HIERARCHICAL DEEP REINFORCEMENT LEARNING AGENT WITH COUNTER SELF-PLAY ON COMPETITIVE GAMES

Anonymous authors
Paper under double-blind review

ABSTRACT

Deep Reinforcement Learning algorithms lead to agents that can solve difficult decision making problems in complex environments. However, many difficult multi-agent competitive games, especially real-time strategy games are still considered beyond the capability of current deep reinforcement learning algorithms, although there has been a recent effort to change this (OpenAI, 2017; Vinyals et al., 2017). Moreover, when the opponents in a competitive game are suboptimal, the current *Nash Equilibrium* seeking, self-play algorithms are often unable to generalize their strategies to opponents that play strategies vastly different from their own. This suggests that a learning algorithm that is beyond conventional self-play is necessary. We develop Hierarchical Agent with Self-Play , a learning approach for obtaining hierarchically structured policies that can achieve higher performance than conventional self-play on competitive games through the use of a diverse pool of sub-policies we get from Counter Self-Play (CSP). We demonstrate that the ensemble policy generated by Hierarchical Agent with Self-Play can achieve better performance while facing unseen opponents that use sub-optimal policies. On a motivating iterated Rock-Paper-Scissor game and a partially observable real-time strategic game (http://generals.io/), we are led to the conclusion that Hierarchical Agent with Self-Play can perform better than conventional self-play as well as achieve 77% win rate against FloBot, an open-source agent which has ranked at position number 2 on the online leaderboards.

1 INTRODUCTION

Deep reinforcement learning (RL) has achieved significant success on many complex sequential decision-making problems. (Silver et al., 2017) Most of the problems are in robotics domain (Levine et al., 2016) or video games (Mnih et al., 2015). However, complex real-time strategic competitive games still pose a strong challenge to the current deep reinforcement learning method due to the requirement of the ability to handle long-term scheduling, partial observability and multi-agent collaboration/competition. (Vinyals et al., 2017; OpenAI, 2017). Competitive games such as Go, in which each player optimize their own interests by finding the *best response* to opponents' strategies, are usually studied mainly on finding the *Nash Equilibrium* solutions (Silver et al., 2016; 2017), namely a combination of players' strategies upon which neither player can obtain higher rewards by modifying their strategy unilaterally (Shoham & Leyton-Brown, 2008). However, in the real-world, opponents can have a variety of strengths and play styles and do not always adopt the equilibrium solutions. In fact, human players are often remarkably good at analyzing strategies, tendencies, and flaws in opponents' behavior and then exploiting the opponents even if the resulting exploiting strategies themselves are subject to exploitation. Exploitation is a central component of sports and competitive games. This is also applicable in other real-world competitive domains, including airport and network security, financial and energy trading, traffic control, routing, etc. Therefore, exploring game-playing strategies that intentionally avoid the equilibrium solution and instead "learn to exploit" is a promising research direction toward more

capable, adaptable, and ultimately more human-like artificial agents. Hence, we develop a new algorithm Hierarchical Agent with Self-Play that learns to exploit the suboptimality of opponents in order to learn a wider variety of behaviors more in line with what humans might choose to display.

In this work, we focus on two-player, symmetric, extensive form games of imperfect information, though generalization to more players and asymmetric games is feasible and relatively straightforward. First, we adopt recent Proximal Policy Gradient (PPO)(**?**) methods in deep reinforcement learning (RL), which has been successful at handling complex games (Mnih et al., 2015; Silver et al., 2016; Mnih et al., 2016) and many other fields (Schulman et al., 2016; OpenAI, 2017) Second, we aim to automatically acquire a strong strategy that generalizes against opponents that we have not seen in training. Here we use self-play to gradually acquire more and more complex behaviors. This technique has proven successful at solving backgammon (Tesauro, 1995), the game of Go (Silver et al., 2017), imperfect information games such as Poker (Heinrich & Silver, 2016; Lanctot et al., 2017), continuous control (Bansal et al., 2018), and modern video games (OpenAI, 2017).

In this paper, we investigate a new method for learning strong policies on multi-player games. We introduce Hierarchical Agent with Self-Play , our hierarchical learning algorithm that automatically learns several diverse, exploitable polices and combines them into an ensemble model that draws on the experience of the sub-policies to respond appropriately to different opponents. Then, we show the results of some experiments on two multiplayer games: iterated Rock-Paper-Scissors and a partially observable real-time strategy game based on a popular online game generals.io (http://generals.io/). We show that compared to conventional self-play, our algorithm learns a more diverse set of strategies and obtains higher rewards against test opponents of different skill levels. Remarkably, it can achieve 77% win rate against the FloBot, the strongest open-sourced scripted bot on the generals.io online leaderboard.

2 Background and Related Work

2.1 Markov Games and Multi-agent Reinforcement Learning

Many real world multi-agent problems can be described by Markov games (Littman, 1994). A Markov game of N players at time step t has a full state s_t and assigns each player i ($1 \leq i \leq N$) an observation $o_{t,i}$. Then player i samples an action $a_{t,i}$ from its policy $\pi_i : \mathcal{O}_i \times \mathcal{A}_i \to [0, 1]$, where $\mathcal{O}_i$ and $\mathcal{A}_i$ are the observation and action spaces. Given all players' actions, the environment transits to a new state s_{t+1} and sends a reward $r_{t,i}$ to player i. The goal for each player i is to maximize expected total discounted rewards $J_i(\pi) = \mathbb{E}_\pi \left[\sum_{t=0}^{T} \gamma^t r_{t,i}\right]$, where $\gamma \in (0, 1]$.

A typical characterization of optimal strategies / policies is a Nash equilibrium, namely $\pi = (\pi_1, \ldots, \pi_N)$ satisfying $J_i(\pi) \geq J_i(\pi_1, \ldots, \pi'_i, \ldots, \pi_N)$ for any other π'_i. Finding equilibrium strategies can be very challenging, even for single-step (a.k.a. normal form) games (Shoham & Leyton-Brown, 2008). Most studies focus on either fully cooperative ($r_{t,i} = r_{t,j}\ \forall i, j$) or two-player, zero-sum ($N = 2, r_{t,1} = -r_{t,2}$) competitive cases. Methods for cooperative games include optimistic and hysteretic Q learning (Lauer & Riedmiller, 2000; Matignon et al., 2007; Omidshafiei et al., 2017), and recently centralized critic with decentralized actors (Foerster et al., 2017; Lowe et al., 2017).He et al. (2016) also shows the ability to find exploiting strategies; however it is in under relatively simple games and can be potentially combined with our method to have better performance. Panait & Luke (2005) and Busoniu et al. (2008) have comprehensive surveys on this topic. For two-player, zero-sum games, though small problems can be solved by linear programming, more complex (and symmetric) ones usually require self-play and some form of learning.

2.2 Self-Play

Self-play, namely training the agent against itself, is a powerful technique to bootstrap from an initially random agent. Classic game-theoretic techniques often offer provable convergence

to Nash equilibrium, and some also achieve success while combined with deep reinforcement learning. Such methods include fictitious play (Brown, 1951; Heinrich & Silver, 2016), counterfactual regret minimization (Zinkevich et al., 2008; Neller & Lanctot, 2013; Brown & Sandholm, 2017), replicator dynamics (Taylor & Jonker, 1978), double oracle (McMahan et al., 2003; Lanctot et al., 2017), and so on. The combination of deep learning, planning, and self-play led to the famous Go-playing agents AlphaGo (Silver et al., 2016) and AlphaZero (Silver et al., 2017). Most recently, self-play achieved success on full five versus five Dota 2, beating a team of 99.95th percentile Dota players (OpenAI, 2017).

2.3 Generals.io

Games are often popular environments for reinforcement learning research, and recently there has been a heavy interest (OpenAI, 2017; Vinyals et al., 2017) in real-time strategy (RTS) games. These games prove difficult for current methods due to the importance of long-term decisions as well as large action spaces. We propose generals.io (Generals) as an interesting and economical research environment that has many of the same challenges as Dota 2 and Starcraft II (SC2) while being very fast to simulate and having a sizable community of players online to evaluate against.

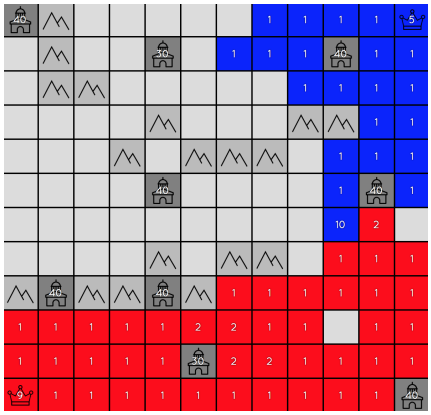
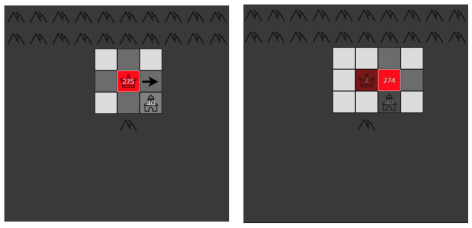
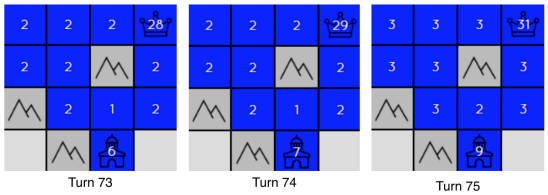


Figure 1: A full map of generals.io game. The grid with two mountains are the mountain grid that can not be go through. The grids with a building are the city grids. The grids with a crown are the generals grids. All the friendly grid are mark blue and opponent's grid are marked red. Neutral grids are marked with grey. The number is the amount of army staying in this grid.



(a) One action in the generals game. An agent needs to select which grid to execute and then perform an action to a one of the 4 adjacent grid. After each action, one of the arm will be left in the original grid and all the army will be moved to the new grid.



(b) The army aggregation scheme in the game. For each turn, the taken city and generals grids will generate one more army on the grid. After 50 turns, all the plain grid will increase 1 army.

Generals games in Fig. 1 take place on a rectangular grid. There can be anywhere from two to eight players at a time, although in this work we only consider the case of two players. Each player gets their own color, as well as a "General", and they cannot see any tile that is not adjacent to a tile they own. Moves happen every 0.5 seconds, and consist of each

player moving some army from one tile to an adjacent tile. Players can choose the grid to move and then move their army freely between tiles that they own, and if a player decides to move its army into a tile that is not their own color, the amount of army they are moving is subtracted from that tile and if the total is now negative, the player takes ownership. Fig. 2a demonstrates an example of how to move the army. There are also cities scattered throughout the map, which have a high cost to conquer. The army aggregation mechanism is described in Fig. 2b Every other turn, each player gets one army on their general as well as on every city they own. Every 50 turns, each player also gets an additional army on each tile they own. The goal of the game is to take ownership of the opponent's general.

The fog-of-war in Generals means that agents have to actively learn to seek out information. Agents also need to learn how to prevent themselves from being revealed, as well as learn to defend against incoming attacks, know when to strike, manage its own resources by expanding its own territory and investing in cities. Players online show a variety of strategies, including rushing to their opponent in the early game, staying small and hiding while investing in cities, or expanding and slowly suffocating their opponents by gaining slightly more resources than they have.

Using Generals also has several benefits compared to SC2. In Starcraft II game, a bot can easily have access to additional information that a human player cannot know and thus hard to evaluate the quality of a game agent. Another issue with SC2 is that in many games the bot can cheat by having extremely short reaction time which is impossible to achieve by a professional human players. Moreover, usually strategic games are very slow to simulate due to the large amount of information. However, Generals has a simple game board that is provided as a raw image (12x12) to both the agent and human players, as well as a small delay between turns that should lessen the effect of reaction time on the games. With our python version of generals that interfaces with OpenAI Gym (Brockman et al., 2016), we can achieve over 1000 frame per second with an average game time with a single GPU. This platform is considered more fexible and more transparent for evaluation. There are also resources available on the developer's website (http://dev.generals.io/), including an API to let an agent play online versus other bots or humans. They also have a collection of over 200,000 replays available as well as a Github repository full of parsing utilities as well as open-source bots online.

3 Method

In this work, we will have a high-level policy and a set of sub-policies to form a hierarchically structured agent. We will learn a series of sub-policies that can exploit one certain style in the game by self-play. The sub-policies must be strong because they are to be used by a High-level policy as a component. Moreover, the sub-policies must be diverse enough so that no one sub-policy dominates the rest. Then we train the high-level policy to choose from the sub-policies conditioned on observations. For both the sub-polices and the high-level policy in Generals, the observation includes a 12 by 12 map. In Section 3.1, we show the detailed architecture as well as PPO algorithm we use. In Section 3.2, we show how can we achieve the diverse and robust strategies that can exploit other strategies by using counter self play. And in Section 3.3, we show how to train the high-level policy.

3.1 Proximal Policy Optimization and Network Architecture

We adopt the Proximal Policy Optimization () algorithm in our training framework. PPO uses a surrogate objective which is maximized while penalizing large changes to the policy. The algorithm alternates between sampling trajectories from the policy and performing SGD on the sampled dataset to optimize this surrogate objective. Although the algorithm is not new, we need to come up with specific network architecture to learn the policy and the value function. The policy network is a fully convolutional network. The rationale behind this selection of network architecture is that for real-time strategic games, there are multiple similar events will happen in different spatial location. The fully convolutional architecture has the capability of maintaining translation invariance and of generalizing similar events

to different locations. We also share the initial 3 layers of parameters between the value network and the policy network.

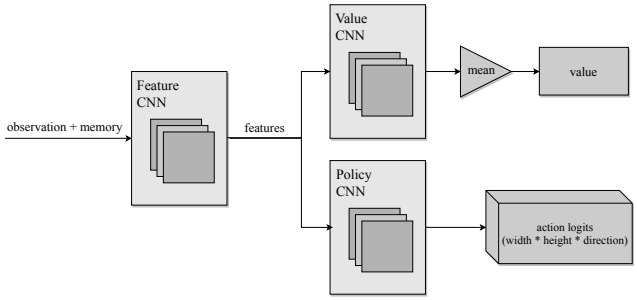


Figure 3: The architecture of the policy and value network. The policy network is a fully convolutional network. Different from regular games, generals has large action space. A policy need to first choose what grid to move and then what direction to move. Therefore, a fully convolutional network is necessary.

3.2 Counter Self Play

In this section, we describe how to achieve diverse and robust sub-policies by applying self-play. To learn a robust sub-policy by vanilla self-play requires occasional success by performing a series of random actions. But due to the large state space and action space, it is too challenging to win a game of Generals by chance on a large map. To alleviate this sparse signal problem in Generals, there are several methods such as engineering a dense reward for each step the agent take, imitating from demonstration replays as warm start or doing an exploration curriculum by adjusting the size of the map. We adopt the second method by imitating from demonstrations as a warm start. This method helps the sub-policy overcome the exploration problem, but it is still only rarely able to win against decent agents. A second problem is that if we start training a self-play agent, it will converge quickly to a local optima and stop generating diverse sub-policies which make this unusable for our high-level policy. In order to get a larger amount of diverse strategies we run conventional self-play with added an additional "achievement reward" as bonus for the initialization. However, we only use this to seed our method; after we get the initial policy, we only give all the agents the true rewards provided by the game and make the agent only want to win and win fast.

During Counter Self-Play, we only train against the most recent self to exploit it instead of training against a set of previous selves as in (Bansal et al., 2018). We also reinitialize the agent's policy each time we save our current sub-policy and start a new one. This way, each agent has a fixed number of training iterations and the main difference between the sub-policies will be "style" not "strength". The current agent will become the counter strategy to the previous agent. After we have enough sub-policies, we can stop training and start the learning of high-level policy. In previous work (Eysenbach et al., 2018), there is another way to generate diverse policies by maximizing an information theoretic objective. However, the sub-policies generated with our method have meaning in that they are best-responses to different opponents rather than just being diverse in the sense that they reach different states in state-space.

3.3 Hierarchical Agent with Self-Play

In this section, we describe how can we train a high-level policy based on the sub-policies we have. We initialize a parameterized high-level policy with random parameters. Then we train this agent with self-play as well. At each step in the game, the high-level policy will take the observation from the map as input and decide which sub-policy to choose. Different from prior work (), we fix the sub-policies during training the high-level policy. After choosing, the sub-policy is fed with the same observation and execute the action based on it for a single step. On simple games such as iterated Rock-Paper-Scissor, the high-level policy can

be computed once by first finding the pairwise expected payoffs of the sub-policies, and then solving a matrix game with Linear Programming as is the standard method in game theory.

Algorithm 1: Hierarchical Agent with Self-Play (HASP)

Initialize parameters of the sub-policies π_θ^i with the achievement rewards and imitation warm start described in 3.2, high-level policy π_ϕ, and value function v_ξ.
Initialize the training pool of policies $\Pi_{\text{train}} = \{\pi_\theta^i\}$
Sample initial policy π_θ^i from the training pool Π_{train}
for each epoch $e \geq 1$ **do**
    for each step $k \leq K$ **do**
        Player 1 and Player2 play the game with π_θ^i and collect Player 1's egocentric trajectories τ^k
        Update θ to θ' of Player 1 with PPO
    Synchronize both player's parameters $\theta = \theta'$
    Save current π_θ to the training pool.
The following can be replaced by Linear Programming
for each meta step $n \leq N$ **do**
    Player 1 and Player 2 play the game with π_ϕ whose actions are chosen from Π_{train}
    Player 1's egocentric trajectories τ^n
    Update ϕ to ϕ' of both Player 1 and Player 2 with PPO

4 Experiments

We test HASP and baseline methods on two discrete two-player competitive games: iterated Rock-Paper-Scissor (RPS) and Generals. [1] We want to answer the following questions with our experiments.

1. What types of sub-policies does our algorithm find?
2. Are the sub-policies from Counter Self-Play meaningfully diverse to be used by the high-level policy?
3. Can HASP lead to a stronger and more robust agent against opponents that are not seen in the training time compared with conventional self-play?

4.1 Baseline methods

For one baseline, we use conventional self-play, which is very similar to the way of how we get our sub-policies. To have a fair comparison with our method, we also incorporate the self-play agent with the imitation warm start stage described in section 3.2.

We also compare with the baseline that randomly chooses from the time-averaged past sub-policies from counter self-play (CSP) as a baseline to see how much improvement we get by training a high-level policy instead of just choosing a sub-policy randomly.

4.2 Iterated Rock-Paper-Scissor: A Motivating Example

Iterated RPS is formulated simply as Rock-Paper-Scissor repeated over multiple turns. We consider two turns for illustration purposes. An agent gets a reward +1 for winning more turns than the opponent, -1 for less, and 0 for equal turns. Though the game has an obvious equilibrium strategy – uniformly random at any turn, sub-optimal players may incline towards specific patterns. For example, some players take the counter action in turn 2 against the opponent's action in turn 1 (the "counter" strategy), under the assumption that the opponent does not change his action ("repeat"). Some may reason further and assume the

[1] Videos, codes and more information will be available at https://sites.google.com/view/hasp/home.

Table 1: The probabilities found after solving the empirical payoff matrix with sub-policies as strategies

sub-policies	1	2	3	4	5	6	7	8	9	10
	0.32	0.26	0.35	0.0	0.07	0.0	0.0	0.0	0.0	0.0

Table 2: Performance on iterated RPS. R = rock_ only, P = paper_only, S = scissors_only, c = "counter", c*i* = "counter *i*", min denotes the minimum performance along a row

winrate (%)	R	P	S	repeat	c	c2	c3	c4	min
HASP	49	47	50	49	52	54	48	48	47
Random	42	50	45	45	51	56	48	49	42
Self-Play	92	20	100	70	43	20	66	60	20
Self-Play	75	100	10	62	39	41	77	47	10
Self-Play	89	13	59	53	78	45	28	51	13

opponent using "counter", so chooses the counter to that ("counter2") instead. The reasoning continues until it loops back to "counter6" = "repeat". If our algorithm is successful and the initial policy is ("repeat"), it should autonomously discover all these sub-policies in the strategy space.

Running phase 1 of HASP here results in ten different policies. We provide a t-SNE projection of the learned policies onto the 2D plane below.

Since the action space of rock-paper-scissors is only three dimensional, we use the Linear Programming (LP) method to find a Nash-equilibrium mixed strategy where the moves consist of the ten learned sub-policies. The resulting policy samples a sub-policy with probabilities shown in Table 1.

In addition, we trained a baseline agent using conventional self-play. We found that using PPO to learn stochastic policies is quite hard, since most best-responses are pure strategies and when our agent reached low entropy, it would stop exploring, even if that policy was no longer good. In general, policies learned under self-play resulted were close to deterministic and reasonable at exploiting a large number of different policies. However, we found that they were highly exploitable.

Table 2 shows the performance of the different methods against some scripted test policies. Notice that the policy learned under HASP is less exploitable and therefore closer to an equilibrium policy than the different self-play runs. Note that randomly choosing a sub-policy also achieves a low exploitability, although not as low as with HASP .

4.3 Generals.io

For all of our Generals experiments, we initialize our agent with behavioral cloning on an open-sourced agent called FloBot. After behavioral cloning, our agent can get only an average of -1.7 reward against FloBot but is able to win rarely. While we include our final results against FloBot, note that they are slightly inflated in the self-play baseline, since the initial FloBot imitator is always in the training pool and therefore we learn a style that is effective against it. We find that when learning our our sub-policies, we can learn styles that are different from that of FloBot's.

In order to break symmetry between strategies, we added a small negative reward to each time step. This discourages our agents from playing a safe, defensive strategy, and encourages them to use the information they know about their opponents in order to end the game faster. The final reward that we use for training in Generals is:

Table 3: Reward on 12x12 Generals.io 1v1 mode.

reward	1	2	3
trained vs. all	-0.67	-0.87	-0.73
best sub-policy	**-0.45**	**-0.73**	**-0.58**

Table 4: Reward on 12x12 Generals.io 1v1 mode. expand_left, expand_right, and expand_more are all trained using self-play along with "achievement" rewards

reward achieved	expand_left	expand_right	expand_more	FloBot	min
HASP	**-0.63[0.04]**	**-0.44[0.02]**	**-0.33[0.08]**	-0.23[0.1]	**-0.63[0.4]**
Random-Ens	-0.64	-0.46	-0.36	-0.22405	-0.64
Self-Play	-0.75[0.11]	-0.64[0.11]	-0.34[0.04]	**-0.034[0.05]**	-0.77[0.08]

$$r(s, a, s') = \begin{cases} 1 & \text{win} \\ -1 & \text{loss} \\ -0.005 & \text{else} \end{cases}$$

We initialize the first phase of HASP with an agent trained to prefer owning territory on the top half of the map by adding the achievement reward described in Section 3.2. Running phase 1 of HASP , we obtained 6 different sub-policies that are diversely distributed in the sub-policy space. From observation, we find that the sub-policies are different from each other in terms of playing styles. We also find that, as we expected, each of them is trying to counter the previous self by a large margin, although it is exploitable itself.

We performed the same test of training an agent against all our sub-policies to see if our sub-policies are meaningfully diverse. Table 3 shows the performance of our agent trained vs. several sub-policies generated via CSP compared to the best response to those policies. Since the best-responses have the highest reward, we conclude that knowing ahead of time which opponent we are facing is valuable and therefore our strategies are meaningfully diverse.

For Generals, we trained our ensemble model using the Algorithm 1. The final usage frequencies (averaged over several games) of use of the sub-policies when facing another style of playing agent "expand_down" is shown in Table 5. Note that our agent learns to play a mixture of all strategies.

Table 5: The frequency at which our policy selects each sub-policy when playing against expand_left

sub-policies	1	2	3	4	5	6
	0.14	0.2	0.18	0.22	0.12	0.14

Table 4 shows the performance of our HASP agent as well as the baselines against several held-out agents that we trained using self-play with an additional "achievement" reward: expand-left, expand-right, and expand-more. The agent trained using HASP achieves a higher reward against all of the held-out agents that we tested on. In contrast with our Iterated Rock-Paper-Scissors results, randomly selecting a sub-policy at each turn is competitive with our policy that can change the probability of sampling a sub-policy with the current state. This suggests that the power of our method in Generals comes from the diverse sub-policies themselves.

5 Conclusions

In this paper, we investigate a novel learning approach Hierarchical Agent with Self-Play to learning strategies in competitive games and real-time strategic games by learning several opponent-dependent sub-policies. We evaluate its performance on a popular online game, where we show that our approach generalizes better than conventional self-play approaches to unseen opponents. We also show that our algorithm vastly outperforms conventional self-play when it comes to learning optimal mixed strategies in simpler matrix games. Though our method has achieved good results, there are some areas which could be improved in future research. In the future, we hope to also achieve good performance on larger versions of Generals, where games last longer and therefore learning is harder. We would also like to investigate further the effects that our algorithm has on exploration with sparse reward.

References

Bansal, Trapit, Pachocki, Jakub, Sidor, Szymon, Sutskever, Ilya, and Mordatch, Igor. Emergent complexity via multi-agent competition. *International Conference on Learning Representations*, 2018.

Brockman, Greg, Cheung, Vicki, Pettersson, Ludwig, Schneider, Jonas, Schulman, John, Tang, Jie, and Zaremba, Wojciech. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Brown, George W. Iterative solution of games by fictitious play. *Activity analysis of production and allocation*, 13(1):374–376, 1951.

Brown, Noam and Sandholm, Tuomas. Safe and nested subgame solving for imperfect-information games. In *Advances in Neural Information Processing Systems*, pp. 689–699, 2017.

Busoniu, Lucian, Babuska, Robert, and De Schutter, Bart. A comprehensive survey of multiagent reinforcement learning. *IEEE Trans. Systems, Man, and Cybernetics, Part C*, 38(2):156–172, 2008.

Eysenbach, Benjamin, Gupta, Abhishek, Ibarz, Julian, and Levine, Sergey. Diversity is all you need: Learning skills without a reward function. *arXiv preprint arXiv:1802.06070*, 2018.

Foerster, Jakob, Farquhar, Gregory, Afouras, Triantafyllos, Nardelli, Nantas, and Whiteson, Shimon. Counterfactual multi-agent policy gradients. *arXiv preprint arXiv:1705.08926*, 2017.

He, He, Boyd-Graber, Jordan, Kwok, Kevin, and Daumé III, Hal. Opponent modeling in deep reinforcement learning. In *International Conference on Machine Learning*, pp. 1804–1813, 2016.

Heinrich, Johannes and Silver, David. Deep reinforcement learning from self-play in imperfect-information games. *CoRR*, 2016.

Lanctot, Marc, Zambaldi, Vinicius, Gruslys, Audrunas, Lazaridou, Angeliki, Perolat, Julien, Silver, David, Graepel, Thore, et al. A unified game-theoretic approach to multiagent reinforcement learning. In *Advances in Neural Information Processing Systems*, pp. 4193–4206, 2017.

Lauer, Martin and Riedmiller, Martin. An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In *In Proceedings of the Seventeenth International Conference on Machine Learning*. Citeseer, 2000.

Levine, Sergey, Finn, Chelsea, Darrell, Trevor, and Abbeel, Pieter. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

Littman, Michael L. Markov games as a framework for multi-agent reinforcement learning. In *Machine Learning Proceedings 1994*, pp. 157–163. Elsevier, 1994.

Lowe, Ryan, Wu, Yi, Tamar, Aviv, Harb, Jean, Abbeel, OpenAI Pieter, and Mordatch, Igor. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems*, pp. 6382–6393, 2017.

Matignon, Laëtitia, Laurent, Guillaume J, and Le Fort-Piat, Nadine. Hysteretic q-learning: an algorithm for decentralized reinforcement learning in cooperative multi-agent teams. In *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, pp. 64–69. IEEE, 2007.

McMahan, H Brendan, Gordon, Geoffrey J, and Blum, Avrim. Planning in the presence of cost functions controlled by an adversary. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pp. 536–543, 2003.

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A, Veness, Joel, Bellemare, Marc G, Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K, Ostrovski, Georg, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

Mnih, Volodymyr, Badia, Adria Puigdomenech, Mirza, Mehdi, Graves, Alex, Lillicrap, Timothy, Harley, Tim, Silver, David, and Kavukcuoglu, Koray. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937, 2016.

Neller, Todd W and Lanctot, Marc. An introduction to counterfactual regret minimization, 2013.

Omidshafiei, Shayegan, Pazis, Jason, Amato, Christopher, How, Jonathan P, and Vian, John. Deep decentralized multi-task multi-agent rl under partial observability. *arXiv preprint arXiv:1703.06182*, 2017.

OpenAI. Openai dota 2 1v1 bot, 2017. https://openai.com/the-international/, 2017. Accessed: 2018-05-13.

Panait, Liviu and Luke, Sean. Cooperative multi-agent learning: The state of the art. *Autonomous agents and multi-agent systems*, 11(3):387–434, 2005.

Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. High-dimensional continuous control using generalized advantage estimation. *International Conference of Learning Representations*, 2016.

Shoham, Yoav and Leyton-Brown, Kevin. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press, 2008.

Silver, David, Huang, Aja, Maddison, Chris J, Guez, Arthur, Sifre, Laurent, Van Den Driessche, George, Schrittwieser, Julian, Antonoglou, Ioannis, Panneershelvam, Veda, Lanctot, Marc, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

Silver, David, Schrittwieser, Julian, Simonyan, Karen, Antonoglou, Ioannis, Huang, Aja, Guez, Arthur, Hubert, Thomas, Baker, Lucas, Lai, Matthew, Bolton, Adrian, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

Taylor, Peter D and Jonker, Leo B. Evolutionary stable strategies and game dynamics. *Mathematical biosciences*, 40(1-2):145–156, 1978.

Tesauro, Gerald. Temporal difference learning and td-gammon. *Communications of the ACM*, 38 (3):58–68, 1995.

Vinyals, Oriol, Ewalds, Timo, Bartunov, Sergey, Georgiev, Petko, Vezhnevets, Alexander Sasha, Yeo, Michelle, Makhzani, Alireza, Küttler, Heinrich, Agapiou, John, Schrittwieser, Julian, et al. Starcraft ii: a new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

Zinkevich, Martin, Johanson, Michael, Bowling, Michael, and Piccione, Carmelo. Regret minimization in games with incomplete information. In *Advances in neural information processing systems*, pp. 1729–1736, 2008.