Multi Particle Environment Predator-Prey Game - Group 35

André F. de Matos
Instituto Superior Técnico
Lisboa, Portugal

Davi G. Valério
Instituto Superior Técnico
Lisboa, Portugal

João Guilherme R. Câmara
Instituto Superior Técnico
Lisboa, Portugal

ABSTRACT

This project investigates the performance of different predator strategies in a multi-agent predator-prey game. This game was simulated using the "Simple Tag" environment from the PettingZoo Python library. In it, we developed a greedy strategy, social norms, role-based policies, and learned behaviors. In our results, a strategy that intercepts future positions of the prey was the most effective. The greedy strategy also performed well due to flaws in the designed prey policy. The DQN strategy performed poorly, likely due to the high dimensionality of the state space, simplicity of the neural network, lack of a target network, absence of a replay buffer, sparse rewards, or insufficient training. We conclude the study with the need for an improved prey behavior and suggested steps for enhancing the performance of learned policies in the predator-prey scenario.

1 INTRODUCTION

With the increasing demand for autonomous systems capable of adaptive decision-making and cooperative behavior, our project focuses on a multi-agent system deployed within a simulated predator-prey game. By selecting the "Simple Tag" multi particle environment from the PettingZoo library, we create a controlled scenario where agents face conflicting goals and coordination challenges. With three predators pursuing a prey and randomly appearing obstacles, our project aims to explore different agent behaviors, such as simple greedy strategies, policies grounded in social norms theory and learned behaviours. [Papoudakis et al. 2021]

To support these experiments, the theory of Multi-Agent Markov Games (MAMG) was used to provide a mathematical foundation for modeling the interactions between the agents and the environment of the predator-prey game. In MAMG, the agents make decisions based on a Markov Decision Process (MDP) with joint actions. Formally, a MAMG is defined by a state space S, an action space A_i for each agent i, a state transition probability function P and a reward function R_i for every agent i. With this setup, each agent has a policy $\pi_i : S \mapsto \Delta(A_i)$ that maps the current state to a probability distribution in the action space.

This policy may be simple, such as a deterministic function that maps the state to an action, or more complex, such as a Neural Network that learns to optimize a Loss Function. Between these two extremes, social norms and conventions can be used to simplify the decision-making process and promote collaboration between agents. These norms are defined before a game starts, and may include the use of role attribution.

On the more complex side of the spectrum, Reinforcement Learning (RL) theory for multi-agent systems can be used to enable the agents to learn optimal behaviours through the interaction with the environment. Usually, this means minimizing a cost-to-go function

$$J(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t c(s_t, a_t) \mid s_0 = s\right]$$

where $\mathbb{E}$ denotes the expectation; $\gamma, 0 \leq \gamma < 1$ is a discount factor; $c(s_t, a_t)$ is the immediate cost at time step t; s_0 is the initial state; and $\{s_t, a_t\}$ is the sequence of states and actions following policy π.

However, it can be more straightforward to use the Q-function during the implementation of a RL algorithm. It represents the expected cumulative cost of taking a particular action in a given state and subsequently following a specific policy:

$$Q(s, a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \mid s_0 = s, a_0 = a\right]$$

.

One possible implementation of RL is Q-Learning. In this algorithm, the Q-function is updated iteratively using the following update rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$$

where α is the learning rate, which determines how much new information overrides the old information, r is the reward received after taking action a in state s and s' is the new state resulting from taking action a in state s.

Then, the agent selects the action with the highest Q-function value: $a^* = \arg\max_a Q(s, a)$. However, Q-Learning only converges to the optimal policy if all state-action pairs are visited infinitely often. This poses two challenges: first, balancing exploration of the environment and the exploitation of the found policy; and second, dealing with very large state spaces.

To balance exploration and exploitation, a common strategy is to use an ϵ-Greedy Policy that introduces randomness and promote exploration. With this strategy, a random action is chosen with probability ϵ, while a^* is chosen with probability $1 - \epsilon$.

Dealing with large state spaces is a bit harder, and can be done by applying Deep Q-Learning, which is an extension of the Q-Learning. In it, instead of using a table to represent the Q-function, it is approximated using a neural network $Q(s, a; \theta)$ parameterized by by θ. It takes the state s as input and outputs Q-values for all possible actions a. The simplest form of Deep Q-learning optimizes the network to minimize the Mean Squared Error between the Q-value of the current state and action and the one given by the network. More complex implementations may include a transition replay buffer that stores the states and actions to stabilize training and a second network that is used as the target.

However, the convergence of Q-Learning relies on a stationary transition model P. At the same time, each agent is being learned individually, which means that they do not consider the actions of other agents during training. Thus, they are perceived as part

of the environment, (i.e. P). As a result, since multiple agents are learning simultaneously, the environment is constantly changing, which may lead to bad policies. Despite this, Q-Learning is used in practice for Multi-Agent Systems, achieving some success.

Supported by this theoretical background, the goal of this project is to investigate the efficacy of different multi-agent architectures and decision-making algorithms in solving a complex coordination problem: the predator-prey scenario. Through empirical evaluation, we seek to assess the strengths and weaknesses of different strategies, offering insights into the design and implementation of intelligent systems capable of navigating dynamic, cooperative environments.

2 METHODS

To test different multi-agent policies in the predator-prey scenario, we designed and implemented a series of experiments using the "Simple Tag" multi-particle environment from the PettingZoo library. We will design a fixed policy for a prey and several strategies for predators. Then, we will evaluate the performances of the different behaviours, and try to identify if collaboration emerged from the policies.

2.1 Environment

The predator-prey game can be modelled as a Markov Game with four agents: three predators that are collectively awarded when they hit the prey and one fourth agent that is negatively rewarded when hit by one of the predators. It is faster than the predators and is also penalized for going outside of the area of the game. In addition, two static obstacles appear randomly in the map at each run. Each run has a limit of 25 steps, and the goal of the prey is to avoid being hit for the duration of the game. [Vlassis 2009].

Since the prey is faster than the predators, they will have to cooperate to consistently catch the prey. Thus, this environment has enough complexity to enable experiments with several multi-agent behaviour architectures.

The state space S consists of the following components: the position and velocity of each agent, including each predator's position (x_i, y_i) and velocity (v_{xi}, v_{yi}) for $i = 1, 2, 3$, and the prey's position (x_p, y_p) and velocity (v_{xp}, v_{yp}). Additionally, it includes the position of obstacles, specifically the positions of two obstacles (o_1, o_2). Formally, the state space S is defined as: $S = (x_1, y_1, v_{x1}, v_{y1})$, $(x_2, y_2, v_{x2}, v_{y2})$, $(x_3, y_3, v_{x3}, v_{y3})$, $(x_p, y_p, v_{xp}, v_{yp})$, (o_1, o_2)

The actions will jointly update the state of the game with a stochastic transition model $p(s'|s, a)$. The action space A can be defined as:

$$A = \{\text{Move up, Move down, Move left, Move right, Do nothing}\}$$

The observation space for each predator is defined similarly to the state, the main difference being that the velocities of other predators are not observed. For the prey, and it does not observe the predators velocities.

The reward function gives agent i reward $R_i(s, a)$, where s is the current state and a the joint action of all the agents. The reward for the predators is shared and can be defined as:

$$R_{\text{predator}}(s, a) = \begin{cases} 10 & \text{if any of the predators catches the prey} \\ 0 & \text{otherwise} \end{cases}$$

The prey is penalized if it leaves the bounds of the environment according to the function:

$$\text{bound}(x) = \begin{cases} 0 & \text{if } x < 0.9 \\ (x - 0.9) \times 10 & \text{if } 0.9 \leq x < 1.0 \\ \min(e^{2x-2}, 10) & \text{if } x \geq 1.0 \end{cases}$$

Additionally, if the prey is caught by any predator, it will receive a penalty of -10. The total reward for the prey is the sum of the penalties for leaving bounds and for being caught by a predator.

The probability function p determines the likelihood of transitioning from state s to state s' when action a is taken. It is formally defined as $p(s' \mid s, \mathbf{a})$, where s represents the current state of the environment, s' represents the next state of the environment, and $\mathbf{a} = (a_1, a_2, a_3, a_4)$ represents the joint action of all agents, with each a_i being the action taken by agent i. It is important to note that the PettingZoo environment defines this probability function through a physics engine. In the code, the actions generate an acceleration that causes the velocity change. As a result of this approach, the agents have inertia and bounce when they collide with each other or with obstacles in the environment.

2.2 Prey's strategy

The prey's policy was fixed since the focus of the experiments was on testing different predator strategies. To confine it within an approximated square around $x, y \in -1 \leq x \leq 1, -1 \leq y \leq 1$, the policy adjusts the probability of each action based on the prey's current position and the relative positions of the adversaries. The prey's current position, x and y, is taken from its last observation. An array of probabilities, initially set to 0.2 for each of the five possible actions, is modified according to the minimum distances to the closest adversaries in each direction (up, down, left, right). The probabilities for taking each action are set based on the distances of the closest adversary in each direction, favoring movements away from nearby adversaries. Additionally, if the prey is close to the boundaries of the square (i.e., $x > 0.9$ or $x < -0.9$, $y > 0.9$ or $y < -0.9$), the probability of moving further out of bounds is set to zero. Finally, the probabilities are normalized to ensure they sum to one, creating an coherent action distribution for the prey.

2.3 Predators' strategies

Several strategies were designed for the predators:

(1) **Random Predator:** Takes random actions from a uniform distribution of the action space. Its goal is quantifying randomness caused by the initial state and by the environment obstacles.
(2) **Greedy Predator:** Represents the baseline greedy behavior where the predator always moves towards the prey.
(3) **Intercept Predator:** Smarter than greedy strategy that calculates positions based on the prey's velocity and relative position to predict and intercept the prey's future location.
(4) **Distract-Pursue Predator:** Applies different roles to predators, such as distracting and pursuing, to improve the efficiency of catching the prey.
(5) **Surround Predator:** Attempts to surround the prey by coordinating movements with other predators with social norms

and roles with the goal of forming an equilateral triangle around the prey.

(6) **DQN Predator:** Implements a Deep Q-Network (DQN) approach to learn and adapt predator strategies through reinforcement learning.

The non trivial policies can be formalized as follows:

2.3.1 Greedy Predator. This predator calculates the distances dx and dy from its current position to the prey's position using the last observation. The action is initially set to 'no_action'. If the absolute horizontal distance $|dx|$ is greater than the absolute vertical distance $|dy|$, the predator will prioritize horizontal movement. If $dx < 0$, indicating the prey is to the left, the predator will choose the action 'move_left'. However, if $dx > 0$, indicating the prey is to the right, the predator will choose 'move_right'. If the vertical distance is greater, the predator will prioritize vertical movement. If $dy < 0$, indicating the prey is below, the predator will choose the action 'move_down'. If, however, $dy > 0$, indicating the prey is above, the predator will choose 'move_up'. In addition, if both $|dx|$ and $|dy|$ are very small (less than 0.1), indicating the prey is very close, the predator will choose 'no_action' to avoid unnecessary movement. This strategy ensures that the predator moves in the direction that reduces the distance to the prey in a greedy way.

2.3.2 Intercept Predator. Its strategy is designed to predict and intercept the prey's future position by calculating an intercept point based on the prey's current position and velocity. The predator first determines the prey's absolute velocity by combining its own observed velocity components (vel_x, vel_y) with the relative velocity of the prey (ag_relvel_x, ag_relvel_y). The predator then calculates the intercept range, which depends on its distance ranking relative to other predators. This range is determined by the index of the predator in a sorted list of distances to the prey, multiplied by a scaling factor intercept_mult. The intercept position is calculated by projecting the prey's current position (ag_relpos_x, ag_relpos_y) forward by the product of the prey's velocity and the intercept range. The predator then selects an action to move towards this intercept position. If the horizontal distance $|dx|$ is greater than the vertical distance $|dy|$, the predator moves left or right to minimize dx. However, if the vertical distance is greater, the predator moves up or down to minimize dy. If both distances are very small (less than 0.1), the predator chooses 'no_action' to avoid unnecessary movement. This strategy ensures that the predator moves in a calculated manner to intercept the prey's predicted future position.

2.3.3 Distract-Pursue Predator. The Distract-Pursue Predator starts by defining its role based on its initial distance to the prey and distances of other predators. If the predator is the closest to the prey, it assumes the role of "Pursuer"; otherwise, it becomes a "Distractor". The "Pursuer" always as a greedy behaviour and follows the prey, whereas the "Distractor" alternates between getting closer to the prey and choosing its movements randomly when it is close to the target.

2.3.4 Surround Predator. This strategy uses social norms and roles to coordinate multiple predators to surround the prey by forming an equilateral triangle with the prey at its centroid. The predator first calculates the distance between its current position and the prey's position using the Euclidean distance. The strategy assigns roles to the predators based on their proximity to the prey, with the closest predator designated as the "prey" follower. The remaining two predators are assigned to the vertices of the triangle.

The coordinates of these vertices are determined by rotating the vector from the prey to the follower by $\pm 60^\circ$. These rotations are represented by rotation matrices:

$$+60^\circ = \begin{bmatrix} \cos 60^\circ & -\sin 60^\circ \\ \sin 60^\circ & \cos 60^\circ \end{bmatrix},$$

$$-60^\circ = \begin{bmatrix} \cos 60^\circ & \sin 60^\circ \\ -\sin 60^\circ & \cos 60^\circ \end{bmatrix}$$

The roles are assigned by finding the predators closest to each of these vertices. Each predator then targets its respective vertex in a greedy manner.

2.3.5 DQN Predador. The DQN Predator strategy utilizes a Deep Q-Network (DQN) to learn optimal actions through Q-learning. The Q-network consists of three fully connected layers, with ReLU activation functions between them, mapping input observations to Q-values for each possible action. The predator's decision-making process involves exploring the environment with an ϵ-greedy policy, where a random action is selected with probability ϵ, and the action with the highest Q-value is selected otherwise.

The learning process involves updating the Q-network based on the observed state, action, reward, and next state. The loss between the current Q-values and the target Q-values is minimized using the Mean Squared Error (MSE) loss function, and the network parameters are updated using the Adam optimizer. The train of the neural network was done by simply saving the weights at the end of each run for the duration of 5000 games.

Finally, this predator receives an extra small reward to encourage movement towards the prey. When its euclidean distance to the prey decreases, it receives a reward of 1. This is ten times smaller than the one received when a capture occurs.

2.4 Code Structure for the experiments

We used an Object Oriented Programming approach to design a wrapper around PettingZoo's basic environment and make it easier to test different predators. The `BaseActor` class serves as a parent component for all agents in the environment. It stores the last observation along with the observation, action, and reward histories. Inheriting from `BaseActor`, the `BaseAgent` class parses each observation into its components, such as x and y velocity, self position, and positions of obstacles and adversaries. In a similar way, the `BaseAdversary` class, is responsible for parsing the observation of predators, which are different from the agent's. The prey inherits from `BaseAgent`, while all the predators are childs of `BaseAdversary`. Note that in the repository there are other prey classes, which will not be covered in this report.

The `Simulation` class orchestrates the simulation runs. It initializes the environment, selects the appropriate prey and predator classes, executes actions for each step, calculates performance metrics, and resets the environment for the next run.

The `main.py` script serves as the entry point for running multiple simulations and visualizing results. It configures the number of

simulations to run, selects behavior strategies, and generates plots that allow to analyse the performance of the defined policy.

2.5 Experiments

To test each predator strategy, each one played against the fixed prey for 100 games with 25 steps. The percentage of games where there was a catch was recorded in a bar chart. The total number of catches and the time it took for the first catch were recorded and plotted in box-plots. In addition, to visualize the performance of the different strategies in the long run, 50 simulations with 1000 steps were run for the top performing policies.

2.6 Results

In Figure 1, the percentage of games where a catch occurred for each predator behaviour can be seen.

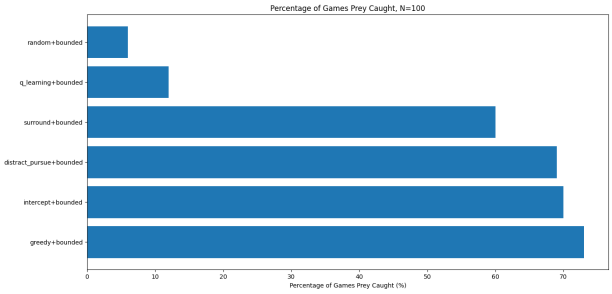

Figure 1: Percentage of games where the prey was caught for each predator strategy.

In Figure 2, box-plots of the number of catches for each strategy can be analysed.

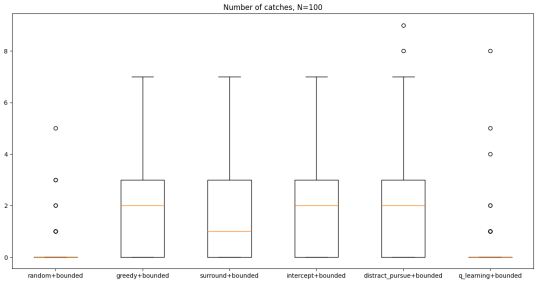

Figure 2: Box plots showing the number of catches for each predator strategy.

In Figure 3, box-plots of the number of steps needed for the first catch are shown.

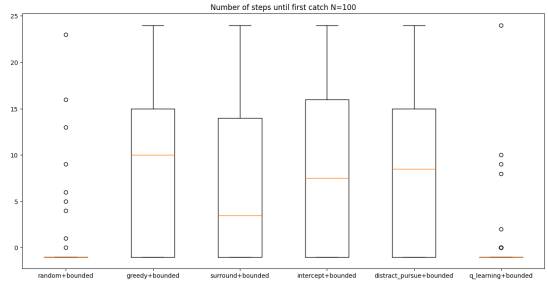

Figure 3: Box plots showing the number of steps needed for the first catch for each predator strategy.

Finally, the results of the number of catches over the long run can be seen in Figure 4.

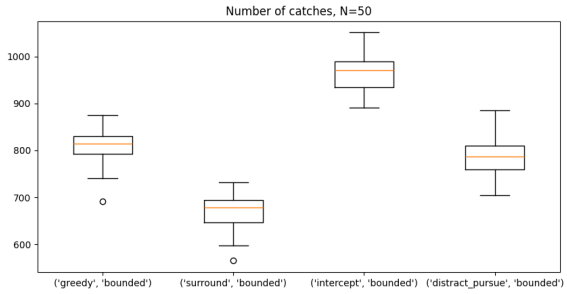

Figure 4: Box plots showing the number of catches for each predatory strategy over long runs.

2.7 Discussion

It is known that in predator-prey scenarios, the optimal prey is never caught by greedy predators. However, in our experiments, two elements contributed to the prey being caught. First, the prey was not optimal, and its current policy had flaws that were naturally exploited by greedy agents. Second, the randomly generated obstacles could trap the prey, making it easier to be caught.

In the bar chart, it is evident that the greedy predator shows the best performance, capturing the prey in more than 70% of the games. This strategy is closely followed by the intercept and distract-pursue behaviors. Despite being more complex than the first three, the surround strategy did not lead to better performance. The top three performing strategies (greedy, intercept, and distract-pursue) all have a median number of catches of two, with some instances showing up to seven catches.

Also, it can be noted that the number of steps for the intercept behavior is lower than that of the other two top-performing predators, which was expected. This indicates that the best policy is the intercept strategy, as it achieves a similar number of catches but does so more quickly.

A qualitative analysis of the surround strategy, which was supposed to form a triangle with the prey at its center, shows that

due to collisions between the agents and with the environment obstacles, the adversaries rarely surrounded the prey as intended. When they did, they were sure to catch it. However, this policy did not perform as expected, adding complexity but showing results worse than the greedy strategy.

In the long run, both greedy and distract pursue behaved similarly, with around 800 average catches. As predicted by the other plots and confirmed by our analysis of the behaviors, surround fell under 700 catches. Intercept was also once again proven to be the best strategy surpassing 950 catches, which was an unexpectedly big margin.

The biggest surprise in the analysis was the performance of the Deep Q-Learning (DQN) strategy, which showed results similar to the random behavior. Several hypotheses can be made for this failure.

First, the inherent problem of using an algorithm designed for single-agent systems in Multi-Agent Markov Games (MAMGs) might mean that this game is a case where good results with Q-Learning cannot be expected. The state space size (18 continuous variables) suffers from the curse of dimensionality and may be too large for Deep Q-Learning to handle effectively. The fitted network might be too simple to capture the game's complexity. The absence of a separate target neural network might have led to exploding gradients, causing the algorithm to diverge. The lack of a replay buffer and the stochastic use of steps might have made training unstable. The sparsity of rewards could have made learning difficult, and adding a small intermediate reward was insufficient. The training duration of 5000 games might have been too short for effective learning.

The authors believe that a combination of these factors contributed to the failure of Deep Q-Learning in this scenario.

2.8 Conclusion

To conclude, it can be said that the prey was not well-designed for the tests. As a consequence, the greedy adversary performed well, which is not expected in predator-prey scenarios. As a result, improvements must be made in the prey's behavior to allow for better evaluation of different adversaries. Finally, the failure of the learned DQN policy shows that more tests are needed to assess the most significant causes and orient future experiments with learned policies. Potential improvements could include using multi-agent reinforcement learning algorithms designed for MAMGs, increasing the complexity of the neural network, incorporating a separate target network, using a replay buffer, improving the reward function, and extending the training duration to achieve better results.

REFERENCES

Georgios Papoudakis, Filippos Christianos, Lukas Schäfer, and Stefano V. Albrecht. 2021. Benchmarking Multi-Agent Deep Reinforcement Learning Algorithms in Cooperative Tasks. arXiv:2006.07869 [cs.LG]

Nikos Vlassis. 2009. *A Concise Introduction to Multiagent Systems and Distributed Artificial Intelligence*. Vol. 1. https://doi.org/10.2200/S00091ED1V01Y200705AIM002