] L. Matignon, L. Jeanpierre, and A.-I. Mouaddib, "Coordinated multi-robot exploration under communication constraints using decentralized markov decision processes," in *Twenty-sixth AAAI conference on artificial intelligence*, 2012.

[8] V. Nanduri and T. K. Das, "A reinforcement learning model to assess market power under auction-based energy pricing," *IEEE transactions on Power Systems*, vol. 22, no. 1, pp. 85–95, 2007.

[9] F. A. Oliehoek, C. Amato *et al.*, *A concise introduction to decentralized POMDPs*. Springer, 2016, vol. 1.

[10] M. Minsky, "Steps toward artificial intelligence," *Proceedings of the IRE*, vol. 49, no. 1, pp. 8–30, 1961.

[11] B. Kartal, P. Hernandez-Leal, and M. E. Taylor, "Using monte carlo tree search as a demonstrator within asynchronous deep rl," *arXiv preprint arXiv:1812.00045*, 2018.

[12] B. Kartal, P. Hernandez-Leal, C. Gao, and M. E. Taylor, "Safer deep rl with shallow mcts: A case study in pommerman," *arXiv preprint arXiv:1904.05759*, 2019.

[13] P. Peng, L. Pang, Y. Yuan, and C. Gao, "Continual match based training in pommerman: Technical report," *arXiv preprint arXiv:1812.07297*, 2018.

[14] A. Malysheva, T. T. Sung, C.-B. Sohn, D. Kudenko, and A. Shpilman, "Deep multi-agent reinforcement learning with relevance graphs," *arXiv preprint arXiv:1811.12557*, 2018.

[15] C. Resnick, W. Eldridge, D. Ha, D. Britz, J. Foerster, J. Togelius, K. Cho, and J. Bruna, "Pommerman: A multi-agent playground," *arXiv preprint arXiv:1809.07124*, 2018.

[16] P. Hernandez-Leal, B. Kartal, and M. E. Taylor, "Agent modeling as auxiliary task for deep reinforcement learning," *arXiv preprint arXiv:1907.09597*, 2019.

[17] C. Resnick, R. Raileanu, S. Kapoor, A. Peysakhovich, K. Cho, and J. Bruna, "Backplay:" man muss immer umkehren"," *arXiv preprint arXiv:1807.06919*, 2018.

[18] C. Gao, P. Hernandez-Leal, B. Kartal, and M. E. Taylor, "Skynet: A top deep rl agent in the inaugural pommerman team competition," *arXiv preprint arXiv:1905.01360*, 2019.

[19] D. Perez-Liebana, R. D. Gaina, O. Drageset, E. Ilhan, M. Balla, and S. M. Lucas, "Analysis of statistical forward planning methods in pommerman," 2019.

[20] H. Zhou, Y. Gong, L. Mugrai, A. Khalifa, A. Nealen, and J. Togelius, "A hybrid search agent in pommerman," in *Proceedings of the 13th International Conference on the Foundations of Digital Games*. ACM, 2018, p. 46.

[21] T. Osogami and T. Takahashi, "Real-time tree search with pessimistic scenarios," *arXiv preprint arXiv:1902.10870*, 2019.

[22] D. Shah, N. Singh, and C. Talegaonkar, "Multi-agent strategies for pommerman."

[23] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*, 2015, pp. 1889–1897.

[24] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[25] L. Panait and S. Luke, "Cooperative multi-agent learning: The state of the art," *Autonomous agents and multi-agent systems*, vol. 11, no. 3, pp. 387–434, 2005.

[26] A. Y. Ng, D. Harada, and S. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," in *ICML*, vol. 99, 1999, pp. 278–287.

[27] A. S. Lakshminarayanan, S. Sharma, and B. Ravindran, "Dynamic action repetition for deep reinforcement learning," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[28] S. Sharma, A. S. Lakshminarayanan, and B. Ravindran, "Learning to repeat: Fine grained action repetition for deep reinforcement learning," *arXiv preprint arXiv:1702.06054*, 2017.