Implementation project of Multi-Agent Reinforcement Learning: patrolling agents

Adrien Benamira
CentraleSupélec
adrien.benamira@supelec.fr

Benjamin Devillers
CentraleSupélec
benjamin.devillers@supelec.fr

Abstract

We explore deep reinforcement learning methods in the multi-agent setting. In this project report, we apply Deep Q Network (DQN) [1], Double Deep Q learning (DDQN) [2] and Multi Agent Deep Deterministic Policy Gradient (MADDPG) [3] to a multi-agent environment build for competitive and cooperative scenarios. In particular, we test the effectiveness of these methods in the classic predator-prey game, with multiple different facets: two and three dimension, infinite world, obstacles,... The results for the DQN and the DDQN approach are surprisingly good, whereas the MADDPG algorithm seems to be not well suited. Some gifs are available on our GitHub project page.

1 Introduction

Recently, some attempts have been made to use machine learning to try and predict where crimes will happen in a city[1]. They leverage for instance the location information given by CCTV cameras. When the certainty of a crime is high, the system can dispatch a number of officer on the scene to try to prevent the crime to happen by discouraging the troublemaker to commit the act. However, when the certainty is low, a possible way to handle the problem is to send some drones to the scene to gain more information.

In this work, we will model a crime scene where a troublemaker (prey) and several officer drones are evolving (predators). We also experiment with a 3D world, which is interesting to model UAV attacks for example. It would also be possible to extend the model for nautical combat scenarios between submarines and ships. For this purpose, we leverage the advances in deep multi-agent reinforcement learning (MARL) to tackle our "prey/predators" problem.

2 The Predator-Prey Environment

We develop an environment for a predator-prey game with obstacles and 3D possibilities. Two kinds of agents are evolving in the environment. We denote by "prey" (blue) and "predator" (orange) the two types of agents. We want to teach the predators to follow and catch the prey(s). Multiple agents can exist on either team and the goal of each agent is to maximize its own reward in this mixed cooperative and competitive setting. In the simplest modelisation, the agents are set in a closed square environment where there are no obstacles and every agent has the same observation of the environment: the position of every agents.

[1] https://www.newscientist.com/article/2186512-exclusive-uk-police-wants-ai-to-stop-violent-crime-before-it-happens

2.1 States

We define our state space by $\mathcal{S} = [0,1]^{3N}$ where N is the number of agents in the environment. Therefore, $S \in \mathcal{S}$ represents the normalized 3D coordinates of the N agents. In most of our simulations, the z coordinate is fixed to 0 and the agents are forced into the 2D plane. We also experiment with agents progressing in a 3D environment. This would be interesting for flying drones for instance. To simplify the problem, we discretize the state space to allow only a fixed number of 3D (or 2D) positions and we can therefore consider $\mathcal{S} = \{1, 2, \cdots, N_s\}^{3N}$, where N_s is the number of steps we use in our discretrization. In most of our experiments, we use a discretization of 15 steps along both axis which gives 225 possible states in a 2D environment.

Moreover, as we can also add some obstacles on the board, the 2D coordinates of the obstacles are also added to the state.

2.2 Actions

Each agent is free to move in the four main cardinal directions or stay put (immobile). That is to say, the action space for each agent is $\mathcal{A} = \{\text{none, front, left, right, back}\}$.

If the agents progress in a 3D world, we add two new actions so that $\mathcal{A} = \{\text{none, front, left, right, back, top, bottom}\}$.

The agents cannot escape and are forced to stay in the environment: if an agent decides to do an action which would make him leave the boundaries, the environment forces the agent not to move. We also added some noise on the actions: with probability 0.1%, the action does not execute and a random position is assigned.

2.3 Reward function

For all reinforcement learning algorithms, we use a non-sparse reward function. We try with several versions of the reward function $r_i : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ where $i \in \{1, \cdots, N\}$ represents the agent and $r_i(s, a, s')$ is the reward given to agent i for the transition (s, s') done with action a. We experiment with a reward which only depends on s' and we normalize the rewards such that $r_i(s') \in [-1, 1]$.

2.3.1 Reward base

For a predator at position $\boldsymbol{X} = (x, y, z)$ (z being potentially 0 in the case of a 2D world). If $\boldsymbol{X}_p = (x_p, y_p, z_p)$ is the position of the prey closest to the predator, we define

$$r_{\text{predator}}(\boldsymbol{X}, \boldsymbol{X}_p) = e^{-c\|\boldsymbol{X} - \boldsymbol{X}_p\|^2} \tag{1}$$

the reward function. We use a constant $c = 5$ in our experiments.

For the prey, we use a similar reward.

$$r_{\text{prey}}(\boldsymbol{X}, \boldsymbol{X}_p) = 1 - 2e^{-c\|\boldsymbol{X} - \boldsymbol{X}_p\|^2} \tag{2}$$

2.3.2 Add-on 1: "unity is strength"

To incentivise the predator to catch as many preys as possible, we add a constant $r_{\text{incentive}}$ to all the predators if one catches a prey. This incentive is particularly usefull in the case where there are several predators and preys at play.

2.3.3 Add-on 2: "walls on fire"

We added a possibility to sanction the agent for trying to leave the board. An agent receives a -1 reward, regardless of its proximity to any other agent.

2.4 Modularities in the environment

The advantage of having coded the environment is that we can easily adapt it. We added some extensions to the default environment. In addition to the 2D/3D possibilities, we added an "infinite

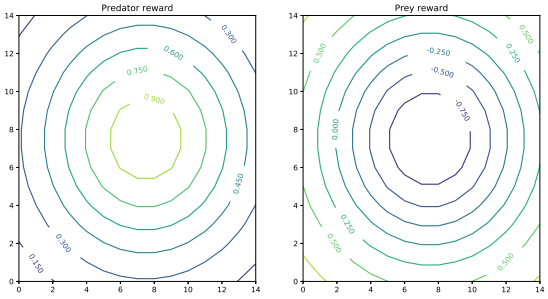

Figure 1: Reward function for a predator (left-hand side) and a prey (right-hand side) when an ennemy is located at the center.

world" scenario where an agent going far right would end up at the left side of the board (the other directions also behave similarly). Obstacles can also be added to hinder the movements of the agents.

3 Multi-Agent Reinforcement Learning

We defined the environment properties in the previous section (state space, action space and reward function).

The reinforcement learning problem here is to find a policy $\pi_i : \mathcal{S} \to \mathcal{A}_i$ for every agent ($1 \leq i \leq N$) maximizing the state Value function $V_i^{\pi_i} : \mathcal{S} \to \mathbb{R}$ associated with the policy. We define for this purpose the state-action value function $Q_i : \mathcal{S} \times \mathcal{A}_1 \times \cdots \times \mathcal{A}_N \to \mathbb{R}$ for every agent.

In the following, $\mathcal{A}$ will denote the action space for every agent as they all share the same action space.

The difficulty in a multi-agent setting (compared with a unique agent one) is that the action space is exponential with the number of agents.

Besides, the state space is defined as $\mathcal{S} = \{1, 2, \cdots, N_s\}^{3N}$ as explained in section 2.1. This exponentially large state space with respect to the number of agents, combined with the fact that the Q function depends on the actions of all agents is the reason why the task of finding the optimal policy difficult.

3.1 Deep Q-learning

Deep Q-learning [1] (DQN) algorithm has proven to be an efficient way to do Q-learning.

As a first approximation, we simplify our problem by making the agents independent to one another. We therefore write $Q_i : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ and model in our implementation[2] the Q functions with a simple 3 layers linear neural network with ReLU activation functions. Each agent has its own Q function which only depends on the the state and its action.

The motivation of the DQN algorithm is that neural networks have proven capable to approximating functions in a robust way [4]. Moreover, it uses the bellman equation and similarly to the value iteration algorithm, it uses two version of the same network at 2 different times: the "policy network" (which would relate to Q_{t+1} in the value iteration algorithm) and the "target network" (which would relate to Q_{t+1}) and teaches the networks to verify the bellman equation by minimizing the temporal difference.

Moreover, we use a memory replay buffer (containing 10 000 samples in our instance) to learn the policy without having a complete non-stationary setting.

[2]Code is available here: `https://github.com/bdvllrs/marl-patrolling-agents`

3.2 Double DQN

In Q-learning, the optimal policy can be deduced once the Q function is known using

$$\pi_i^\star(s) = \arg\max_{a \in \mathcal{A}} Q(s, a)$$

However, during learning the selected policy is obtained by $\pi_i(s) = \arg\max_{a \in \mathcal{A}} Q_t(s, a)$ where Q_t is the approximation of the Q function at time t. As a consequence, it overestimates the value-action function and in fact, in noisy environments, learning is slow. That is why [2] introduced Double Q-learning. In practice, the action selected in the target network are not selected by taking the action maximizing the target network as previously but by the action maximizing the policy network.

3.3 Multi-Agent Deep Deterministic Policy Gradient

To remove the approximation used for DQN learning on the action space, [3] uses an Actor/Critic setting which uses both a policy learning method and Q-learning.

3.3.1 Policy Gradient

In policy-based methods, instead of learning a value function that tells us what is the expected sum of rewards given a state and an action, we learn directly the policy function. It means that we directly try to optimize our policy function π_i without worrying about a value function. The policy is learned to maximize the objective

$$J(\theta) = \mathbb{E}_{s \sim \mathcal{S}, a \sim \pi_{i,\theta}} \left\{ \sum_{t=0}^{T} \gamma^t r_i(t) \right\} \tag{3}$$

by taking steps in the direction of $\nabla_\theta J(\theta)$ (θ represents the parameters of our model). We can them approximate the expectation by averaging values from our replay memory buffer.

There are several advantages of Policy Gradient over value-based techniques, most notably that they are effective in high-dimensional or continuous action spaces, converge faster and can learn stochastic policies.

Indeed, in deep Q-learning, the number of action needs to be finite as we compute $Q(s)$ and then perform the optimization problem by simply taking the highest value among all actions. If the number of action is important or infinite (in the case of continuous actions), this cannot be achieved.

Policy Gradient methods can be naively applied to multi-agent reinforcement learning problems. However, it is difficult to train because the reward of each agent is highly dependent on other agents' actions and so the drawbacks can be too big.

Actor Critic algorithm tends to avoid failures of Policy Gradient Algorithm like REINFORCE. It uses two neural networks: a critic that measures how good the action taken is (value-based and an Actor that controls how our agent behaves (policy-based). Instead of waiting until the end of the episode as we do in Monte Carlo REINFORCE, we make an update at each step. And because we do an update at each time step, we can't use the total rewards. Instead, we need to train a Critic model that approximates the value function.

3.3.2 Mutli-Agent Deterministic Policy Gradient (MADDPG)

MADDPG uses Policy Gradient as well as a Actor / Critic setting. This allows to remove the approximation made with DQN where the Q function of each agent is independent of the action of the other agents.

For each agent, the goal is to learn the Q function (critic) as well as the policy (actor). The critic network now takes the state and the action of all the agents to determine the cumulative reward. Another network (actor) approximate the policy and gives the best action given the state.

Additionally, they introduce a training system utilizing an ensemble of policies for each agent that leads to more robust multi-agent policies and they learn a separate centralised critic for each agent and is applied to competitive environments without addressing multi-agent credit assignment.

Also, the MADDPG is build on top of the DDPG algorithm which is designed for continuous action space. An extension can be made for discrete action spaces using the Gumbel-Softmax distribution [5].

4 Experiments and results

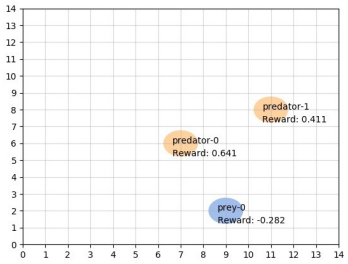

Figure 2: Example of our 2D environment

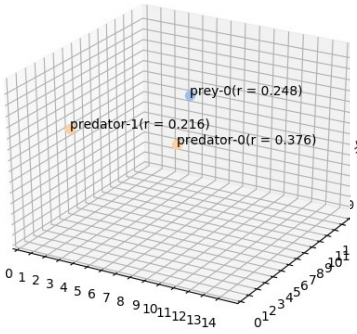

Figure 3: Example of our 3D environment

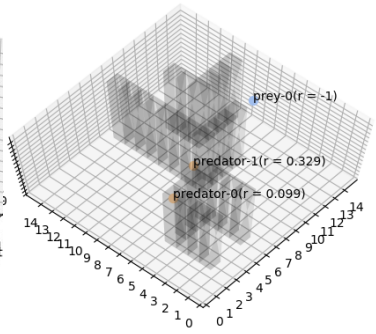

Figure 4: Example of our 3D environment with obstacles

We trained our reinforcement learning models for 2 scenarios with the 3 algorithms (DQN, DDQN and MADDPG). We denote by scenario 1 a setting with 2 predators and 1 prey and by scenario 2 a setting with 2 predators and 2 preys.

We also added some variants (3D world, infinite world, obstacles). For DQN and DDQN, because they learn very fast we train during 1500 episodes. For MADDPG, we set 50000 episodes where one episode is 50 steps long. We set the discount factor γ to 0.9. We do an epsilon greedy exploration policy with $\varepsilon = 0.9$ at first, then decaying to 0.1 after 10000 learning step. The learning rate is set to 0.01 and the frequency for updating the target net (hard update) to every 20 learning step. The maximum size of the replay memory buffer is 10000 and we shuffle batches among the elements in the memory. We also add some noise on the agents' actions and they are not successful with a probability of 0.1%. We use a batch size of 200 for learning. We plot the mean of the reward (averaged over 100 episodes) and the loss averaged on 100 episodes.

4.1 Scenario 1: 2 predators vs 1 prey

Figure 5, 6 and 7 show the losses for each agent during the learning process for respectively the DQN, DDQN and MADDPG algorithms. They also show the expected cumulative reward for each agent averaged over 100 episodes.

We see that for the three algorithms, the expected return for the predators tends to ~ 4 and to ~ 0 for the prey. However, when we look at the effective results and watch the episodes, we observe that the agents trained with DQN and DDQN are much more effective than the ones trained with MADDPG. Indeed, the agents trained with MADDPG do not chase one another. However the prey is chased by the predators when the agents are trained with DQN and DDQN.

We believe that the learning with MADDPG is harder to train because it combines a policy gradient method which is harder to train because it depends highly on the other agents action. Moreover the Q function depends now on the state and on the action predicted by all the actor networks of the other agents. All of this dependence makes the learning difficult and would therefore need more time to train.

We also tried with several action forms to put in the critic network:

- Onehot vector for the 5 (or 7) actions,
- A value between 0 and 5 (or 7),
- A normalized value between 0 and 1.

The three versions yield similar results and we curves presented here are computed using a normalized value of the action.

Besides, for better comparison, we would also need to add the number of times a predator caught the prey and add this as a metric. However this has not been implemented yet.

Figure 5: Losses and returns for each agent with DQN for scenario 1

Figure 6: Losses and returns for each agent with DDQN for scenario 1

Figure 7: Losses and returns for each agent with MADDPG for scenario 1

4.2 Scenario 2: 2 predators vs 2 preys

Figure 10, 9 and 10 show the losses during the learning process for respectively the DQN, DDQN and MADDPG algorithms. Similarly to the scenario 1, we see the expected return converging to around ~ 5. However, the effective result is similar to scenario 1 and the MADDP algorithm does not show agents chasing one another.

Figure 8: Losses and returns for each agent with DQN for scenario 2

Figure 9: Losses and returns with DDQN for each agent for scenario 2

Figure 10: Losses and returns for each agent with MADDPG for scenario 2

4.3 Other Configurations

We also experiment with 4 other configurations from our modularities on scenario 1. In all cases, we use the DDQN algorithm as it performs better than the MADDPG algorithm. We tried :

- Obstacles (fig 11): the agents do not really learn where the walls are but overfit on certain positions to avoid. To prevent this, random obstacles could be added.
- 3D representation (fig. 12) : Results are very similar to 2D. Only the average of the returns is higher and the time needed to learn is more important.
- Reward add-on 1: walls on fire (fig. 13). Results are exactly the same as before. Naturally the agents learn to avoid to be stucked on walls (for the preys) and predator will g
- Reward add-on 2: unity is strength (fig. 15). Results are exactly the same as before. This proof that they do not act as a team. We can maybe force add by adding a penalty for the effort produce to get the prey.
- Infinite world (fig 14). It is more difficult for the agent to learn.

Otherwise, further work can be to test scenario where prey and predators do not have the same algorithm.

Figure 11: Losses and Returns for the obstacle configuration

Figure 12: Losses and Returns for 3D configuration

Figure 13: Losses and Returns for reward add-on 1: walls on Fire configuration

Figure 14: Losses and Returns for the infinite world configuration

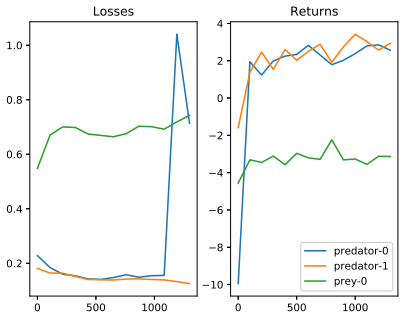

Figure 15: Losses and Returns for the reward add-on 2: unity is strength configuration

5 Conclusion

For this implementation project of Multi-Agent reinforcement learning we coded a full environment patrolling agents in different scenarios. We use 3 different algorithms: DQN, DDQN and MADDPG. We evaluated their performances with a vaste diversity of scenarios. Our results with DQN and DDQN are very impressive: predators learn well and fast to chase the preys and preys however do not learn as well as the predators. Further work is needed to help the preys learn. This can be achieved by making them faster of change the reward function for instance.

Despite many attempts (Gumbel softmax, one hot encoded actions, we also recoded multiple times the algorithm from scratch to see if our implementation was correct), the expected returns are similar, visualy our implementation of MADDPG does not work, even on simple scenarios. One hypothese is that because MADDPG takes the states and actions of all agents into account, it require more hyperparameter tuning and longer training time. To diagnostic the issue, further work can be to plot the number of time the predator caught the prey and to make the scenario even more simple – like putting a static prey for example.

References

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

[2] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, volume 2, page 5. Phoenix, AZ, 2016.

[3] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments.

[4] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[5] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.