AASMA Project Report

Jonas Korkosh, 108463[1] Marius Vårdal, 111458[1]

[1]Instituto Superior Técnico, Lisbon, Portugal

Abstract

This report explores the performance of different policies in the multi-agent predator-prey environment ”Simple Tag” from PettingZoo. Our main focus was to investigate the way different behavioural policies affect performance for the adversarial agents. Both explicitly programmed policies with and without coordination, as well as policies learned through Reinforcement Learning with Proximal Policy Optimization were included in the analysis. Our analysis found significant improvements in adverserial agents upon the use of coordination and even further improvements upon utilizing a Reinforcement Learning approach.

1 Introduction

This project aims to implement the behaviors of adversarial agents (predators) and a good agent (prey) in a multi-agent system. The problem we have chosen to look at is a game from PettingZoo called “Simple Tag”. This game environment provides all the agents with information about velocity and positions for the other agents. The predators try to catch the prey by colliding with it, while the prey tries to escape. The prey is able to move faster than the predators, which is what makes this a challenging task. If there was only one predator and one prey, the prey would comfortably escape every time. Therefore, coordination between the agents is an important theme in this paper.

2 Game and Environment Specification

2.1 Environment specification

The environment is a predator-prey environment that hands out rewards based on the behaviour of adversaries and good agents. Good agents are faster than the adversaries and there is a cost associated with touching the adversaries. (-10 for each collision). Adversaries are slower and are rewarded for hitting good agents (+10 for each collision). The reward is given to all predators upon hitting the prey, making this a pure coordination game.

The action space for the agents is discrete with 5 possible values, $\{no\_action, move\_left, move\_right, move\_down, move\_up\}$.

There is full observability for all agents and adversaries, and the observations are given by a vector on the form:

$$\mathbf{obs} = \begin{pmatrix} \text{self\_vel} \\ \text{self\_pos} \\ \text{landmark\_rel\_positions} \\ \text{other\_agent\_rel\_positions} \\ \text{other\_agent\_velocities} \end{pmatrix}$$

Note that in our case, we chose to simplify the game by not including any landmarks. The values for the positions and velocities lies in the range $(-inf, inf)$, resulting in an infinitely large state space.

A graphical representation of the game is shown in figure 1.

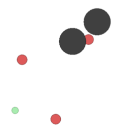


Figure 1: The Simple Tag game from PettingZoo visualized.

2.2 Game theoretical formalism

The game can, with some simplifications, be fit into the Markov game theoretical formalism as described in [1]

- Discrete time $t = 0, 1, 2, \dots$
- A set of $n > 1$ agents.
- A discrete set of states $s \in S$.
- For each agent i, a discrete set of actions $a_i \in A_i$.
- A stochastic transition model $p(s'|s,a)$ that is conditioned on the joint action $a = (a_i)$ at state s.
- For each agent i, a reward function $R_i : S \times A \mapsto \mathbf{R}$, that gives agent i reward $R_i(s,a)$ when joint action a is taken at state s.

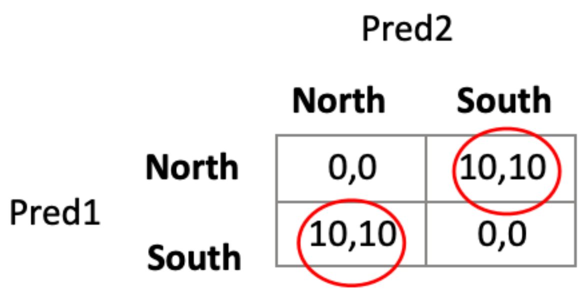
The two first points are fulfilled. We also have the state space as the set of all possible observations (since the environment is fully observable). Since the values of positions and velocities can take on all values in the range $(-inf, inf)$ the state space is not discrete. However, in this discussion we will simplify and consider the state space to be discrete as the actions which dictate the movement within the state space are discrete, and thus (if we neglect the velocity dynamics) we can look at the game as a grid with the agents moving around in it. This is of course not completely accurate, but as we will see, can still yield good results.

We also observe that $A_i = \{Up, Down, Left, Right, Stay\}$ for all agents. Which is discrete.

In this case the stochastic transition model $p(s'|s,a)$ is deterministic given the joint action $a = (a_i)$ at state s. Meaning if we know all the actions that were taken at state s we know where all the agents will end up. This is because if an agents selects action ”Right” it is determined that it will move right in the next time step.

Furthermore, the reward function $R_i : S \times A \mapsto \mathbf{R}$, gives the good agent $R_{prey}(s,a) = -10$ and the adversaries $R_{pred}(s,a) = 10$ when a joint action a is taken at state s such that the one of the adversaries touch the good agent. (ie have the same position, within some threshold). Otherwise $R_i(s,a) = 0$.

Furthermore, the environment provides the prey with an additional negative reward upon moving far from the starting point to prevent it from escaping into infinity. In this project this reward-function was neglected, rather, the prey was programmed to never select an action that would take it out of bounds.



		Pred2	
		North	South
Pred1	North	0,0	10,10
	South	10,10	0,0

Figure 2: The reward matrix for the normal form game where the predators are trying to select which direction to surround the prey from. The Nash equilibria are marked by red circles.

3 Agents engineering

3.1 Coordination game

In this game we have several predators and a prey that is faster than any one of them. It is then clear that the predators must coordinate in order to catch the prey. As previously discussed, all predators share the same payoff function, making this a pure coordination game. We have no way of explicit communication between the agents, and an explicit communication approach could be overly complicated. Therefore, we decide to simplify the problem as a normal form game and use roles to decide on a Pareto optimal Nash equilibrium.

In order to simplify the problem, we simplify the Markov game by looking at it as a normal form game. We create a set of *target points* surrounding the prey, the number of which correspond to the number of predators. Then, the predators need to move to one target point each to surround the prey. If they surround the prey, they get a reward of $R_{pred} = 10$ and the prey gets $R_{prey} = -10$. If not, they all receive $R = 0$ To surround the prey the predators need to select the same Nash equilibrium, ie. they all need to agree on which target point each of them move to. This is summarized in figure 2 with two agents for simplicity. The same reasoning works for multiple agents.

For the adversaries to select which Nash equilibrium to move to, we engineer our agents to include roles. We define a potential function and use it to assign adversaries to their most suitable target points, which they can move towards to surround the prey. This will be discussed in detail in section 3.2. It is worth noting that even though the agents are choosing the right target point in the normal form game, the underlying game is actually a more complex Markov game, so surrounding the prey is not enough, the predators also need to move towards the prey from the different angles at the same time in order to be able to reliably touch it without letting it escape. This mechanism will be covered in detail in section 3.2.6.

3.2 Agents

In this section we lay out the agents and how they were implemented. Each agent was implemented in python inheriting from an abstract base class with a $see()$ method to store the observations. Then each agent implements a $get\_action()$ method to provide the actions given the previous observations. All the agents are pure except the coordinating agents, which store state. The implementations of different agents now follows.

3.2.1 Prey: Immobile Agent

The immobile agent simply stands still in its place, by always selecting $a_{prey} = no_action$. This was implemented for the prey to provide a benchmark for the predators.

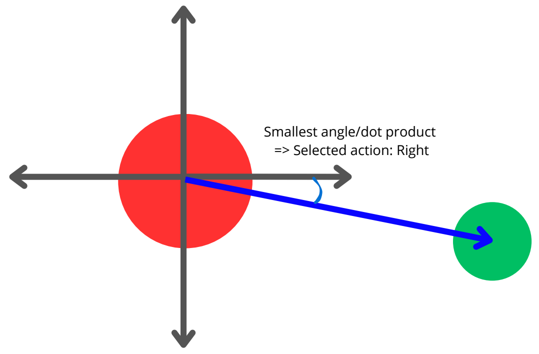


Figure 3: The greedy agent selects the action by comparing the relative position to the prey and the direction of the possible actions, choosing the action that points the most in the direction of the prey. Comparison using dot product.

3.2.2 Random Agent

This agent simply selects one of the five actions in the action space $A = [no_action, move_left, move_right, move_down, move_up]$ uniformly at random. This type of agent was implemented both for predators and preys.

3.2.3 Predator: Greedy Agent

The greedy predator agent always selects the action that takes it towards the prey as quickly as possible. To achieve this we first define four unit vectors pointing in the direction of each of the four actions in the action space. We call these the action vectors. Then we calculate the relative position vector between the predator and prey. The greedy action is the defined as the action corresponding to the action vector whose dot product with the relative position vector is the largest. This is illustrated in figure 3.

3.2.4 Prey: Avoiding Agent

To engineer the prey we came up with a heuristic. The pseudo-code implementation is shown in algorithm 1. The score of each action is determined by the total distance it will produce to all adversaries after it has been made (assuming the adversaries stay in place). To determine this the algortihm first calculates the expected position after taking the action, as seen in line 4. Then, sums the total distance that would produce to all other adversaries. The action which produces the largest total distance is selected. This is illustrated in figure 4. Additionally, the prey is forced to stay within the bounds: $xpos : [-1, 1]$ and $ypos : [-1, 1]$. This prevents the prey from simply escaping by running away to infinity.

In addition, any action that will lead the prey to go out of bounds will be assigned a score of 0. This is to prevent escaping into the horizon from being a viable strategy.

3.2.5 Prey: Avoiding Nearest Adversary Agent

To improve the performance of the prey, a second heuristic was implemented. This heuristic seeks to select the action that maximizes the distance to the nearest predator after the action has been made.

Algorithm 1 Prey: Avoiding Agent

```
1: $max\_dist \leftarrow 0$
2: $tot\_dist \leftarrow 0$
3: for each $(action, action\_vector)$ do
4:     $pos\_after\_action \leftarrow self\_pos + action\_vector$
5:     if not $in\_bounds(pos\_after\_action)$ then
6:         $tot\_dist \leftarrow 0$
7:     else
8:         for each $(adversary, adv\_pos)$ do
9:             $tot\_dist \leftarrow tot\_dist + \text{l2norm}(pos\_after\_action - adv\_pos)$
10:        end for
11:        if $tot\_dist > max\_dist$ then
12:            $max\_dist \leftarrow tot\_dist$
13:            $max\_action \leftarrow action$
14:        end if
15:    end if
16: end for
17: return $max\_action$
```

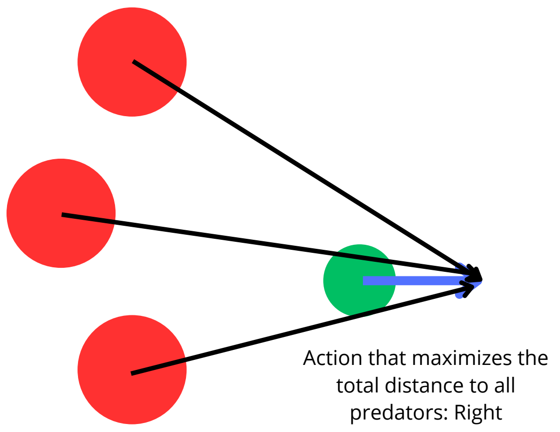

Action that maximizes the total distance to all predators: Right

Figure 4: The AvdoidingAgent prey avoids predators by selecting the action that maximises the total distance to all predators

It iteratively evaluates each action by calculating the distance to the nearest predator from the location the prey would end up at after completing the action. The pseudocode for this is shown in algorithm 2. Also here the prey is enforced to stay within the bounds: $xpos : [-1, 1]$ and $ypos : [-1, 1]$ to prevent it from running to infinity.

Algorithm 2 Prey: Avoiding Nearest Adversary

```
1: $max\_dist \leftarrow -inf$
2: $max\_action \leftarrow$ None
3: for each $(action, action\_vector)$ do
4:     $pos\_after\_action \leftarrow self\_pos + action\_vector$
5:     if not $in\_bounds(pos\_after\_action)$ then
6:         $nearest\_adv\_dist \leftarrow 0$
7:     else
8:         $nearest\_adv\_dist \leftarrow \infty$
9:         ▷ finds the distance to the nearest adversary after the action has been made
10:        for each $(adversary, adv\_pos)$ do
11:            $adv\_dist \leftarrow \text{l2norm}(pos\_after\_action - adv\_pos)$
12:            if $adv\_dist < nearest\_adv\_dist$ then
13:                $nearest\_adv\_dist \leftarrow adv\_dist$
14:            end if
15:        end for
16:    end if
17:    ▷ selects the action that maximizes the distance to the nearest adversary after the action has been made
18:    if $nearest\_adv\_dist > max\_dist$ then
19:        $max\_dist \leftarrow nearest\_adv\_dist$
20:        $max\_action \leftarrow action$
21:    end if
22: end for
23: return $max\_action$
```

3.2.6 Predator: Coordinating Agent

The coordinating agents observe how many adversaries there are in total in the environment and use this information to create the same number of internal "target points" that are evenly distributed around the good agent. Each of the target points corresponds to a role that can be occupied by one adversary. The point is to have the adversaries go towards each of the target points, thereby attempting to surround the good agent. The way we make sure that all the adversaries go toward different target points is by utilizing a heuristic function to assign the roles. This heuristic assigns highest priority to the adversary who is closest to one of the target points, meaning that this adversary gets to choose its target point first. Since every adversary wants to choose the target point closest to itself and all adversaries have full observability and therefore knows the priority of all the adversaries, each adversary can determine which target point will be assigned to itself without communicating with other adversaries or a global "game master". This heuristic can be viewed as a potential function to assign the roles.

When the adversaries are close to the target points, they decide to reduce the distance from the target points to the good agent, thereby making the adversaries approach the prey from different angles.

Another detail in the CoordinatingAgent-algorithm is that when the distance between itself and the prey is below a certain threshold, it functions as a GreedyAgent. This is to ensure a balance between surrounding the prey and actually trying to catch the prey.

The coordinating agent is illustrated in figure 5

3.2.7 Predator: RL Agent

In addition to all the explicitly programmed agents mentioned above, we trained a Reinforcement Learning (RL) agent on our

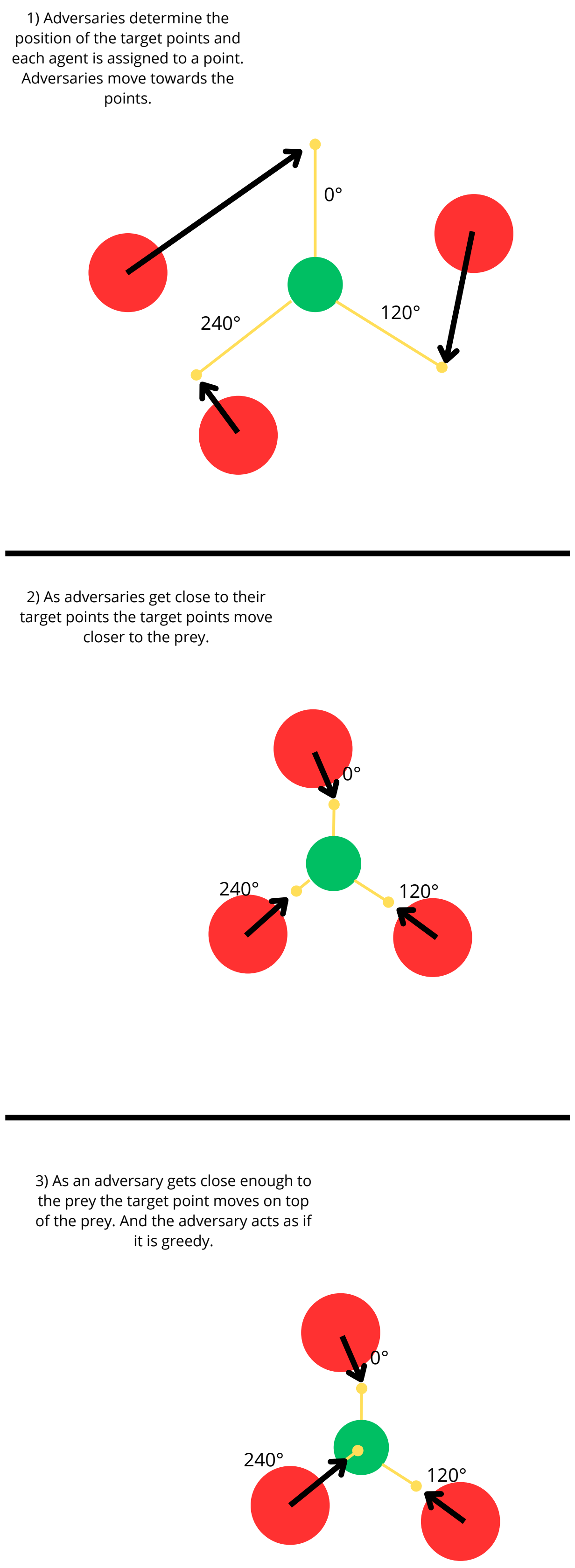

1) Adversaries determine the position of the target points and each agent is assigned to a point. Adversaries move towards the points.
0°
120°
240°
2) As adversaries get close to their target points the target points move closer to the prey.
0°
240°
120°
3) As an adversary gets close enough to the prey the target point moves on top of the prey. And the adversary acts as if it is greedy.
0°
240°
120°

Figure 5: The coordinating agents use target points to surround the prey. The target points are calculated by each prey independently and assigned using a potential function that considers the distance from the target points to the adversaries.

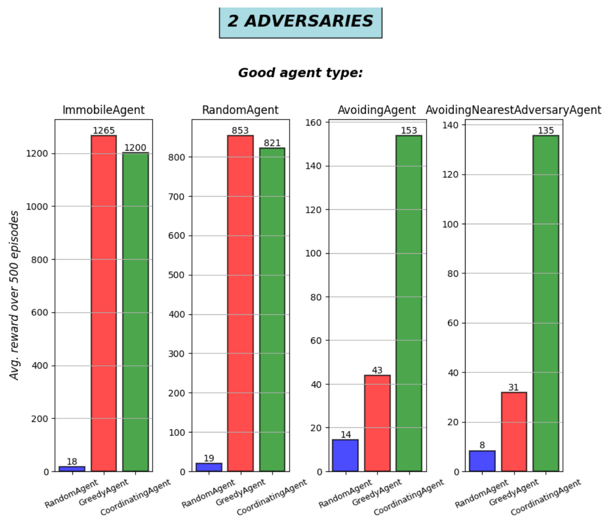


Figure 6: The average 500-episode-rewards for ImmobileAgent, RandomAgent, AvoidingAgent and AvoidingNearestAdversaryAgent as prey against two adversaries of type: RandomAgent, GreedyAgent and CoordinatingAgent.

environment. We did this with the help of the stable-baselines[1] API and the Proximal Policy Optimization (PPO) with a Multilayered Perceptron (Mlp) neural network. The good agent with the AvoidingAgent-policy was integrated as a part of the environment and agents trained with RL were the adversarial agents. Some preprocessing had to be done to make this happen. Mainly, this consisted of integrating the prey as a part of the environment (since we were only interested in training the adversaries), converting this to a custom PettingZoo parallel environment and converting this environment to a vectorized environment using a SuperSuit[2] wrapper. No preprocessing of the observation-vectors was done because their values already resembled those of a standard normal distribution.

4 Comparative analysis

4.1 Explicitly programmed agent types

To evaluate our agent's performances, we used the average reward over 500 episodes as the metric. When it comes to the explicitly programmed agents, our main focus is evaluating the performance of the CoordinatingAgent adversaries against the AvoidingAgent and the AvoidingNearestAdversaryAgent prey types. The other agent types, ImmobileAgent, RandomAgent and GreedyAgent are used as benchmarks to better understand the obtained results.

As can be seen from the figures 6, 7 and 8, the relative performance of the different agents are qualitatively quite similar. All adversary types perform best against the Immobile prey type, which make sense, considering that the prey doesn't move and should be very easy to catch. The second easiest prey to catch is the RandomAgent prey. The two last preys types AvoidingAgent and AvoidingNearestAgent are significantly harder to catch for the GreedyAgent and the CoordinatingAgent adversary types. As can be seen in figure 7, three CoordinatingAgents have an average score of 2151, 1529, 300 and 254 against the ImmobileAgent, RandomAgent, AvoidingAgent and AvoidingNearestAdversaryAgent prey types, respectively. The same numbers are 2374, 1674, 152 and 100 for the GreedyAgent. From these numbers, we can see that the GreedyAgent adversary type slightly outperforms the CoordinatingAgent adversary type when up against a "stupid" prey like

[1]https://stable-baselines.readthedocs.io/en/master/ accessed 22.05.2024
[2]https://github.com/Farama-Foundation/SuperSuit, accessed 22.05.2024

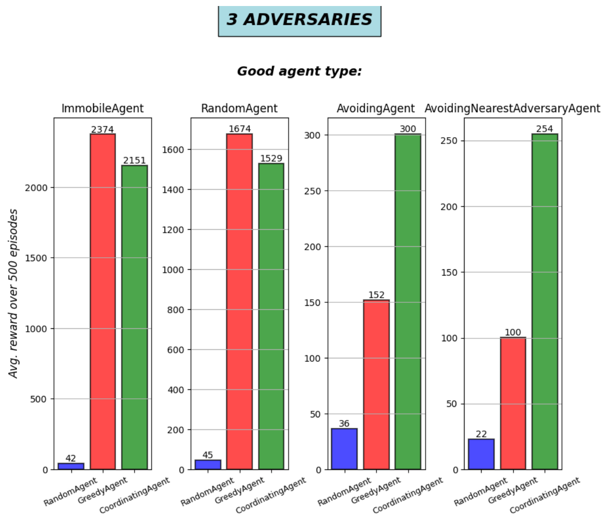


Figure 7: The average 500-episode-rewards for ImmobileAgent, RandomAgent, AvoidingAgent and AvoidingNearestAdversaryAgent as prey against three adversaries of type: RandomAgent, GreedyAgent and CoordinatingAgent.

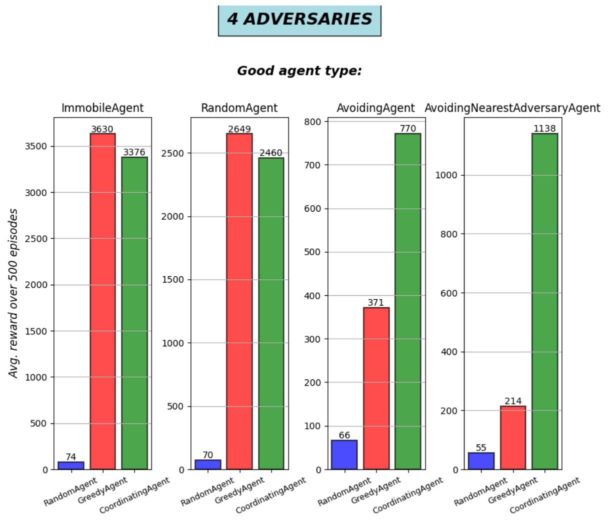


Figure 8: The average 500-episode-rewards for ImmobileAgent, RandomAgent, AvoidingAgent and AvoidingNearestAdversaryAgent as prey against four adversaries of type: RandomAgent, GreedyAgent and CoordinatingAgent.

num adversaries	prey type	
	Avoiding	AvoidingNearestAdversary
2 adversaries	3.56	4.35
3 adversaries	1.97	2.54
4 adversaries	2.08	5.32

Table 1: Ratio of average rewards for CoordinatingAgent over GreedyAgent against different types of prey.

the ImmobileAgent or the RandomAgent. This is to be expected as catching such a prey should not require any coordination, and efforts to coordinate may result in loss of effectiveness. However, the CoordinatingAgent significantly outperforms the GreedyAgent when trying to catch one of the smart preys, the AvoidingAgent and the AvoidingNearestAdversaryAgent. The ratio of average rewards for the CoordinatingAgent compared to the GreedyAgents when running against an AvoidingAgent and an AvoidingNearestAdversaryAgent are $300/152 = 1.97$ and $254/100 = 2.54$ respectively. These results are summarized for a different number of agents in table 1

These results highlights the fact that our coordination-policy is more effective than the greedy policy when dealing with a prey that is hard to catch. When using a greedy algorithm to try to catch a prey that is observing the positions of all the adversaries and makes the move that maximizes the distance to all or one of them (like the AvoidingAgent and the AvoidingNearestAdversaryAgent), the adversaries are likely to end up chasing the prey from similar angles. This makes the AvoidingAgent and the AvoidingNearestAdversaryAgent very effective, as there are some moves that are significantly better than others. For example, it if, say, three adversaries are approaching the prey from the left, the prey could simply move right, up or down and easily increase the distance to all the adversaries. If however, the adversaries are coordinated, then the task of the AvoidingAgent and the AvoidingNearestAdversaryAgent become harder. If we assume that the prey and the, say, four adversaries are in a state in the game where the adversaries have perfectly surrounded the prey, then there are not any good move choices for the prey. If the prey moves either to the left, right, up or down, it will move directly toward one of the adversaries. It it does not move, the adversaries will close in on the prey (see section 3.2.6) and eventually catch the prey. This is, of course, a special case in which there are four adversaries where the adversaries can, potentially, position itself such that there is one adversary in all the directions that the prey can move, thus leaving the prey with no good options. This is not true to the same degree when considering the case of fewer adversaries. If there are, say, two adversaries, and they are positioned to the left and right of the prey, the prey could still move up or down and probably escape the adversaries. This could be an explanation for why the average rewards for different number of CoordinatingAgents are not proportional to the number of adversaries. If we take a closer look at the numbers presented in figure 6, figure 7 and figure 8, we see that the ratios of the performance between three CoordinatingAgents and two CoordinatingAgents are $300/135 = 1.96$ and $254/135 = 1.88$ against the AvoidingAgent and the AvoidingNearestAdversaryAgent, respectively. These ratios are both greater than the ratio between three adversaries and two adversaries, which is $3/2 = 1.5$. We see the same trend between the results from the game with four adversaries and the game with three adversaries. Here, those same ratios are $770/300 = 2.57$ and $1138/254 = 4.48$. These ratios are both significantly larger than the ratio between four and three, which is $4/3 = 1.33$. These numbers highlight the fact that coordination performance, in our case, is super-proportional to the number of coordinating agents working together. There was an especially high jump in performance when

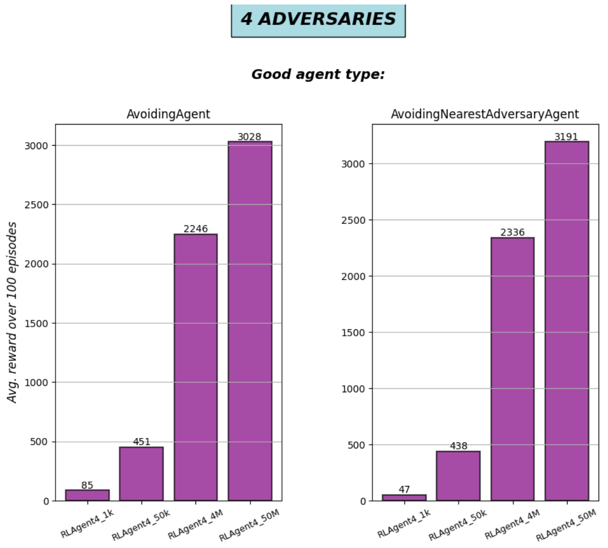


Figure 9: The average 500-episode-rewards for AvoidingAgent and AvoidingNearestAdversaryAgent as prey against four adversaries of type: CoordinatingAgent and RLAgent4.

going from three to four adversaries, which can be explained by the abovementioned scenario where the adversaries can potentially block the prey in all directions.

4.2 RL agents

In addition to our explicitly programmed coordinating and greedy agents, we made an attempt to use RL to train our adversaries and test their performances against those of the other agents and benchmarks.

Figure 9 shows the progression of four adversaries average reward when trying to catch an AvoidingAgent and an AvoidingNearestAdversaryAgent prey while being trained as RL agents for a different number of time steps. After only 1000 steps, the agent performed poorly, getting an average reward of 85 and 47. However, we can clearly see in the figure the progression in performance, and after 50000000 time steps, the RL agent is getting an average reward of 3028 and 3191.

Figure 10 shows the performance of four RL agents trained for 50 million time steps compared to the CoordinatingAgent adversaries. We chose to only include the coordinating agents performances as benchmarks when evaluating the RL agents because they were the most effective of all the explicitly programmed adversary types in catching the AvoidingAgent and the AvoidingNearestAdversary preys. This can be seen in the figures 6, 7 and 8. The performance ratios between the RL agents and the CoordinatingAgents are $3083/768 = 4.01$ and $3140/1114 = 2.82$ when running against an AvoidingAgent and an AvoidingNearestAdversary prey, respectively.

Figure 11 Shows the performance of three RL agents trained for 100 million timesteps compared to the CoordinatingAgent adversaries. As we can see from figure 11, the RL agents outperforms the CooordinatingAgents by a large margin. The ratio between the performances of the RL agents and the CoordinatingAgents are, in this case, $2429/301 = 8.01$ and $2438/258 = 9.45$ against AvoidingAgent and AvoidingNearestAdversaryAgent, respectively.

In figure 12 we can see the performance of two RL agents trained for 50000000 time steps compared to two CoordinatingAgents when going after an AvoidingAgent and an AvoidingNearestAdversaryAgent. Unsurprisingly, the RL agents outperforms the CoordinatingAgents here as well. In this case the ratios between

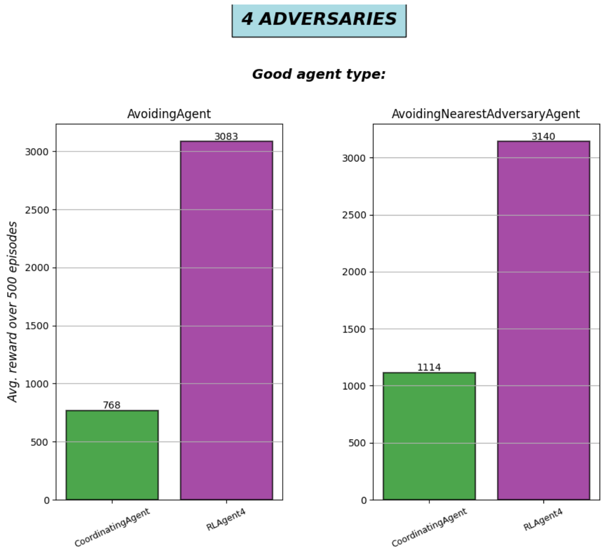


Figure 10: The average 500-episode-rewards for AvoidingAgent and AvoidingNearestAdversaryAgent as prey against four adversaries of type: CoordinatingAgent and RLAgent4.

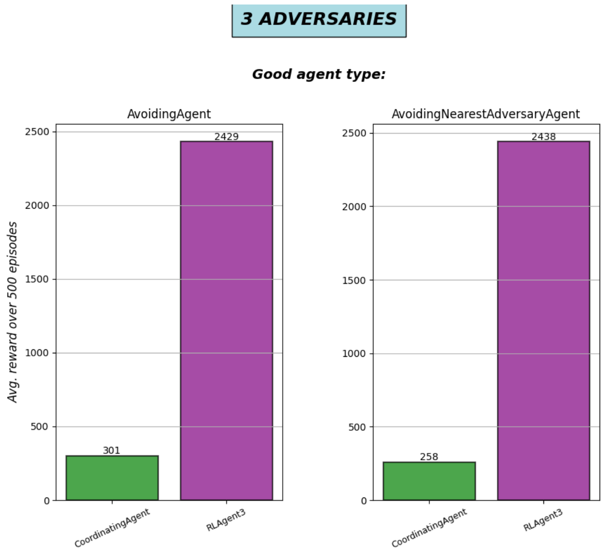


Figure 11: The average 500-episode-rewards for AvoidingAgent and AvoidingNearestAdversaryAgent as prey against three adversaries of type: CoordinatingAgent and RLAgent3.

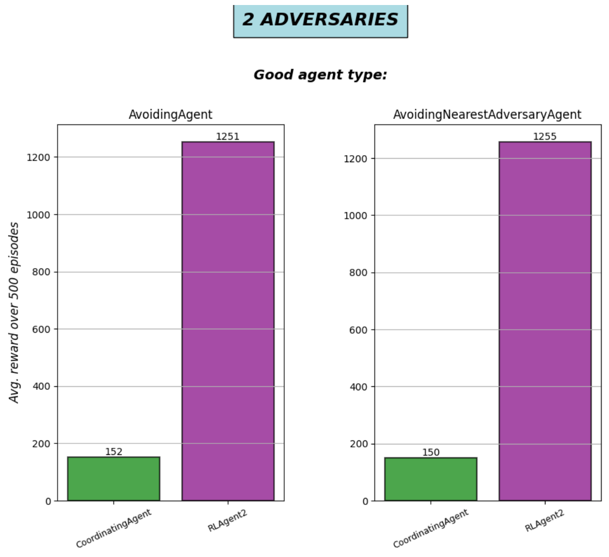


Figure 12: The average 500-episode-rewards for AvoidingAgent and AvoidingNearestAdversaryAgent as prey against two adversaries of type: CoordinatingAgent and RLAgent2.

the performance of the RL agents and the CoordinatingAgents are $1251/152 = 8.23$ and $1255/150 = 8.37$ when being run against AvoidingAgent and AvoidingNearestAdversaryAgent, respectively.

As is clear from these results, the RL agents outperforms all of our explicitly programmed agents with a large margin. One small detail that should be added to this discussion is that the observations returned by the PettingZoo environment not only included the positions of the adversaries and preys, but also the preys velocity. This was something that we did not consider when making our explicitly programmed policies, whereas the RL agent was fed this extra bit of information. This gives the RL agents a slight advantage in terms of information upon which to make a decision about which action to take. However, the difference in performance between the explicitly programmed agents and the RL agents were so large that this detail likely did not make a big difference. Even if we chose to take this extra piece of information into account, we still would have been far outperformed by the RL agents.

References

[1] N. Vlassis. *A Concise Introduction to Multiagent Systems and Distributed Artificial Intelligence*. Morgan & Claypool, first edition, 2007.