Comparing Unity ML Agents' RL algorithms for training multi agent non-zero-sum adversarial games

Alberto Giudice[1], **Andrea Trevisin**[1], **Fabio Scita**[1]
[1]IT University, Copenhagen 2300, Denmark
email: algi@itu.dk, atre@itu.dk, fasc@itu.dk

Abstract

Unity ML-Agents is a powerful tool that allows developers to easily train agents with machine learning. The main use case it's designed for is training of a single agent within a consistent environment, using the PPO or SAC algorithms. A third algorithm, MA-POCA, allows for training multiple agents concurrently. In this paper we compared using each of these 3 algorithms for training two agent types, either separately or together, in a non-zero-sum adversarial game simulation. Our results show that MA-POCA, by training the two agents concurrently, leads to more balanced behaviors, while training one agent at a time can incur the pitfall of learning extremely specialized behaviors that cannot adapt to certain changes in the environment.

1 Introduction and problem statement

Reinforcement learning (hereon shortened as RL) is a branch of machine learning that involves one or multiple agents that repeatedly interact with an environment (and possibly each other), and aims to learn policies that maximize the agent's rewards for their actions. At each step of learning, the RL agent observes the current state of the environment and uses an internal policy to determine which available action would yield the most reward, the goal being maximizing the cumulative rewards over the duration of an episode (a simulation that reaches an end condition).

Basic reinforcement learning usually only concerns itself with a single agent acting to maximize its reward within an environment (partially or completely observable). Multi-agent systems, where multiple agents act at the same time in a competitive or cooperative fashion, introduce an additional layer of complexity as each agent needs to factor in the behavior of the other agents when learning its behavior policy. Multi-agent reinforcement learning (MARL) is the branch of RL that deals with these systems, studying how multiple agents learn to interact with each other and their common environment. In MARL, typically, each agent has sensors to perceive the surroundings, and actuators to perform actions.

MARL presents three distinct, broad types of problems:

- Fully cooperative: the agents want to work together towards a common goal (maximizing a global reward);
- Fully competitive: the agents compete with each other to reach their goal (maximizing an individual reward);
- Mixed: agents are divided in groups with shared goals, each group competes against other groups to achieve their own goal (maximizing the group reward).

We focused on the third category, since it mixes in elements of both and makes for an interesting study case to be applied to games: many modern video games present scenarios that can be modeled as a multi-agent RL mixed problem, such as sports games, multiplayer online battle arenas, and more.

Specifically, we were intrigued by the idea of applying different algorithms to a specific resource-gathering problem modeled after the "cell stage" of the video game Spore. To put it simply, it's an environment that simulates a simplified ecosystem of preys and predators. The specifics of the problem, agents and environment will be discussed further later in the paper.

As for the algorithms, we decided to use Multi Agent Posthumous Credit Assignment (MA-POCA), Proximal Policy Optimization (PPO), and Soft Actor-Critic (SAC), all implemented within the Unity ML-Agents package. The following section will have more in-depth information about these techniques and the ML-Agents framework.

Therefore, our research question is the following: given three different algorithms (MA-POCA, PPO and SAC) applied to a mixed MARL problem of preys and predators, we want to learn how they perform and how their performances differ. We hope that the result of the experiments will provide interesting insights on which algorithm performs best and on the differences between these 3 approaches.

2 Background

Most of the research within MARL is quite recent, and usually focused on cooperative problems as they have more applications in fields like robotics (e.g.: coordinating

robot swarms), economics (e.g.: defining a tax system). Merdivan, Hanke, and Geist [1] conducted a survey similar to ours on three algorithms, namely SAC, PPO and a modified version of PPO - however, their research is limited to cooperative problems such as the Hopper task from OpenAI Gym.

Vanneste et al. [2] consider mixed problems with a focus on communication between agents belonging to the same groups, which is a very interesting concept to be explored in future experiments. Their findings show that by communicating in a partially observable environment, agents can achieve performances as if the environment was fully observable.

Of the three algorithms we chose to compare, PPO and SAC seem to be the most well known as they are present in multiple papers, whereas MA-POCA has had little to no research due to how recent it is. This should make for an interesting comparison as we should be able to understand more of how MA-POCA works by comparing its results to more well-known algorithms.

The Unity Machine Learning Agents Toolkit (ML-Agents) is an open source project which enables researchers and developers to create simulated environments using the Unity Editor and interact with them via a Python API [3].

The ML-Agents Toolkit is mutually beneficial for both game developers and AI researchers as it provides a central platform where advances in AI can be evaluated on Unity's rich environments and then made accessible to the wider research and game developer communities [3].

The ML-Agents Toolkit comes with implementations (based on PyTorch) of a few state-of-the-art algorithms. For the purposes of this paper we take into consideration among those Multi Agent Posthumous Credit Assignment (MA-POCA), Proximal Policy Optimization (PPO), and Soft Actor-Critic (SAC).

2.1 Multi Agent Posthumous Credit Assignment (MA-POCA)

MA-POCA addresses the Posthumous Credit Assignment problem, created when agents terminate early or spawn mid-episode, without the use of inefficient absorbing states (inefficient as they introduce more complexity and require more resources even for agents that no longer impact the environment) [4].

In cooperative settings with shared rewards, an agent acts to maximize the expected future reward of the group. In scenarios where an agent's actions bring group rewards after the agent's termination (e.g. a self sacrificial event), the agent will not receive those rewards and will be unable to observe the state of the environment at the time the group receives the reward. Therefore, an agent must learn to maximize rewards that it cannot experience, presenting a critical credit assignment problem Cohen et al. call the Posthumous Credit Assignment problem [4].

By working within the centralized training, decentralized execution framework and applying a self-attention mechanism [5] to only the active agent information before the critic, MA-POCA can naturally handle agents that are created or destroyed within an episode but share a reward function and can scale to an arbitrary number of agents [4].

MA-POCA learns a centralized value function to estimate the expected discounted return of the group of agents and a centralized agent-centric counterfactual baseline to achieve credit assignment in the manner of COMA (COunterfactual Multi-Agent Policy Gradients) [6].

2.2 Proximal Policy Optimization (PPO)

Policy gradient methods work by computing an estimator of the policy gradient and plugging it into a stochastic gradient ascent algorithm. While it is appealing to perform multiple steps of optimization on this loss L^{PG} using the same trajectory, doing so is not well-justified, and empirically it often leads to destructively large policy updates.

In Trust Region Policy Optimization [7], an objective function (the "surrogate" objective) is maximized subject to a constraint on the size of the policy update. This problem can efficiently be approximately solved using the conjugate gradient algorithm, after making a linear approximation to the objective and a quadratic approximation to the constraint. While the theory justifying TRPO actually suggests using a penalty instead of a constraint, TRPO uses a hard constraint rather than a penalty because it is hard to choose a single value for the penalty coefficient that performs well across different problems—or even within a single problem, where the the characteristics change over the course of learning.

PPO is a family of policy optimization methods that use multiple epochs of stochastic gradient ascent to perform each policy update. These methods have the stability and reliability of trust-region methods but are much simpler to implement, requiring only a few lines of code change to a vanilla policy gradient implementation, applicable in more general settings, and have better overall performance [8].

2.3 Soft Actor-Critic (SAC)

Model-free deep reinforcement learning algorithms hold the promise of automating a wide range of decision making and control tasks. However, these methods typically suffer from two major challenges: very high sample complexity and brittle convergence properties, which necessitate meticulous hyperparameter tuning.

One cause for the poor sample efficiency of deep RL methods is on-policy learning: some of the most commonly used deep RL algorithms, such as TRPO [7], PPO [8] or A3C [9], require new samples to be collected for each gradient step. This quickly becomes very expensive, as the number of gradient steps and samples per

step needed to learn an effective policy increases with task complexity.

Off-policy algorithms aim to reuse past experience. Unfortunately, the combination of off-policy learning and high-dimensional, nonlinear function approximation with neural networks presents a major challenge for stability and convergence [10].

The maximum entropy reinforcement learning framework is a framework where the actor aims to maximize expected reward while also maximizing entropy. That is, to succeed at the task while acting as randomly as possible.

Soft Actor-Critic (SAC) is an off-policy maximum entropy deep reinforcement learning algorithm that provides sample-efficient learning while retaining the benefits of entropy maximization and stability [10].

3 Test bench mechanics

The test bench we decided to develop to compare the three algorithms described in the previous section is meant to simulate a basic non-zero-sum adversarial game, with multiple agents needing to be trained for different goals but without a clear win or lose scenario. This decision was made because existing examples in the Unity ML-Agents Toolkit for multi-agent training tend to focus on symmetrical games or cooperative behaviors to achieve a binary win/lose final state, which is only a subset of possible applications of multi-agent training.

The environment we developed is a 2-dimensional simulation of a square petri dish, where simple organisms (hereon also referred to as "cells" or "agents") of different types (herbivores and carnivores) need to move in order to collect food to replenish their constantly decreasing energy level and survive. All cells share the same basic parameters, such as maximum speed, size, energy and vision radius. The key difference is that herbivores refill part of their energy by collecting green plant food pellets, while losing energy if collecting red meat food pellets, whereas carnivores are affected by pellets in the opposite way. Carnivores can additionally kill herbivores, by colliding with them, to transform them into a meat pellet and eat them. During training, the petri dish is a square of a finite size. In each training petri dish there are a fixed amount of plant pellets and meat pellets distributed randomly across the petri dish area, with food pellets being randomly repositioned upon consumption (or deleted if the food count is greater than the maximum allowed value). Agents are also all starting at a random location at the beginning of each episode. Each episode is then run for a maximum specified amount of fixed update steps, or until all training agents are dead. A snapshot of one training environment is visible in Figure 3a.

Agents themselves are simple diamond sprites, colored green for herbivores and red for carnivores, with a purple circle placed towards their forward direction. Each cell has a vector observation space dimension of 1330 floats, corresponding to 5 stacked vectors of 31

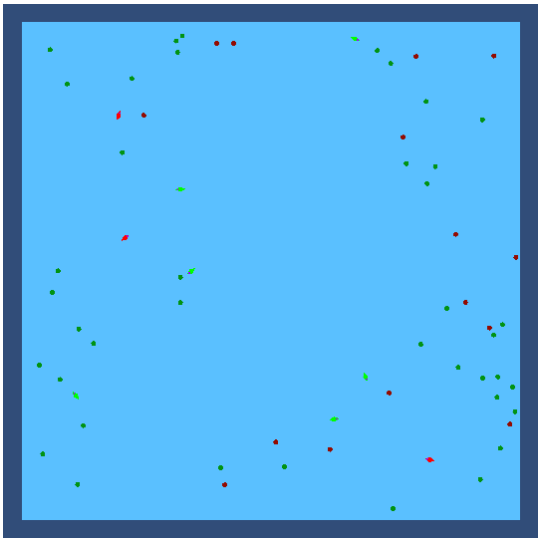

Figure 3a -The environment used during training.

forward-facing ray-casts distributed over 200 degrees and 7 backward-facing ray-casts distributed over 150 degrees, each detecting 5 possible object tags (wall, herbivore or carnivore agent, plant or meat food pellet) along with information on weather something was hit by the ray and the object distance. The forward rays' length is double than the backward ones, to simulate a greater awareness of the surroundings in the forward direction (Figure 3b). The agents' output is a 2 dimensional continuous action vector, which controls the forward speed (clamped between 0 and 1), and the rotation direction (counterclockwise if negative, clockwise if positive). The reason why the actions are continuous instead of discrete is because the energy loss at each fixed update is related to the current speed of the agent, so allowing control over the movement speed can lead to interesting behaviors between exploration and predation. The agent's reward function gives a small negative reward for touching a wall and an even smaller one if the cell keeps colliding with it, while it gives a big reward for collecting a food pellet which is either positive or negative based on the compatibility of the collected pellet with the agent's diet. In addition to this, carnivores are given a small reward for colliding with herbivores, while herbivores are given a big negative reward for colliding with carnivores. All agents are then given a big negative reward for running out of energy and dying.

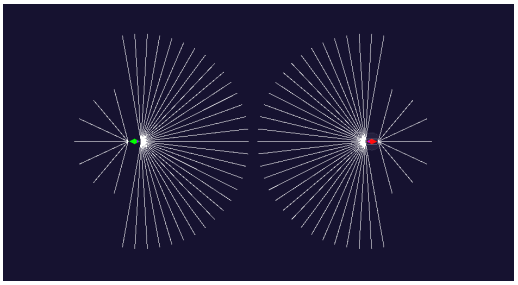

Figure 3b - A herbivore agent (on the left) a carnivore agent (on the right), highlighting their ray-casts observations.

Given that only MA-POCA supports multi-agent training, to compare it with PPO and SAC we also created basic scripted versions of the two types of cells that were used with the two single-agent algorithms to simulate the presence of the opposing type of agent. These scripted cells have a circular vision of the same radius length as the backward ray-casts of the agents, which they use to detect other objects. Their scripted behavior makes these cells rotate away from walls and rotate and move towards the nearest compatible food pellet (which also includes herbivore cells for scripted carnivores) at a faster average speed.

4 Methods

The various trainings were conducted on 5 separate scenes, with a single one for both herbivores and carnivores on MA-POCA and two separate ones for the different species on PPO and SAC. Each training scene was set up with 4 petri dishes of size 50x50, filled with 40 plant pellets and 15 meat pellets and populated with 7 herbivores and 3 carnivores. These training scenes were run for a maximum of 4000 steps at a time.

While training MA-POCA, all these cells were actual training agents with a self-play configuration [11] and divided into two multi-agent groups based on their diet. Using this multi-agent training configuration, in addition to the individual rewards for each agent described in the previous section, each group of agents received a group reward based on the performance of their whole species during an episode. In particular, each group received a small reward each time an agent died during an episode, which was negative if the agent survived less than half the episode's steps or positive if it survived more than half the steps. In addition to this, at the end of each episode each group received a positive reward proportional to the percentage of individuals still alive in the group or, alternatively, a negative reward if the entire group population died. These group rewards were then assigned to the group's individual agents after the end of the episode, based on the posthumous credit assignment performed by MA-POCA, even if the environment does not really require cooperative behavior by each species. The real advantage of using MA-POCA in this scenario is that herbivores and carnivores were performing against a version of the other species which was constantly improving over time, which should make their trained policies perform more reliably compared to agents trained only against scripted cells. For both PPO and SAC, herbivores and carnivores were trained separately, each against the appropriate number of scripted cells of the opposite species.

To achieve the best results with each of the 3 algorithms we tested, we set up the hyperparameters fed to the training algorithm through a configuration file supported by ML-Agents [12]. We ran multiple training sessions of each algorithm with different parameters and compared them until reaching the most efficient results, as briefly discussed in section 5. After all hyperparameters were finely tuned, we trained the PPO agents for 30 millions steps, the SAC ones for 3 millions steps and both MA-POCA carnivores and herbivores for 10 millions steps each. All these training sessions were performed on computers equipped with an Intel Core i7 8700 and a NVIDIA GeForce RTX 2080.

Lastly, after having generated the final models for each species with each algorithm, we run a comparison test of each possible combination of those. To abstract this test from the training conditions, we created a petri dish of size 150x150, filled with 300 plant pellets and 100 meat pellets and populated with 30 herbivores and 15 carnivores. For each pair of herbivore-carnivore models we then ran 100 episodes lasting 10000 steps each and recorded data at the end of each episode about the average percentage of survivors and the average age of individuals, for both herbivores and carnivores.

5 Parameters

In order to determine the optimal hyperparameters to use on each algorithm, we ran a series of controlled tests in which we confronted the training results of different sets of parameters. With some of the parameters being interconnected (e.g.: the ones tied to the experience buffer), some experiments required us to change more than one parameter at a time. Moreover, while PPO and MA-POCA share all of their hyperparameters (therefore we could fine-tune only PPO and obtain comparable results), SAC presents similar parameter names with different value ranges (such as batch size and buffer size), which is why the values in the config files are different. As a general guideline to speed up the tuning, we considered the “ratio” between the default values in PPO and SAC for these parameters as a starting point for our experiments (e.g.: the buffer size default value in PPO is about 5 times smaller than in SAC, therefore after finding an optimal value for PPO we used 5 times that value as a starting point for fine-tuning SAC) [13].
The hyperparameters we experimented with are:

- learning rate, which influences the “strength” of each gradient descent step; it was set at 0.001 (BiggerLR configuration), 0.0001 (standard configuration) and 0.00001 (SmallerLR configuration);
- batch size and buffer size (jointly, since they are interconnected), which influence how many of the previous experiences are used to learn from in each iteration; the BigBatch configuration used 5120 batch size / 40960 buffer size for PPO and 1024 / 200000 for SAC, whereas the SmallBatch configuration used 2048 batch size / 20480 buffer size for PPO and 512 batch size / 100000 buffer size for SAC;
- training steps, adjusted as needed from the results of training.

5.1 Parameters of MA-POCA

MA-POCA parameters are mostly common to the ones used in PPO, for which reason we decided to use the parameters tuned for PPO (discussed in section 5.2) for MA-POCA too. In addition to the majority of parameters which are common to PPO, MA-POCA has some specific ones regarding self-play which we used with the default configuration provided, as it leads to a balanced and stable learning process while introducing a high enough diversity of evolved opponent's behaviors to learn from.

The only parameter we tuned specifically for MA-POCA was the training steps. We tried to train both carnivores and herbivores for the same number of steps as PPO (30 millions) at first, which took almost 3 times as long, and later tried to reduce the steps to 10 millions per agent type, which led to very similar training times to PPO's ones.

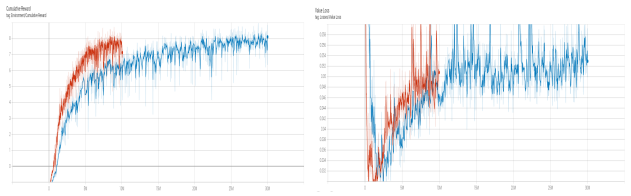

Figure 5.1a - Cumulative reward (left) and value loss (right) plots of carnivores trained with MA-POCA with longer (blue) and shorter (red) configurations.

As shown in Figure 5.1a, the shorter training led to comparable results in terms of cumulative reward and value losses, with the shorter version converging faster and the longer one only achieving marginally higher results on paper. To confirm the hypothesis of the shorter training being comparable, if not better, we also ran our final comparison test with both of the resulting neural networks. Despite the results always being numerically slightly better for the shorter-trained brains, they showed to be not significant due to the standard deviations being higher than the actual differences in the averages of the data collected. Having chosen to rely on either of the MA-POCA final configurations should consequently have borne no significant difference in the final results, which led us to choose the shorter version for a fairer relative training time comparison.

5.2 Parameters of PPO

When optimizing PPO, we chose to fine-tune the algorithm for one of the two tasks (the carnivore behavior) and apply the same configuration to the other agents. The reasoning behind this is that the complexity of the tasks and overall action spaces are extremely similar, so the same parameters would do well on both tasks. We decided to start by tuning the learning rate. As previously mentioned, we tested three different configurations (0.001, 0.0001 and 0.00001 learning rate) while keeping all other parameters identical. Figure 5.2a shows the cumulative reward plots for all three values: the smaller learning rate converges much more slowly than the others, with little significant difference between the standard and larger versions.

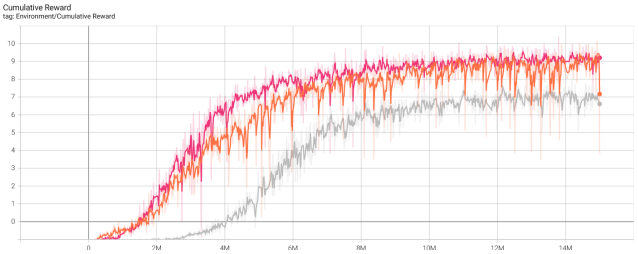

Figure 5.2a - Cumulative reward plots of carnivores trained with PPO with 0.001 LR (orange), 0.0001 LR (magenta) and 0.00001 LR (gray).

However the value loss plot (Figure 5.2b) is more stable for the standard learning rate, so we decided to keep that value going forward.

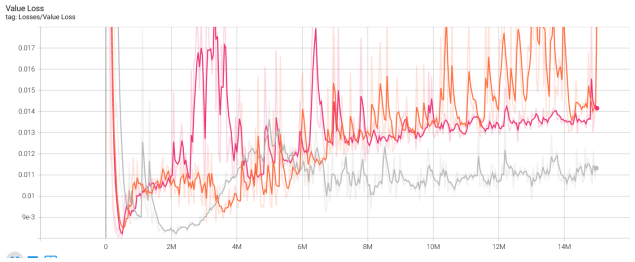

Figure 5.2b - Value loss plots of carnivores trained with PPO with 0.001 LR (orange), 0.0001 LR (magenta) and 0.00001 LR (gray).

When comparing PPO training results using different batch/buffer sizes, both versions reach approximately the same cumulative reward (Figure 5.2c), but the version with smaller values converges noticeably faster, with a more stable value loss plot (Figure 5.2c); therefore it was chosen to be used for the final experiment against the other algorithms.

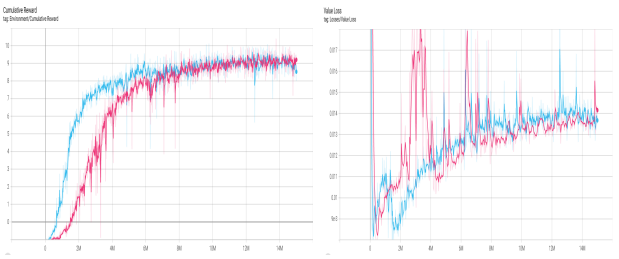

Figure 5.2c - Cumulative reward (left) and value loss (right) plots of carnivores trained with PPO with BigBatch (magenta) and SmallBatch (cyan) configurations

5.3 Parameters of SAC

The SAC algorithm seems to yield slightly better results when training with a smaller batch size. When looking at the plots for cumulative reward from training with SmallBatch and BigBatch configurations, the SmallBatch one is slightly more stable after about 2 million steps, and converges to a slightly higher cumulative reward value

(Figure 5.3a). While the SmallBatch version presents a sharp drop around 2 million steps, its counterpart has even sharper declines later in the training process.

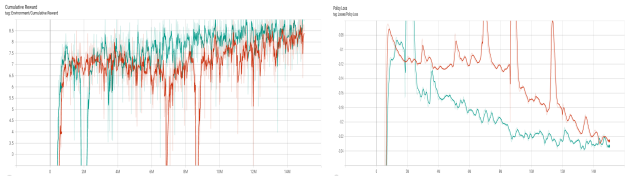

Figure 5.3a - Cumulative reward (left) and policy loss (right) plots for SAC with BigBatch (red) and SmallBatch (green) configuration.

The policy loss plots, however, display a significantly better trend and converge in a faster and more stable manner for the SmallBatch configuration (Figure 5.3a). With the Q1/Q2 losses also displaying a more stable convergence to a near-zero value, we decided to keep the smaller batch and buffer size for SAC.

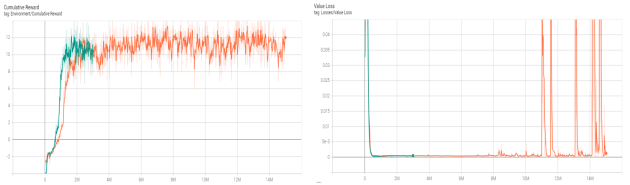

Figure 5.3b - Cumulative reward (left) and value loss (right) plots for SAC with Final (orange) and FinalShorter (green) configurations.

The other adjustment we made was to reduce the training steps, as we noticed that the cumulative reward and loss graphs weren't improving much. Therefore we reduced the training steps from 15 million to 3 million (we will be referring to these new parameter configurations as "Final" and "FinalShorter"). We noticed that, overall, reducing the amount of training steps resulted in much more stable training.

In the shorter training the cumulative reward converges around the same value faster than the longer variant which, as it can be seen in Figure 5.3b, does not improve much if at all after 2 million steps. The FinalShorter version also presents consistently stable Q1, Q2 and value loss graphs, and all of the losses converge to near-zero values (Figure 5.3c).

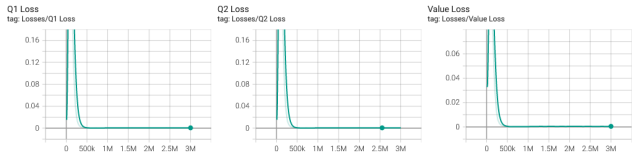

Figure 5.3c - Loss graphs for SAC with FinalShorter configuration.

Similarly, the policy loss only shows marginal improvements when trained for longer as it can be seen in figure 5.3d. Therefore, considering that the training time for SAC with 15 million steps is well over 2 days (whereas 3 million steps take around 12 hours, a result comparable with the PPO trainings), we opted to use the FinalShorter parameters (small batch size, 3 million steps) for our experiments.

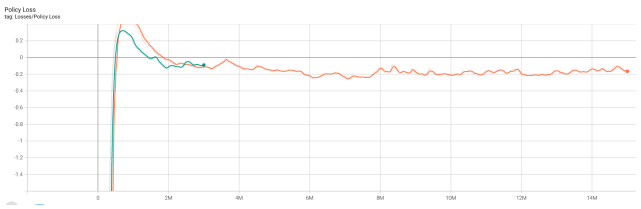

Figure 5.3d - Policy loss plots for SAC with Final (orange) and FinalShorter (green) configuration.

6 Results

The training performed using the final parameters discussed above led to comparable training times on the machines we used. The average training time was 12 hours, 19 minutes and 1 second, with a standard deviation of 33 minutes and 32 seconds. The only significantly faster training were the herbivores trained with PPO, which required only 11 hours and 12 minutes.

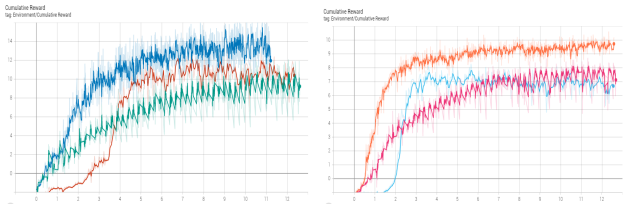

Figure 6a - Cumulative rewards comparison of the final training of herbivores (left) and carnivores (right) with MA-POCA (green/magenta), PPO (blue, orange) and SAC (red/cyan).

As shown in Figure 6a, just looking at the raw cumulative rewards from the final training would suggest that PPO reaches the best result, followed by SAC and MA-POCA achieving similar values, but SAC converging to them faster.

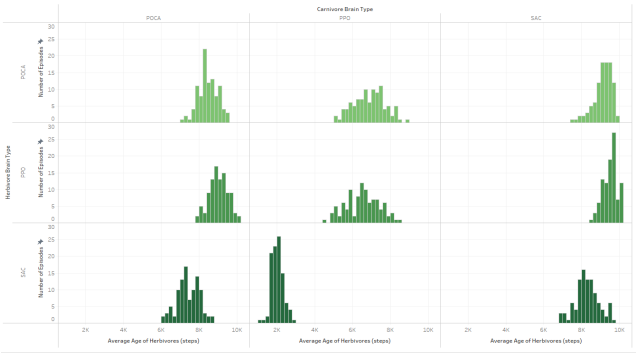

Figure 6b - Average age distributions for herbivores.

To better compare the actual behaviors, though, we performed a comparison test as described in the last paragraph of section 4. We then collected the resulting data and analyzed, for each run, the average age achieved by the agents of both roles as well as the percentage of

survivors, as in agents that lasted until the end of the episode. These two parameters allowed us to gauge how much each algorithm succeeded in the goal of staying alive as long as possible.

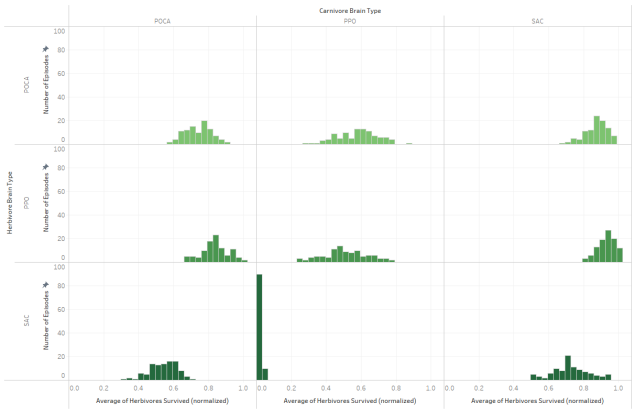

Figure 6c - Average survival rate distributions for herbivores.

When comparing the average age (Figure 6b) and survival rate (Figure 6c) of herbivores, it can immediately be noticed how SAC performs generally worse than the others both against MA-POCA and SAC-driven carnivores, but SAC performs very badly against PPO-driven carnivores, with all agents dying before 4000 steps on average, and the average survival rate being extremely close to 0% (about 0.33%). PPO and MA-POCA have fairly similar results, with PPO-driven herbivores slightly outperforming MA-POCA carnivores.

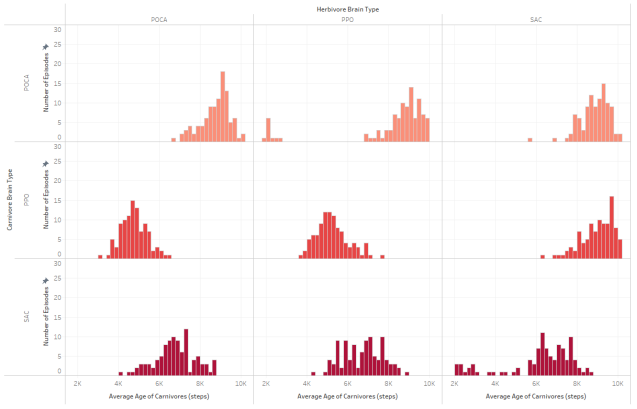

Figure 6d - Average age distributions for carnivores.

So far it would seem that PPO produced the better results, as PPO-driven herbivores tend to survive more than the others and the plots suggest that PPO carnivores are also more efficient in hunting, as herbivores in all algorithms have worse performances when going against PPO carnivores. However, examining the plots from the perspective of carnivores (Figures 6d and 6e) reveals more interesting results.

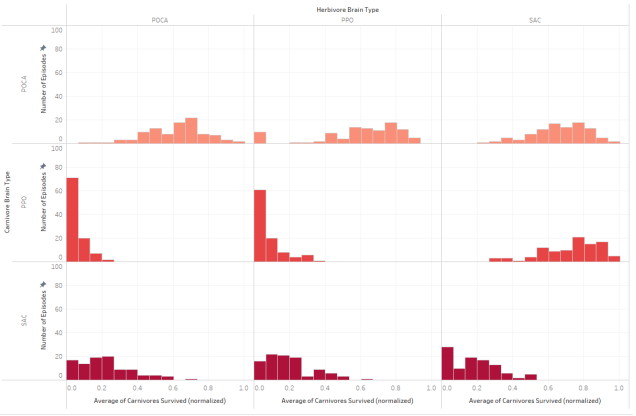

Figure 6e - Average survival rate distributions for carnivores.

While SAC keeps having overall worse performances when compared to the other algorithms, its results are still somehow consistent, with the carnivores hardly having more than half their agents alive by the end of the episodes. MA-POCA also shows very consistent performances across the board, with a trend of the majority of the carnivore agents surviving the whole episode and an age distribution consistently hovering above 7000 steps. PPO, however, has very inconsistent results, doing better than MA-POCA against SAC-driven herbivores but otherwise displaying a high number of runs ending with no carnivore survivors.

Herbivore Brain Type	Carnivore Brain Type	AVG of Herbivores Survived (%)	SD of Herbivores Survived (%)	AVG Age of Herbivores (steps)	SD of Herbivores Age (steps)
POCA	POCA	73,167	7,674	8.459,732	499,926
	PPO	56,833	11,734	6.889,676	817,377
	SAC	86,233	6,661	9.120,277	474,769
PPO	POCA	82,933	7,832	9.026,252	481,487
	PPO	49,900	12,778	6.531,502	878,230
	SAC	92,300	5,096	9.504,569	355,450
SAC	POCA	52,833	8,059	7.422,046	585,584
	PPO	0,333	1,005	2.036,789	303,973
	SAC	71,600	10,725	8.298,116	625,707

Figure 6f - Survival data of each herbivore algorithm against each carnivore algorithm

Herbivore Brain Type	Carnivore Brain Type	AVG of Herbivores Survived (%)	SD of Herbivores Survived (%)	AVG Age of Herbivores (steps)	SD of Herbivores Age (steps)
POCA	POCA	73,167	7,674	8.459,732	499,926
	PPO	56,833	11,734	6.889,676	817,377
	SAC	86,233	6,661	9.120,277	474,769
PPO	POCA	82,933	7,832	9.026,252	481,487
	PPO	49,900	12,778	6.531,502	878,230
	SAC	92,300	5,096	9.504,569	355,450
SAC	POCA	52,833	8,059	7.422,046	585,584
	PPO	0,333	1,005	2.036,789	303,973
	SAC	71,600	10,725	8.298,116	625,707

Figure 6g - Survival data of each carnivore algorithm against each herbivore algorithm

Looking at standard deviations for our results (Figure 6f and 6g), carnivores have generally higher standard deviation than herbivores, suggesting their performances are more inconsistent - this lines up with the fact that investing energy into chasing herbivores is a high risk/high reward behavior and that the carnivore's food source is therefore less stable. This is consistent with what we observed in the value distributions. PPO carnivores

have comparatively lower standard deviations (relative to the same herbivore "matchups" for the other algorithms) and very polarized performances, confirming that they either perform very well or very badly with little in-between.

From these results, it can be surmised that PPO-driven carnivores tend to be very aggressive and prioritize chasing and killing herbivores. This explains why herbivores, in general, perform worse against PPO-driven carnivores. However this aggressive behavior is high-risk/high-reward, leading to a high rate of failure against "smarter" herbivore agents. MA-POCA, on the other hand, displays much more balanced results - both the herbivore and carnivore agents have adapted to survive, with the carnivores being generally less aggressive and focusing more on environmental food than actively chasing herbivores. We believe this is thanks to how MA-POCA trains the two types of agent against each other concurrently: as the herbivores become better at evading carnivores, the latter learn to rely more on environmental food pellets to survive longer. SAC is consistently worse than both MA-POCA and PPO on all fronts when applied to this problem.

7 Conclusions

We observed MA-POCA to be generally more consistent in its final behaviors in our non-zero-sum adversarial game simulation. Despite PPO being a strong contender within certain conditions, having a separate learning for each adversarial agent can lead to behaviors that are too specialized and cannot adapt. SAC, while learning less specialized behaviors than PPO, consistently underperformed. In conclusion, MA-POCA appears to be the best choice for training a multi agent environment in a non-zero-sum game.

Generally, our analysis could be improved by conducting more tests to fine-tune parameters in different combinations. Unfortunately the time required for training was a bottleneck that prevented us from conducting further tests, but we would have tried more parameter combinations to see if we could achieve even better results (especially in terms of policy and value losses). Moreover, experimenting with different parameters for herbivores and carnivores has the potential to yield interesting results.

As far as changes to the environment are concerned, we realized that further separating rewards obtained from food and rewards from killing agents could have given us more interesting insights. Future work on this project could include implementing domain randomization [14] to introduce more variance in the training environments and observe how the agents learn to adapt.

Finally, we considered the possibility of training PPO and SAC a second time against a pre-trained agent (rather than a scripted one) to improve their performances and compare them to MA-POCA which already does concurrent training.

References

[1] E. Merdivan, S. Hanke, M. Geist, Modified Actor-Critics, 2019

[2] A. Vanneste, W. V. Wijnsberghe, S. Vanneste, K. Mets, S. Mercelis, S. Latré, P. Hellinckx, Mixed Cooperative-Competitive Communication Using Multi-Agent Reinforcement Learning, 2021

[3] A. Juliani, V. P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Matter, D. Lange, Unity: A General Platform for Intelligent Agents, 2018

[4] A. Cohen, E. Teng, V. P. Berges, R. P. Dong, H. Henry, M. Mattar, A. Zook, S. Ganguly, On the Use and Misuse of Absorbing States in Multi-agent Reinforcement Learning, 2021

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, I. Polosukhin, Attention is all you need, 2017

[6] J. Foerster, G. Farquhar, T. Afouras, N. Nardelli, S. Whiteson, Counterfactual multi-agent policy gradients, 2018

[7] J. Schulman, S. Levine, P. Abbeel, M. Jordan, P. Moritz, Trust region policy optimization, 2015

[8] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, Proximal policy optimization algorithms, 2017

[9] V. Mnih, A. P. Badiav, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, K. Kavukcuoglu, Asynchronous Methods for Deep Reinforcement Learning, 2016

[10] T. Haarnoja, A. Zhou, P. Abbeel, S. Levine, Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor, 2018

[11] Unity Technologies, Unity ML-Agents Toolkit, Training-Self-Play.md, https://github.com/Unity-Technologies/ml-agents/blob/0.14.0/docs/Training-Self-Play.md

[12] Unity Technologies, Unity ML-Agents Toolkit, Training-Configuration-File.md, https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-Configuration-File.md

[13] Unity Technologies, Unity ML-Agents Toolkit, Training-ML-Agents.md, https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-ML-Agents.md

[14] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, P. Abbeel, Counterfactual Multi-Agent Policy Gradients, 2017