Multi-Agent Reinforcement Learning with MuZero

Justin Deutsch
djustin8@vt.edu
Virginia Tech

Alex Wilson
apw@vt.edu
Virginia Tech

ABSTRACT

In recent decades computers have been showing their super-human intelligence in many different fields. AlphaGo and AlphaZero are two of the most notable examples that achieved this level of intelligence by playing with itself when training. This strategy is called self-playing and is used for many reinforcement learning algorithms. Deepmind has gone further on top of this idea and developed MuZero, which is a model-based and general-purpose algorithm. Meanwhile collaborative agents is another import RL research field that can potentially solve some very complex real-world problems and push the capabilities of computers to another level. One such example is OpenAI's model-free hide and seek algorithm which trains each agent individually but the agents learn how to cooperate with each other over time. Though there are many astonding breakthroughs like these in the reinforcement learning fields, there has been very little work done regarding multi-agent reinforcement learning using mode-based algorithms. Our program will use MuZero algorithm to achieve OpenAI's hide and seek's functionality for multi-agent reinforcement learning.

KEYWORDS

Multi-Agent Reinforcement Learning, Collaborative Agents, Self Play, MuZero

ACM Reference Format:
Justin Deutsch and Alex Wilson. 2021. Multi-Agent Reinforcement Learning with MuZero. In *Proceedings of Virginia Tech '21: Data-Centric Computing Capstone (VTCS ML Cap. '21).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

While machine learning is a wide field with applications in nearly all areas of life, one area of marked advancement over the past decade is the field of automated game playing through reinforcement learning. While reinforcement learning itself is nothing new, a series of architectural advances have created new and more efficient approaches for training agents on ever more complex tasks; the culmination of this research has been MuZero, which purports to be able to learn any game based on observations and win-loss feedback alone. [1]

However, while the MuZero algorithm has already been examined for classic one and two player games[1], its original authors did not adapt the algorithm for games of more than two players. Multi-agent reinforcement learning is a rapidly developing field, and MuZero succeeding in that arena could represent a significant advancement on the state of the art; thus, we set out here to explore ways to adapt MuZero's success to a team-based multi-agent environment.

1.1 Problem Definition

We have created a multi-agent environment using the PettingZoo Framework's multi-agent battle scenario [2]. In this environment, two teams of six participate in a battle where the objective is to attack agents on the other team without being hit. In general, the reward structure recognizes hitting enemies, scoring kills, winning rounds, losing rounds, dying, and being hit; the former three generally produce rewards, and the latter three penalties. We experimented with several different reward structures and found that these had significant impacts on the convergence properties of the model.

By design, the battle scenario lends itself well to team-level strategy; agents have limited health but slowly regenerate it, encouraging targeted many-on-one attacks against individual agents. Since agents cannot move and fight on the same turn and all agents have the same base speed, a single battler cannot prevent a wounded enemy from escaping; Thus, we anticipated that the most successful strategies would involve team members working in concert to 'trap' isolated enemies and eliminate them without risk of retaliation.

Our goal is twofold. First, we seek to adapt the existing MuZero algorithm from one- and two-player games to games with an arbitrary number of players on two teams. This will increase the number and types of games to which the algorithm may be applied. Second, we wish determine how quickly and how effectively agents running MuZero can converge to collaborative behavior in the scenario described above, if they converge at all.

1.2 Motivation

The applications of cooperative/competitive multi-agent reinforcement learning are myriad, ranging from the mundane (such as superhuman performance in multi-player poker) to the exotic (interpreting and navigating human social landscapes.) [3] Given the complexity of the environments, much study remains to be done in this area; in particular, adding agents to an environment is notoriously difficult to scale and generally causes a superlinear increase in the difficulty of training.

While MuZero's size somewhat increases its computational overhead compared to prior methods, its broad adaptability and use of a dynamics function (see 3.2) have allowed the algorithm to

[1]eg. Go, Chess, the Atari Suite

converge much more rapidly for one and two player games [1]. If these gains can be adapted to multi-agent reinforcement learning (MARL), the field could see a significant reduction in training time and improvement in model adaptability stemming from MuZero's architecture.

2 RELATED WORK

2.1 Literature survey

2.1.1 Emergent Tool Use From Multi-Agent Autocurricula. In *Emergent Tool Use From Multi-Agent Autocurricula*, OpenAI researchers sought to train two teams of agents to undertake a hide-and-seek task in a complex environment. [4] Agents were placed in a 3D environment with both movable and immovable obstacles; The movable obstacles could be 'picked up' by agents or locked in place to prevent agents on the other team from moving them.

The defining factor of the hide-and-seek task was the fact that the agents were divided into two teams of two, each with different goals. The seekers were rewarded for achieving line-of-sight on the hiders, while the hiders were rewarded for avoiding the vision of the seekers. This zero-sum game meant that each team had to optimize for a moving target; the resulting 'arms races' are referred to as *autocurricula*.

The agents were trained using Proximal Policy Optimization (PPO), a policy gradient ascent technique using a 'surrogate objective' function for optimization. [5] This training was augmented by Generalized Advantage Estimation (GAE), a specialized policy gradient estimator well suited for continuous motion tasks in three dimensions. [6] Data was then fed to these algorithms through a sophisticated partial-information system making use of Long-Term Short Memory (LSTM) and calculating knowledge based on line of sight.

This methodology met largely with success. Agent behavior progressed through a number of distinct 'phases', each representing some fundamental strategic advancement. For instance, one phase began when the hiding team learned how to construct shelters from mobile objects in the arena; the seeking team soon learned to respond by using ramps to scale the shelter walls, and the hiders responded by learning to lock ramps to prevent the seekers from using them. Both teams thus demonstrated a number of emergent behaviors demonstrating the use of both environmental objects ('tools') and their teammates; several of these strategies would have been impossible for any single agent within the time provided.

One notable point on which our approach differed from that in *Multi-Agent Autocurricula* was reward structure. *Autocurricula* rewarded its agents only as a team; a penalty received by one agent was received by all of them, enforcing cooperation as a direct consequence of the scenario. *Autocurricula* also made use of very large batch sizes and a long training period, which may explain the strength of their results relative to ours.

2.1.2 Human-level performance in 3D multiplayer games with population-based reinforcement learning. In *Human-level performance in 3D multiplayer games with population-based reinforcement learning*, DeepMind researchers applied population-based reinforcement learning an environment based on the fully-3D video game Quake and its Capture The Flag (CTF) mode. [7] In this task, agents had to move through a pseudo-random 3D environment that changed each game; two teams began on opposite sides of the map and had to retrieve a flag located near their enemy's start. The scenario gained an adversarial element because agents could 'kill' enemy teammates by tagging them with a built-in laser; thus, the agents had to divide their attentions between defending a friendly flag, approaching an enemy flag, and evading enemies while carrying their flag.

This task is particularly relevant to this paper, as its goals and incentive structure bear striking similarities to our own. Like our defined scenario, agents must balance collaborative and adversarial strategies; specifically, in order to have any chance at their actual goal they must individually coordinate with other agents. Agents have their perception limited to their immediate surroundings and must cooperate based not on direct communication but by perceiving opportunities to do so; the paper's authors refer to this as "ad-hoc teamwork". However, there are some critical implementation differences arising from the complex, full 3D environment.

While our implementation receives direct, noiseless observations from the environment, the agents in the paper were given only RGB inputs corresponding to what a typical player would see on the screen. The model was trained using a policy-gradient method (as above) with several RNNs operating on different timescales to provide both fast 'reflex' actions and longer-term strategic thinking. Critically, the authors perform reward evaluation on both score (game goals) and winning/losing (metagame goals). The former was trained in using conventional RL, while the latter used a population-based training technique; many agents were trained in parallel, with the worst-performing being replaced with mutated versions of the best.

While DeepMind's scenario dwarfs our own in both complexity and computational resources, some of their lessons learned can nonetheless inform our own decisions. In particular, their choice to train many independent agents with population-based training is a potential modification to our present architecture, which relies on many copies of just one.

2.2 Limitations of Existing Approaches

While successful, the approaches we've examined here are united in their enormous computational burden. *Emergent Tool Use* produced striking results in a single-model context, but needed nearly 500 million episodes to converge upon their final equilibrium. *Human-level performance* played 500,000 games, but at considerably higher computational cost per game; CTF games were long and used a different recurrent model for each player, all of which had to be trained independently. "Ad-hoc teamwork" is a worthy goal, but present implementations can require weeks of training and hundreds of dollars to achieve it.

In this, we see a niche for an algorithm that can produce rudimentary versions of these behaviors in a (comparatively) small number of (comparatively) simple games. Adapting MuZero into such an algorithm is the purpose of our research.

3 PROPOSED APPROACH

3.1 Techniques and Changes

Before we discuss the ways in which we have changed MuZero to make it compatible with MARL, we will briefly examine the algorithm's structure and the way that data flows during normal operation.

MuZero's notability as an algorithm stems from its ability to simulate a game environment rather than being provided with one. In a traditional reinforcement learning context[2], the goal of the algorithm is to train some policy network $\pi : \mathcal{S} \to \mathcal{A}$ mapping the present game state to an action such that the total reward produced is maximized. In this framework, the model must receive:

(1) The state of the game based on the input provided.
(2) The effect of some action $a \in \mathcal{A}$ on the current state.
(3) The reward given by the environment for that action.

This information naturally must be provided to the algorithm, generally through a perfect game model. For example, a chess-playing algorithm is initially given a perfect representation of a board state. For any action a, it can (by consulting a programmed chessboard analyzer) exactly predict the state of the board after a, as well as if this state represents a win or a loss. This provides robust learning capability, but only for games which can be exactly modeled in code.

MuZero cannot make direct assumptions about the game state's rules, so it instead learns to predict these using one *initial inference* and one or more *recurrent inferences*. MuZero makes these predictions by augmenting the policy network π with two new pieces, a dynamics function g responsible for the initial inference and a representation function h responsible for the reccurent inferences. The representation function $h : O \to \mathcal{S}$ learns to map a raw observation to some hidden state $\mathcal{S}$ representing the game state, and the dynamics function $g : \mathcal{S} \times \mathcal{A} \to \mathcal{S} \times \mathbb{R}$ maps the current state representation plus some action to the next state representation and the estimated reward given by that action to the agent. By training these models, the algorithm learns how the game works at the same time that it learns strategy, allowing it to play games without being 'taught' the rules.

In implementation, the training proceeds in three distinct stages that run in parallel. The Self-Play module uses the most recent version of the model, acquired from the Shared Storage, to play games against itself, using its own policy to select the best move for every player. These games are sent to the Replay Buffer module, which is mostly used to store and batch individual episodes for training; however, it has a secondary functionality called Reanalyse, which uses the latest version of the model to recalculate training statistics for old games, creating novel insights for training at a reduced performance cost. Finally, the Training module collects the played games in batches from the Replay Buffer module and makes adjustments to the model. These updates are then stored in the Shared Storage module so that Self Play can use them for later games.

In order to allow for MARL, we had to make a series of changes to various parts of the algorithm. While most of these changes were merely to tweak invariants or adapt the code to the training environment, the most significant changes to the architecture itself are as below:

[2]Such as that used by MuZero's precursor AlphaGo

3.1.1 Monte Carlo Tree Search. The first significant architectural change needed to make MuZero MARL compatible involved the Monte-Carlo Tree Search algorithm used by Self-Play. In order to choose an action in its self play, the algorithm must 'think ahead' to predict which action will maximize its expected future reward. It does this using Monte-Carlo Tree Search (MCTS). In MCTS, the game tree is stochastically expanded with priority given to moves the model perceives to be better. In testing, the best move will be chosen for play, while training includes an 'exploration bonus' to encourage the development of unfamiliar paths and hence new strategies.

The game tree format works well for a two player game, but proves unwieldy for one with 12 individual agents in it. In the worst case, the search algorithm will become lost in the 21^{11} possible combinations of other-agent moves and fail to expand even one second action for the agent to move, effectively robbing it of its ability to plan ahead.

We attempted to redress this using a simplification based on a detail of our environment. While all twelve agents must move in a turn, these moves are processed *simultaneously*; that is, the environment updates exactly once between any agent's first move and their second. From the perspective of a single agent, the order of the other agents' moves doesn't matter; in fact, given the black-box nature of MuZero environment parsing, the other agents' moves might as well be a feature of the environment itself. Thus, a sufficiently empowered dynamics network could predict the moves of the other agents as part of its normal state-transition calculations, allowing the singular agent to pretend that it was playing a complicated yet single player game.

The main actual impact this strategy has had on the code has been to cause MCTS to treat our many-agent scenario as a single-agent scenario. The versatile nature of the dynamics network allows it to approximate the next state without having to completely expand the game tree.

3.1.2 Self Play. Pursuant to the above changes, we have altered the mode of operation of Self-Play itself. In the standard implementation of MuZero, games with more than one player would be self-played and stored as one game. The standard MuZero player would accrue reward and seek to maximize total reward, while the notional 'human player' (also being played by MuZero) would incur negative reward and seek to minimize total reward. However, given that this no longer reflected the state of the game tree, to ease training we modified the way that saved games ('episodes') are stored.

Rather than saving one episode of six agents for a given team, Self-Play saves six episodes of one agent each. Each agent's end-of-game reward is determined by whether or not its team won, but otherwise, the rewards and state values are left as calculated by the existing model. The trainer will proceed in each of these twelve episodes as if it was training the model to play a single-player game; despite this radical departure from the nature of the environment itself, MuZero's sheer adaptability should allow the model to converge regardless.

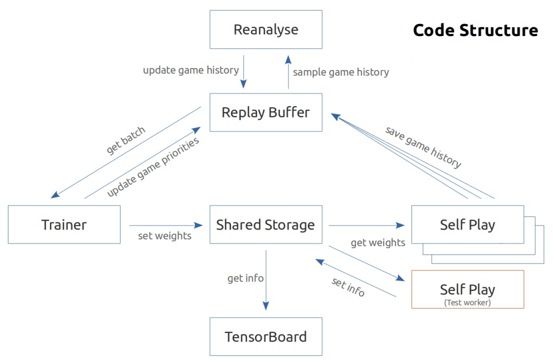

Figure 1: Code structure; from [8]

3.1.3 A-B Models. A final optimization we made to aid the learning process is to carry out self play between two different versions of the model; specifically, one of the two teams is played by the current best model and one by a version of that model several training steps behind. By splitting the model into alpha and beta versions and playing them against each other, we aimed to break the symmetry between the teams and reinforce emergent strategies that counter the model's current policy.

3.2 Code Structure

Our final code structure is typical of a reinforcement learning approach using self-play, as seen in Figure 1. A series of threads continuously generate game histories, which are buffered for analysis and training; a separate trainer thread semi-randomly samples individual moves from these histories for training, using 'Prioritized Experience Replay' or PER to preferentially sample moves that are 'interesting' or low-probability. [9] The trainer stores the new weights in a storage buffer for validation and access by the self-play thread, thus restarting the process.

Concurrently to this two-step process, a reanalysis thread recalculates value estimates for moves in old games to improve their relevance to current training. This helps to prevent updates to the value and reward networks from being drowned out over time.

4 EXPERIMENTAL EVALUATION

4.1 Methodology

Traditional metrics for measuring model behavior proved inadequate for the needs of multiagent MuZero. As training is performed on batches of individual moves (rather than entire games), reward graphs proved ineffective at measuring model progress; in fact, per-game reward actually decreased over time in the most successful training cycles as more sophisticated short-term strategies (tactics) forced the other team to take more penalties.

Due to constraints on development time and resources, evaluation was instead performed by inspection: a script was used to generate animated GIFs of games at regular intervals, which were then examined and categorized based on the model's dominant strategies.

4.2 Findings

In training the model with various hyperparameters, we found that several distinct strategies emerged to capitalize on the environment's reward structure. Even so, in our view none of these fully grasped the complexity of the environment.

As discussed later in this section, we saw a progression of behaviors as we responded to results and made updates to the training process and reward structure. Snapshots from generated GIFs highlighting the different behaviors are shown in Figure 2. Overall, the model learned and unlearned a variety of behaviors throughout training much like the emergent behaviors seen by OpenAI. [4] In their work, behavioral eras would potentially last for hundreds of thousands to millions of training steps. Due to our limited computing resources and wanting to evaluate adjustments more frequently, we opted to end training sessions after roughly 30,000 training steps or when the model's behavior remained consistent and graphs such as loss and average reward were relatively flat for a considerable time. Therefore, we do not know if the model could have further progressed out of these patterns, but we believe in many of these the model converged on a sub-optimal policy.

Across all the experiments, we allowed the environment to progress for 250 steps before terminating it and when planning in the MCTS, 25 steps were simulated. Unless noted in the experiments, the reward values were left as the defaults provided by PettingZoo. Fully Connected Networks were used with the representation network being 3 layers of size 16 and the remaining networks being 4 layers of 32 nodes. The representation network would encode the observation down to an encoding size of 8. All models were optimized using the Adam optimizer with the weight decay set to 10^{-4} and the momentum as 0.9. The loss values were calculated using cross-entropy loss. The same optimization pattern as described in the paper and general implementation was used.

4.2.1 Clustering. In an initial attempt to train the model, it rapidly converged to clustering all agents to a single location within the environment. There was a progression of behavior in which the agents attacked one another but over time stopped. This transition could be due to the model associating the negative reward of being hit with the attacking as well because when successful, the attack will generate an example of a positive and negative rewards because observations from both teams were used as training data.

This model was trained with a batch size of 1024 and a replay buffer size of 1000. The learning rate for all models was initialized to 0.02 with a 0.9 learning rate decay factor taking affect every 1000 training steps.

4.2.2 Hyper Aggression. In response to the model unlearning to attack opponents, we increased the positive reward for hitting an opponent to 0.5 from 0.2. This was done to further incentivize the model to attack. The batch size was also increased to 2048. The larger batch sizes provides the model with more information about the environment and can lead to more general updates.

After these updates, the model began to constantly attack, even when the attacking agent wasn't near an enemy; we termed this phase of behavior 'Hyper Aggression'. The clustering pattern persisted as well leading to all the agents grouping in one area and relentlessly attacking one another. Because the attack reward was

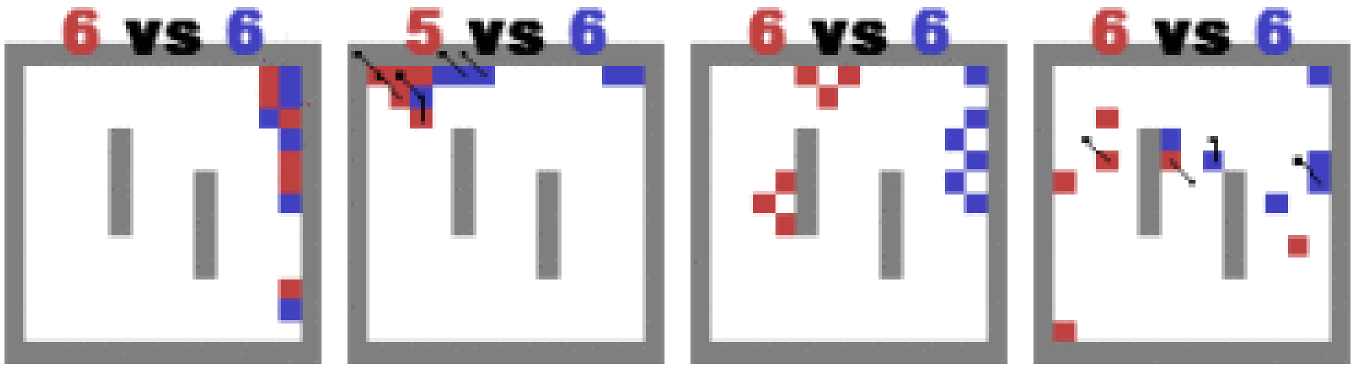

Figure 2: Graphics displaying behaviors in the environment. From left to right: clustering, hyper aggression, doing nothing, randomness.

significantly more and the pattern led to kills which come with an even greater reward, the model now over-valued attacking over the negative rewards of being hit or dying. Unfortunately, an error occurred shortly into training that resulting in the experiment crashing. The model would have had the chance to improve further and learn to optimize this strategy. Brief experiments where the attack reward was decreased but still remained above 0.2 should more signs of meaningful behaviors but generally converged to clustering.

4.2.3 Doing Nothing. Through a series of changes, the training process implemented the A-B models discussed previously in hopes of providing a gap between the two models' behaviors and to combat clustering. Additionally, new maps were added to the training script to provide a rotation and diversity. This was also done in hopes of negating clustering and also forcing the model to be responsive to its environment. The attack reward was set to 0.35 as it was deemed to best combat clustering and encourage attacking, and the learning rate decay was removed with the learning rate set to a constant 10^{-4} to provide more stable progress. The batch size was decreased to 512 to provide faster training, and to address less general updates, the replay buffer size was increased to 1,750.

After not seeing new patterns emerge from these changes, the overall step penalty of -0.005 was removed to further encourage attacking and also provide clearer signals to the model of the association between actions and their rewards. The model responded to these changes by learning to do nothing. Agents simply remained in a stationary position. Watching videos of the model in the environment, this behavior pattern may be a derivation of clustering in which the agents are choosing to move only in one direction and are being blocked by something giving the appearance of doing nothing.

4.2.4 Randomness. To encourage some behavior beyond doing nothing or clustering, all negative rewards were removed from the environment on top of the step penalty. This means that the model can only receive positive rewards from successfully attacking or killing an opponent. Furthermore, the batch size was increased again from 512 to 1000 to provide those more general updates.

Viewing the model in the environment, there does not appear to be any conceivable pattern to the agents' behaviors. It is promising to see that all actions are being taken moving and attacking in all directions. However, this is where future work is needed to add additional metrics to the environment. Without further metrics being recorded, behavioral patterns cannot be identified or justified in this experiment.

5 FUTURE WORK

While the model has demonstrated its ability to produce novel behaviors based on the environment's reward structure, as discussed above we did not find direct evidence of strategic or tactical coordination. Potential causes of this issue may include weaknesses in the algorithmic components and MCTS implementation, training issues of scale and hyperparameter choice, and problems in the environment design preventing the development of a single robust strategy.

A major assumption was made when adjusting MuZero's original MCTS for multi-agent environments. Bundling all other agent's actions into a single step in the tree requires the network to be very effective when simulating the environment. It is believed that this assumption may be one of the main causes for MuZero's subliminal performance. Monte Carlo Tree Search implementations exist that account for many other agents. To be compatible with multi-agent environments, MuZero's MCTS may need to be adapted to one of these versions. [10]

Development was also frequently blocked by model performance, as training would take hours before results could be seen. This led to issues in other areas as well such as experimenting with hyperparameters and other training configurations. MuZero having 5 different networks within it means it is best trained at scale especially on a problem of this complexity. Work must be done to scale the training further. This would be best achieved by moving the training to a cluster or remote service such as AWS.

Due to the significant time needed to see results in training, hyperparameters could not be adequately tuned. Each adjustment required a few hours worth of training to review the affects. Some hyperparameters that should be targeted in future training include the encoding space size, batch sizes, network architectures, and network sizes. Throughout training, the encoding space size was been set to 8. The representation network is responsible for encoding an observation down to the encoding space size. With observations being 7x13x13x41, that is a significant reduction in information. Increasing the encoding space size may allow for better results from the representation network which feeds the rest of the model. Additionally, increasing the batch size allows the model to produce more general updates. Larger batch sizes have produced the best results, but the training time also increases. The framework also provides the ability to change the network architectures between Fully Connected and Residual Networks. Only Fully Connected networks have been used, and their sizes have remained the same at 3 or 4 layers of 32 nodes each. Overall, more work must be done training with a variety of different parameters.

6 CONCLUSION

While some work remains outstanding, particularly with regards to the reward structure in use for our chosen scenario, our work to date is at least a proof of concept for wider deployment of the MuZero framework in team-based multi agent reinforcement learning. While the model has frequently produced degenerate strategies, it has also produced more promising behaviors recently could advance towards a more optimal strategy for the environment.

Moving forward, we plan to address issues with the reward structure and investigate a number of avenues for improved game-playing ability. We're also investigating ways to improve the model's ability to generalize by having A-B models, training on different maps, and researching more advanced MCTS implementations.

REFERENCES

[1] Julian Schrittwieser et al. "Mastering Atari, Go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (Dec. 2020), pp. 604–609. ISSN: 1476-4687. DOI: 10.1038/s41586-020-03051-4. URL: http://dx.doi.org/10.1038/s41586-020-03051-4.

[2] Justin K Terry et al. "PettingZoo: Gym for Multi-Agent Reinforcement Learning". In: *arXiv preprint arXiv:2009.14471* (2020).

[3] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. *Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms.* 2019. arXiv: 1911.10635 [cs.LG].

[4] Bowen Baker et al. *Emergent Tool Use From Multi-Agent Autocurricula.* 2020. arXiv: 1909.07528 [cs.LG].

[5] John Schulman et al. *Proximal Policy Optimization Algorithms.* 2017. arXiv: 1707.06347 [cs.LG].

[6] John Schulman et al. *High-Dimensional Continuous Control Using Generalized Advantage Estimation.* 2018. arXiv: 1506.02438 [cs.LG].

[7] Max Jaderberg et al. "Human-level performance in 3D multiplayer games with population-based reinforcement learning". In: *Science* 364.6443 (2019), pp. 859–865. ISSN: 0036-8075. DOI: 10.1126/science.aau6249. eprint: https://science.sciencemag.org/content/364/6443/859.full.pdf. URL: https://science.sciencemag.org/content/364/6443/859.

[8] Aurèle Hainaut Werner Duvaud. *MuZero General: Open Reimplementation of MuZero.* https://github.com/werner-duvaud/muzero-general. 2019.

[9] Tom Schaul et al. *Prioritized Experience Replay.* 2016. arXiv: 1511.05952 [cs.LG].

[10] Nicholas Zerbel and Logan Yliniemi. "Multiagent Monte Carlo Tree Search". In: *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems.* AAMAS '19. Montreal QC, Canada: International Foundation for Autonomous Agents and Multiagent Systems, 2019, pp. 2309–2311. ISBN: 9781450363099.