CentraleSupélec

Multi-Agent System with Reinforcement Learning Using Communication

Graduation Internship Report
June 2020 - December 2020

Antonin DUVAL
MSc in Artifical Intelligence

School Supervisors:
Wassila Ouerdane
Céline Hudelot

Company Supervisors:
Alexandre Kazmierowski

Contents

Acknowledgements

I would like to thank, first of all, my company mentor, Alexandre. He was a great help during all my internship, showed me all I needed to know about SE-Star. He followed my project from the start and was able to give me precious insights that saved me from getting too lost in my research.

I would also like to thanks Nizam and Ludovic, my colleagues at ThereSIS. Their passion and knowledge in RL was a source of inspiration. Moreover, it was always a pleasure to discuss with them and see friendly faces during the time of containment. I wish them the best.

I would like to thank Jérôme, our team leader for his help during those 6 months. He was always attentive to my progress and very supportive. His vision for the future of reinforcement learning inspired me and motivated me to pursue in this direction.

I am very grateful to my academic tutor, Wassila Ouerdane, which gave me numerous advises about my internship, and always responded quickly to my interrogations. I want to also thank Céline Hudelot and Vincent Mousseau for this Master, and for the time they will take to read this report.

Finally, I want to thank all my fellow comrades from the MSc at Centrale Supelec. They are all passionate people about deep learning with various backgrounds. Working with them along the year was a real pleasure, and I learnt a lot from them. I will definitely miss our passionate talks about AI, but I am sure we will keep seeing each other in the future.

Abstract

Reinforcement learning(RL) has seen a recent growth and has been applied in many domains, from robotic to game playing. Most of the successes of RL have been however in single agent domains, where modeling or predicting the behavior of other actors in the environment is largely unnecessary. However, there are many applications that implies interactions between agents that are co-evolving together, for example, robot swarms or autonomous cars. Those agents could also communicate with each other, creating complex behaviors. As such, successfully scaling RL to environments with multiple-agents is crucial to building artificially intelligent systems that can productively interact with humans and each other.

At Thales, our team **ThereSIS** is interested in building a large library of state-of-the-art algorithms in RL that can be use afterwards for solving projects where our help and knowledge in AI is requested. Our contribution in this internship is two fold.

First, following a recent paper, we implemented COMIX, a state of the art MARL algorithm that work on continuous action space. To find the best hyper parameters, we trained our agents on a toy problem, the *multiparticle environment*. Once we were satisfied with the results, we tested it on a Thales environment suited for drone's control. We compared our result against other similar state-of-the-art algorithms.

One a second part, we got interested in how recent approaches were implementing a sense of abstraction to the agents, allowing them to better interact witch each other. We focused on those techniques to improve our algorithms, both in term of sample efficiency and in performance, by adding a communication network. We show that by allowing the agents to exchange messages, we are able to get new behaviors that show a deeper sense of cooperation. We implemented the communication network GA2Net and proposed a new environment that can be use to benchmark future algorithms.

Chapter 1

Introduction

At Thales, in our team called **Theresis**, we have a strong interest in application such as drone defense where numerous drones need to cooperate in order to solve an objective. Logically, we are looking for all the state-of-the-art method in multi-agent reinforcement learning that can help solving those problems. Although the research is still in its infancy, we believe that the field is very promising and deserve our attention. To this end, we follow the latest research and try to implement our own implementation of recent papers. In order to test and compare those different techniques from the MARL theory, we use our own simulator to create environments that match the kind of problems we are face with. Those are often with continuous action space, that involve multiple agents with a sparse reward. We detail below several key points to understand the MARL theory, as well as some related researches that inspired our work and motivated us in our approach.

1.1 Reinforcement Learning

1.1.1 Main concept

We define an agent as an entity that is acting in an environment. At each timestep, the agent is making an action $a \in A$ to switch from one state $s \in S$ to the next s'. Which state the agent will arrive in is decided by transition probabilities between states (P). Once an action is taken, the environment delivers a reward ($r \in R$) as feedback.

In this internship, we place ourselves in the domain of model free RL, meaning we don't know the transition between states and we are not trying to learn a model of the environment.

The agent's policy $\pi(s) = P[a|s]$ provides guidelines on what is the optimal action to take. Its goal is to always maximise his total rewards. Each state is associated with a value function $V(s)$ predicting the expected amount of future rewards we are able to receive in this state by acting the corresponding policy.

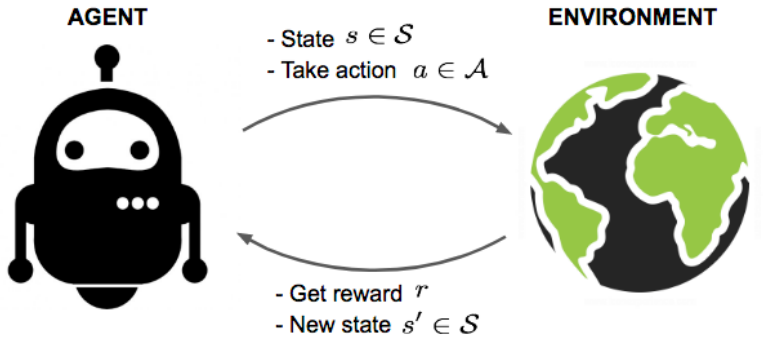

Figure 1.1: Illustration of an agent in a RL environment

We can distinct two types of reinforcement learning algorithm :

1. **On-policy** : Use the deterministic outcomes or samples from the target policy to train the algorithm.

2. **Off-policy** : Training on a distribution of transitions or episodes produced by a different behavior policy rather than that produced by the target policy.

The policy can be either stochastic ($\pi(a|s) = \mathcal{P}_\pi[A = a|S = s]$) or deterministic $\pi(s) = a$. The value function measures how rewarding is a state by the prediction of the future reward. This future reward starting from timestep t can be defined as :

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1.1}$$

where $\gamma \in [0, 1]$ is the discounting factor. We can express the expected return of state s at time t. This is the **state-value**:

$$V_\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right] \tag{1.2}$$

From this equation, we can also define the **action-value** of a state-action pair as:

$$Q_\pi(s, a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] \tag{1.3}$$

Additionally, since we follow the target policy π, we can make use of the probability distribution over possible actions and the Q-values to recover the state-value:

$$V_\pi(s) = \sum_{a \in \mathcal{A}} Q_\pi(s, a) \pi(a \mid s) \tag{1.4}$$

To *solve* an environment, an agent must find the optimal value function that produces the maximum total reward.

$$\pi_* = \arg\max_\pi V_\pi(s), \pi_* = \arg\max_\pi Q_\pi(s, a) \tag{1.5}$$

1.1.2 Markov Decision Process

We can described almost any reinforcement environment as **Markov Decision Process** (MDP). One of the key principal of a MDP is that any future state is independent from of the past given the present state. Mathematically, this is expressed by:

$$P \left[S_{t+1} \mid S_t \right] = P \left[S_{++1} \mid s_1, \ldots\ldots, s_t \right] \tag{1.6}$$

It means that the probability between any transition doesn't depend from what happened before. A MDP is composed of a set of 5 elements $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$

1.1.3 Bellman Equation

Bellman Equation helps us to find optimal policies and value function. We know that our policy changes with experience so we will have different value function according to different policies. Optimal value function is one which gives maximum value compared to all other value functions.

$$\begin{aligned} Q(s, a) &= \mathbb{E} \left[R_{t+1} + \gamma V \left(S_{t+1} \right) \mid S_t = s, A_t = a \right] \\ &= \mathbb{E} \left[R_{t+1} + \gamma \mathbb{E}_{a \sim \pi} Q \left(S_{t+1}, a \right) \mid S_t = s, A_t = a \right] \end{aligned} \tag{1.7}$$

Using those equations, we can update our Q-function iteratively :

$$Q_\pi(s, a) = R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P^a_{ss'} \sum_{a' \in \mathcal{A}} \pi \left(a' \mid s' \right) Q_\pi \left(s', a' \right) \tag{1.8}$$

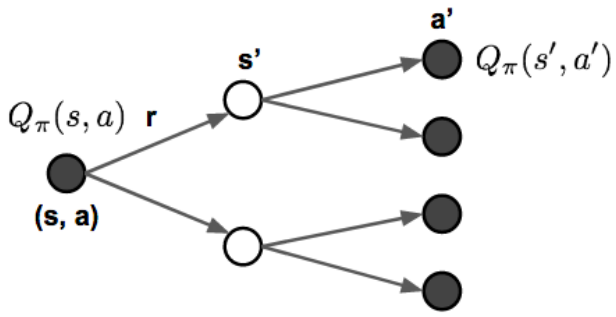

Figure 1.2: Illustration of how Bellman equations update state-value and action-value functions.

1.2 Multi Agent Reinforcement Learning

1.2.1 Notation

We define as multi-agent a system where more than one agents are evolving in an environment. Agents are, partially or not, autonomous. MARL can be divided into three groups : *fully cooperative, fully competitive*, and a *mix of the two*. In our work, we mainly focus on the first group. A fully cooperative multi-agent task can be described as a Decentralized Partially Observable Markov Decision Process, or **Dec-POMDP**. At every timestep, each agent $a \in \mathcal{A} \equiv \{1, ..., n\}$ has a partial observation $o \in \mathcal{O}$ of the true state $s \in \mathcal{S}$. It chooses an action $u^a \in \mathcal{U}$, forming a joint action $\mathbf{u} \in \mathcal{U} \equiv U^n$. This causes a transition in the environment according to the state transition function $P(s'|s, \mathbf{u}) : S \times \mathbf{U} \times S \to [0, 1]$. Because we are in the case of a fully cooperative environment, all agents share the same reward function $r(s, \mathbf{u}) : S \times \mathbf{U} \to \mathbb{R}$ and $\gamma \in [0, 1)$ is a discount factor. Each agent has an action-observation history $\tau^a \in T$, on which it conditions a stochastic policy $\pi^a(u^a, \tau^a)$. The joint policy of all agents π has a joint *action-value function* $Q^\pi(s_t, \mathbf{u}_t) = \mathbb{E}_{s_{t+1:\infty}, \mathbf{u}_{t+1:\infty}}[R_t|s_t, \mathbf{u}_t]$, where $R_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$ is the discounted return.

The *decentralized* part of Dec-POMDP means that each agent only has access to his own history of action-observation τ^a, but the training can be centralised, meaning we can use the global state s as well as the joint history of action-observation τ.

1.2.2 Related Work

Multi-agent reinforcement learning has starting to get more attention after 2015 with the use of deep neural network to approximate the Q-value function [1].Then, the first natural way of dealing with a MARL system in a decentralized way is to have each agent act independently and to compute their own Q-values according to their observations. This is called Independent Q-Learning (IQL) [2] and it is a surprising strong benchmark. It is surprising because the algorithm has no way of dealing with the **non-stationarity** of the environment due to other agents changing behavior while learning. This is a common difficulty to all MARL algorithms. This method is however prone to instability, may not converge and doesn't use extra state information such as the global state s.

If we decentralize the execution, we can however centralise the training. This is what the Value Decomposition Network [3] method does : it learns a fully-centralized Q-value function Q_{tot} which is the sum of the individual value function Q_a. By using this representation of Q_{tot}, a decentralised policy arises simply from each agent selecting actions greedily with respect to its Q_a.

A more recent and successful approach propose to release the constraints on Q_{tot} from the VDN algorithm. Call QMIX, they make the postulate that we don't need the full factorisation to extract decentralized policies. Instead, we need to make the $argmax$ on Q_{tot} the same as the set of all local $argmax$ on each Q_a. The detailed process is described latter with COMIX, that uses the same mixing network as QMIX.

1.3 Communication Network

1.3.1 Concept

Communication is a fundamental aspect of intelligence, enabling agents to behave as a group, rather than a collection of individuals. It is vital for performing complex tasks in real-world environments where each actor has limited capabilities and/or visibility of the world. In MARL, we suppose that agents are acting in a decentralized way. However, that doesn't not prevent them from exchanging messages during execution. This particular branch is called Network Multi Agent Reinforcement Learning (NMARL), where agent are connected via a communication network for a cooperative control objective. Each agent performs decentralized control based on its local observations and messages from connected neighbors. Network System Control is extensively studied and widely applied. Examples include connected vehicle control, traffic signal control, distributed sensing, and networked storage operation.

1.3.2 Related work

Studies on communication between agents in MARL can be divided into 3 four groups based on their communication methods.

The first group is communication via a protocol using an heuristic or direct information sharing. For example, the agents could share at any time their position to all connected agents. The downside of this technique is that the communication isn't designed for performance optimization, and can be redundant. Agents may not need to send a message at every timestep. This can cause overhead latency and slow down learning.

The second group focused on learned communication protocol. In CommNet [4], agents generate a message via a model and share it to the other agents before acting in the environment. They receive the mean transmission via a communication channel and the information is used alongside the observation for choosing the action. More recently, NeurComm [5], a promising approach, uses differentiable communication as well as a spatial discount factor to stabilize training.

The last group focused on learning the topology of the communication network via the use of *attentions networks*. ATOC [6] learns a probability to allocate communication to an agent via soft communication, while IC3Ne uses hard attention to prune links in the communication network.

Chapter 2

Method

2.1 Extending QMIX to continuous action

2.1.1 Context

In March 2020, researchers from the university of Oxford published a paper that propose to extend the QMIX algorithm to the continuous action space. Called COMIX, they overcome the challenge of continuous action space by using the cross-entropy-method. They show that the factorisation of the Q-function is key to performance, and that their algorithm surpass the previous state-of-the-art algorithm MADDPG. Since most of our application at Thales are using continuous action space, such as drone control, we decided to try this approach by developing our own implementation and test in on our home-made environments.

2.1.2 Cross Entropy Method

Since we are working in continuous action space, agents can't select their best action greedily according the action-value function Q_a since we don't have a finite number of actions. The cross-entropy method (**CEM**) is a sampling-based derivative-free heuristic search method that has been successfully used to find approximate maxima of nonconvex Q-networks in a number of single-agent robotic control tasks [7].

At each timestep t of our environment, each agent $i \in Z$ observe its environment and retrieve an observation o_i. CEM draws iterativly a batch of N random samples for a normal distribution $\mathcal{D}_k$ of mean $\mu = 0$ and variance $\sigma = 1$ at each iteration k. We compute the action-value for each of this samples according to the observation $Q_i(o_i, a)$ where $a \in N$. We select the top best $M < N$ samples are used to fit a new Gaussian distribution D_{k+1} and repeat this process K times. This

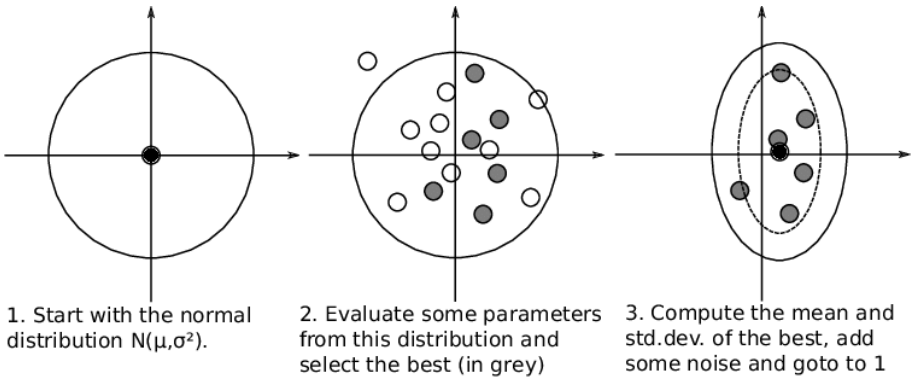

Figure 2.1: Illustration of the cross entropy method

illustration above illustrate this process. The only difference is that we don't had noise to the samples during the process. After the K iterations, we take the best sample as the action to do for the agent.

2.1.3 Mixing Network

The mixing network used in COMIX is the same as in QMIX. The idea is to factor the joint action-value function Q_{tot} with the only condition that a global argmax on it yields the same result as a set of individual argmax operations performed on each Q_a:

$$\underset{\mathbf{u}}{\text{argmax}} Q_{\text{tot}}(\tau, \mathbf{u}) = \begin{pmatrix} \text{argmax}_{u^1} Q_1\left(\tau^1, u^1\right) \\ \vdots \\ \text{argmax}_{u^n} Q_n\left(\tau^n, u^n\right) \end{pmatrix} \tag{2.1}$$

This allows each agent a to participate in a decentralised execution solely by choosing greedy actions with respect to its Q_a.

To satisfy equation 2.1, we represent Q_{tot} as a global sum Q-value of each agent. This approach is called CO-VDN, which is like VDN but now working with continuous action space. An other solution that is less constraining for representing Q_{tot} is the one use in QMIX. By using a mixing network that take as input the local Q-values, and output Q_{tot}, we can force the monotocity via a constraint on the relationship between Q_{tot} and each Q_a :

$$\frac{\partial Q_{tot}}{\partial Q_a} \geq 0, \forall a \in A \tag{2.2}$$

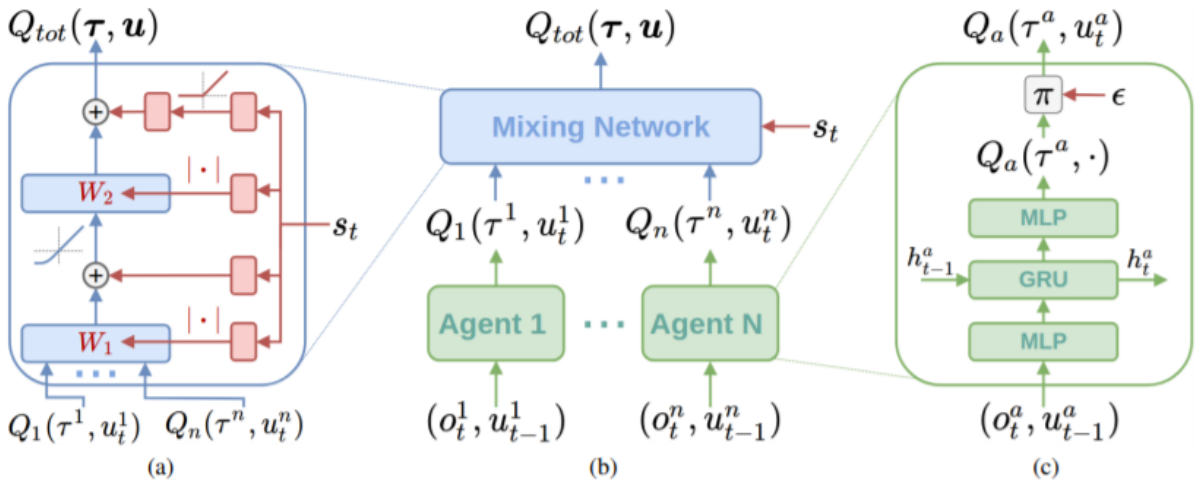

Figure 2.2: Mixing network structure.

To do this, we generate the weights and biases of our mixing network via *hypernetworks* that takes as input the global state s. To satisfy 2.2, we force the weights to be positive. This allows the mixing network to approximate any monotonic function arbitrarily closely. The global architecture can be found in figure 2.2. The reasoning behind this is choice is that, by setting less constraint on how to compute Q_{tot}, we can have a larger diversity of representation of the Q function.

COMIX uses this mixing network in combination with the cross-entropy method to deal with the continuous action space while using the strength of factorizing the Q-function.

2.1.4 Implementation

We create our neural networks and handle the training using Pytorch. To parse the arguments of our experience, and to set values by defaults, we use the python library docopt. All code corresponding to the learning algorithm is put in the module **therlib**, while the neural networks are inside the module **therutils**.
To help speed up the training, we normalize the observation to be in range $[0, 1]$, and eventually scale the action depending of the environment.
To monitor our experiences and compare different algorithms, we use Tensorflow with Tensorboard and look mainly at the reward per episode, the mean Q-value chosen for the action, as well as the loss.

The cross entropy method is implemented in a distributed fashion with pytorch to reduce computational time.

2.1.5 Actor-Critic vs Deep Q Network

To test our implementation of COMIX, we compared it against **MADDPG**, an Actor-Critic method, which is the extension of the mono-agent algorithm DDPG. It is composed of two main components : an policy network $\pi_i(o_i)$ of parameters θ_i that directly outputs the action of an agent (the Actor), and a centralised action-value network $Q_i^{\pi}(x, a_1, \ldots, a_N)$ (the Critic). Using the Q-function, we can update the parameters of the policy network of agent i by computing the gradient of the expected return :

$$\nabla_{\theta_i} J(\theta_i) = \mathbb{E}_{s\sim p^{\mu}, a_i\sim\pi_i} \left[\nabla_{\theta_i} \log \boldsymbol{\pi}_i (a_i \mid o_i) Q_i^{\pi} (\mathbf{x}, a_1, \ldots, a_N)\right] \tag{2.3}$$

The critic is here to judge how well the current actor is performing and then used to improve the policy. We only use the policy network during execution.

2.2 Improving learning via Attention Communication Network

2.2.1 Motivation

After implementing successfully COMIX, we looked at the next challenge we wanted to deal with in MARL. One paper called REFIL came out in the same time and proposed a new technique that uses attention and imagination to solve the issue of **dynamic scenarios** in MARL.
Dynamic scenarios is a special case of MARL where the number of entities in the environment varies during training. The main difficulty is that the size of the local observation and global state change with the number of entities, and that a strategy may not work for all types of configuration. Agents need to adapt their cooperation and get a sense of abstraction. To this end, researchers from the university of Oxford proposed a method called **REFIL** [8], inspired by QMIX, that uses attention and entity-wise factorization of the Q-function to learn across different scenarios. The paper demonstrate the pertinence of their method by testing it on scenarios in the SMAC environment.

In the same time, we discovered a innovative technique called Shared Modular Policies [9] that can also learn a policy that generalize across different scenarios. Suited for robotic control, this method propose to see a morphology as a set of limbs that have their own observation and can act on their own. By using a communication channel, the limbs were able to coordinate their movements and make the whole morphology to move in different simulations of **Mujoco**. We got interested by this approach since it resembles the formulation of a multi-agent system and was able to deal with a varying number of limbs and still generalize well.

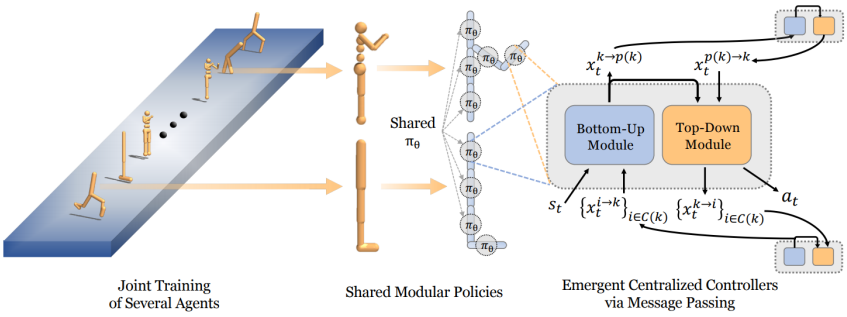

Figure 2.3: Shared Modular Policies utilize communication to learn one policy for different morphologies

However, Shared Modular Policies is an algorithm that was built for robotic control, making it hardly configurable for our environments. Indeed, a robotic morphology such as the ones found

in Mujoco have the advantage of having a static number of limbs. This means that there is a fixed number of modules/agents, making the topology of the communication network fixed during an episode. When using environment such a Starcraft, agents may die or getting out of range of communication, making the communication network dynamic because it needs to adapt to what is happening in the environment.

2.2.2 Using attention for communication

From this idea of using a communication network that could change its topology depending of the situation, we had two options :

1. Communication network within the agent's neighborhood

2. Learn the communication network's topology

The first idea is the simplest to implement since it we only have to use the information of distance between agents to decide if whether or not they can communicate. However, it does not generalize well to all environments. For example, we could have drones that are always far distant from each other but still need to send messages to coordinate their actions. Furthermore, it supposes that agents that are close should communicate, which may not be always the case.
Therefore, we explored the second idea and found that a recent paper had implemented such a network. Called **GA2Net** [10], it uses both soft and hard attention to model the relationship between agents. The use of attention to learn the relation between entities reminded us of the paper REFIL that was used for learning in dynamic scenarios, which is one of our objective.

Attention is key concept that was recently introduced in deep learning, originally in the NLP literature. The goal of this mechanism is to manage and quantify the interdependence between variables. It is use to distinct what is important from what is less. Suppose we have a sequence of input x of length n, and a previous output y of size m. We encode the input into a set of key-values pairs $(\mathbf{K}, \mathbf{V})$, both of dimension n, and y into a query $\mathbf{Q}$ of dimension m. Using those sets, we can compute the scaled dot-product attention :

$$Attention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = softmax(\frac{\mathbf{Q}, \mathbf{K}^T}{\sqrt{n}})\mathbf{V} \tag{2.4}$$

In GA2Net, when computing the attention agent i pay to each other agents $j \neq i$, we compute the key $\mathbf{K} = e_j^T W_k^T$, the query $\mathbf{Q} = W_q e_i$ and the value $\mathbf{V} = W_v e_i$, where e_j and e_i are respectively the embedded observation of agent j and i. Furthermore, we distinct two types of attentions : the soft and the hard.

Soft attention calculates a importance distribution of elements with the softmax function. Given a sequence of inputs $\mathbf{x}$ of size m, we compute a vector of weights $\mathbf{w} \in [0, 1]^m$ to assign to it.

$$w_k = \frac{\exp\left(f\left(T, e_k\right)\right)}{\sum_{i=1}^{K} \exp\left(f\left(T, e_i\right)\right)} \tag{2.5}$$

where e_k is the feature vector of agent k, T is the current agent feature vector, and w_k is the importance weight for agent k.

Hard attention selects a subset from input elements, which force a model to focus solely on the important elements, entirely discarding the others. However, hard attention mechanism is used to select elements based on sampling and thus is non-differentiable. Therefore, it cannot learn the attention weight directly through end-to-end back-propagation.

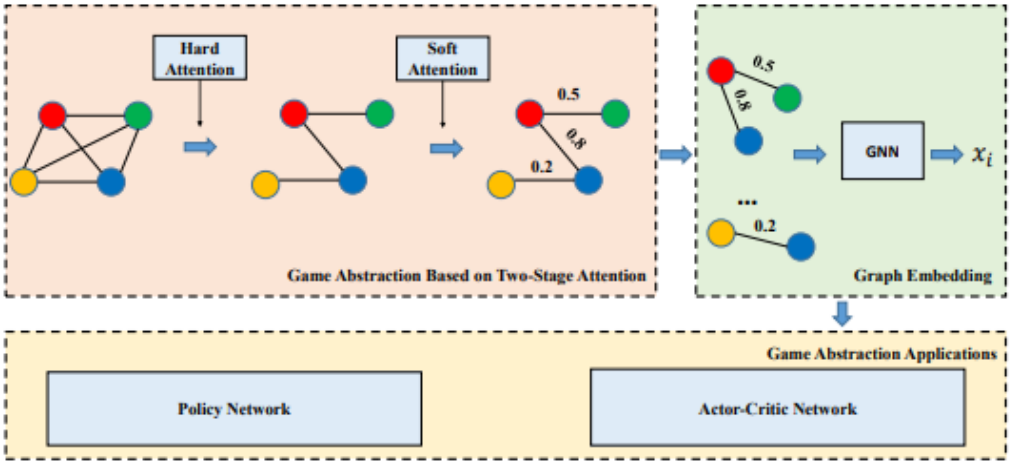

Figure 2.4: How GA2Net uses hard and soft attention

We model our graph of communication as $G = (N, E)$, where N are the nodes which represent the agents, and E are the edges that model the relationship between them. Each agent i has his own representation of the graph : the sub-graph G_i, where agent i is only connected to the agents it need to interact with, and the weight on the edge represent the importance of this relation.

To compute those relations, we use first a LSTM network to encode the vector of local observations **o** into an embedding vector **h**

$$h_i = LSTM(o_i^t, h_i^{t+1}) \tag{2.6}$$

We can compute the hard attention mechanism show in 2.4 by using a bi-LSTM. We need a bi-LSTM because the relationship weight between agent i and agent j also depends on the information of all other agents k in the environment, where agent $k \in 1, ..., n$ and agent k is not in i, j. In order to be able to compute back-propagation, we use the gumbel-softmax function.

$$W_h^{i,j} = \text{gum}\left(f\left(\text{LSTM}\left(h_i, h_j\right)\right)\right) \tag{2.7}$$

where $W_h^{i,j}$ is either 0 or 1.
Once the step of hard attention is done, we can compute the soft attention which is the value associated to each edge. For agent i and j, we compare their associated embedding e_i and e_j using a query-key system and pass the matching value between into a soft-max function.

$$W_s^{i,j} \propto \exp\left(e_j^T W_k^T W_q e_i W_h^{i,j}\right) \tag{2.8}$$

where $W_s^{i,j}$ is the value associated to the edge between agent j and agent i. W_k transforms e_j into a key, W_q transform e_i into a query and $W_s^{i,j}$ is the hard-attention value.

Finally, we can get for each agent the mean weighted message received, using the attention weights $W_s^{i,j}$ computed earlier.

$$x_i = \sum_{j \neq i} w_j h_j = \sum_{j \neq i} W_h^{i,j} W_s^{i,j} h_j \tag{2.9}$$

The intuition behind this is that an agent will pay more attention to the message of a friendly agent if its action depends on it, and should dismiss the communication of other agents that are *irrelevant*.
Once the mean message is received, the agents can compute choose their action following their policy π, that we can learn using either an Actor-Critic framework or any other MARL algorithms like QMIX.

2.2.3 Implementing GA2Net

To implement GA2Net, we refactored our previous code of the algorithms MADDPG, COMIX and QMIX, so we can interspersed a communication network between the observation and the choice

of an action. We had the option to change easily the variation of GA2Net we want to use when launching an experiment. Those variations are ablations of GA2Net we use for evaluation :

1. **GA2Net** with both hard and soft attention
2. **GA2Net** with only soft attention
3. **GA2Net** with no hard and soft, which is equivalent to the algorithm ComNet [4].

For both COMIX and QMIX, we modified the Q function network so it now takes as input the output of the GA2Net communication network $\mathbf{x_i}$:

$$Q_i(o_i, a_i) = f_i(g_i(o_i, a_i), x_i) \tag{2.10}$$

For the neural network of GA2Net, we modified an existing implementation from Github (https://github.com/starry-sky6688/StarCraft).

Chapter 3

Environments

In this chapter, we will develop different environments that were used during this internship. We will explain what challenges they offers to the MARL research and how we tried to overcome them. Finally, we will go in more details about the environment we developed specifically for this internship.

3.1 Multiagent Particles

Created by OpenAI, it is often used as a toy problem due to its simplicity and its speed of computation. There are multiple scenarios that are easily changeable, with both competitive and cooperative tasks. Developed originally for testing the Multi Agent Deep Deterministic Gradient Policy (**MADDPG**) [11], it is now a common benchmark for MARL algorithm that works in continuous action space. Here is some example of scenarios you can find in the multiagent particle environment:

- **Simple-Push (keep-away)** : 1 agent, 1 adversary, 1 landmark. Agent is rewarded based on distance to landmark. Adversary is rewarded if it is close to the landmark, and if the agent is far from the landmark. So the adversary learns to push agent away from the landmark.
- **Simple Reference** : 2 agents, 3 landmarks of different colors. Each agent wants to get to their target landmark, which is known only by other agent. Reward is collective. So agents have to learn to communicate the goal of the other agent, and navigate to their landmark.
- **Simple tag (Predator-prey)** : Good agents (green) are faster and want to avoid being hit by adversaries (red). Adversaries are slower and want to hit good agents. Obstacles (large black circles) block the way.

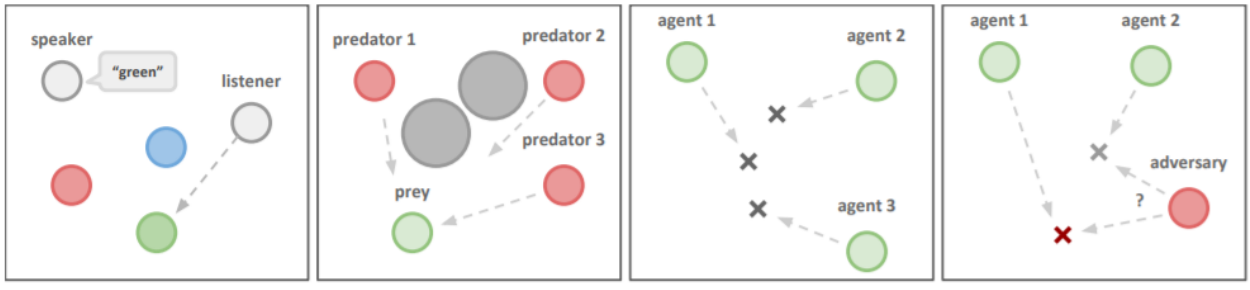

Figure 3.1: Example of scenarios

3.2 Starcraft Multi-Agent Challenge

After beating the world best player in Go [12] in 2016, DeepMind was looking for a new challenge. They focused on the real-time strategy game Starcraft II, where players must use limited

information to make dynamic and difficult decisions that have ramifications on multiple levels and timescales. In 2019, they succeeded in this task as well, with their AI AlphaStar[13]. To do this, DeepMind worked closely with the game editor Blizzard to develop an open-source API that allow to play the game with AI coded in Python.

This same API is used in 2019 to developed a new environment, designed for multi-agent reinforcement learning. Call Starcraft Multi-Agent Challenge, or **SMAC**, it offers a set of scenarios where a group of units must defeat another group, controlled by the game's heuristic AI. Those challenges are partially observable, cooperative, in which teams of agents must learn to coordinate their behaviour while conditioning only on their private observations. This environment's goals is to help MARL algorithm move beyond toy domain, by proposing a set of scenarios anyone can use or modify.

Each unit in SMAC is independent and is controlled by a single agent. The agent can only observe the allies and enemies inside his radius of vision, and has access to the amount of health and shield they have. This means that an agent can not determine if an ally is alive or dead if it is outside of its vision.

The main goal is to maximise the win rate for each battle scenario. A continuous reward is computed at each timestep, which is based on the hit-point damage dealt and enemy units killed, together with a special bonus for winning the battle.

The action-space is discrete, and consists of actions such as moving in a direction, attacking a specific enemy unit, or do nothing. Agents can only attack unit that are in his *shooting-range*, which is slightly lower than is *vision range*.

Figure 3.2: Example of SMAC challenges

3.3 Swarm Defense

This environment was created by my colleagues using the simulation engine *SE-Star*, developed internally at Theresis. Coded in C++, this 3D engine was originally used for simulating large crowd behavior, and is now use for drone's environment.

Using a Python API, we are able to create the environment using Python's code and easily modify the inner parameters. In this simulation, n drones evolve in a 2D space of size 5000x5000 and have to prevent m enemy drones from reaching the center red area. Enemy drones are controlled by the environment. Agents can see allies position as well as enemies position relative to themselves. The action space is continuous and is composed of the x and y coordinates of the targeted location. Every drone have a small cone of vision in front of them that destroy enemy when they are inside. To stop the attack of the opponent, agents need to get sufficiently close and face the enemy drone directly. This implies a lot of precision over the continuous control, as well as coordination with allies when there are numerous enemies. This environment can be either used in 2D or in 3D.

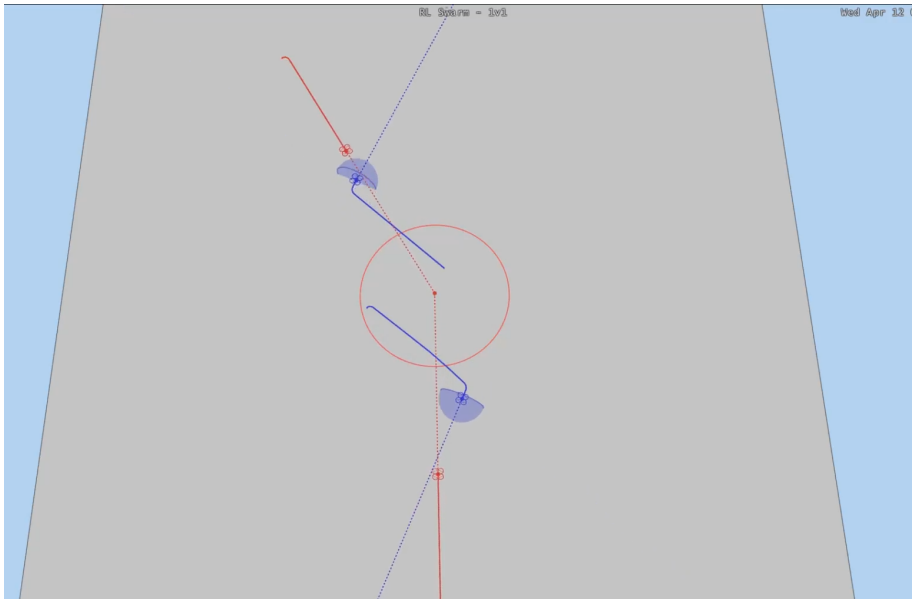

Figure 3.3: Swarm Defense environment with 2 Attackers VS 2 Defenders

3.4 Detector

3.4.1 Context

We developed earlier how we implemented a communication network to see how it can improve training and help the agents discover better strategies. The next step was to test this method on already existing Thales environment built with the simulation engine *SE-Star*. However, as we detailed on the section 4, results weren't as good as expected regarding on how the agents used the communication. We suspect that this is due to an environment where communication isn't that much needed to succeed. The extra communication can even make the training worst, as we show in our section on experiments.

Therefore, we decided to create an new environment where communication could be **key** to good performance while being a pertinent proof of concept for future Thales application.

3.4.2 Goal

In this environment, there are 2 kind of entities :

1. **The Detectors** are drones with a radar to scan the ground. They are the agents controlled by the RL algorithm.

2. **The targets** are objects of interest, enemy forces, that stay on the ground and can not move.

The map is a square of length $L = 5000$. Detector drones receive in their observation the center position of n **targets area**. The target area is a circle of radius $r = 500$ where a target drone will spawn. Detectors are requested to find the targets using the radar they have. The radar scans the ground and reveals the presence of a target. An episode last for $T = 200$ steps, and a request for a target will timeout after $p = 50$ steps.

3.4.3 Action and Observation and Reward

The action is the direction in radians the agent will take and move to at a constant speed.

To make the communication network useful, we do not allow agent to see each other. The can observe their own position and orientation in radians, as well as the position of each target area center t_x^i and t_y^i relative to the agent. In addition, we gave the time left before the target timeouts, divided by p.

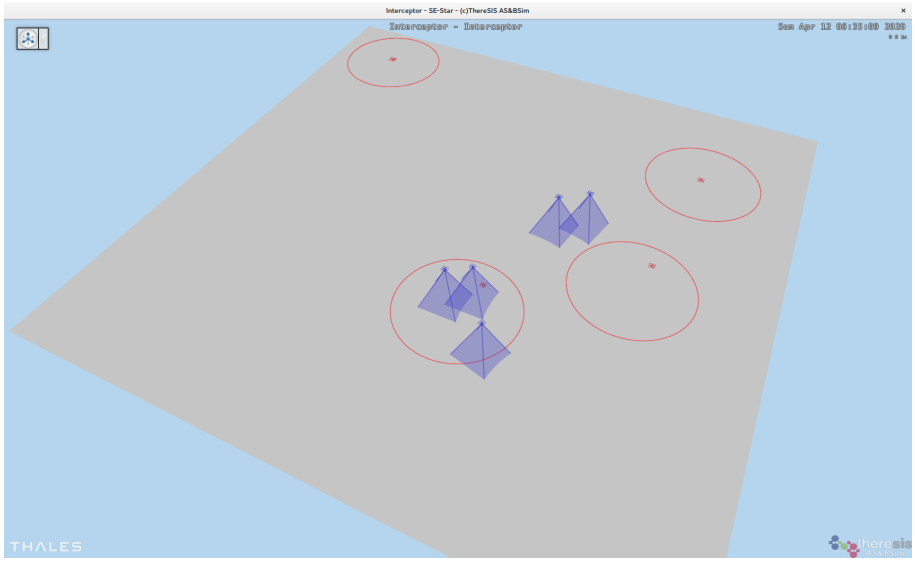

Figure 3.4: Detector environment

The reward is shared between all agents. We penalize them when one is leaving the main area by giving a penalty of $r = -0.1$. When a target is found, we give a reward of $r = 1$ and we create a new request to attend to. When the request's time is at 0, we reset the request with a new target area and give a reward of $r = -0.5$

3.4.4 Challenges

This environment presents several challenging difficulties. The first one is the **allocation of resources** to deal with the issue of **multi-task objective**. Since they are multiple targets which have a limited amount of time before disappearing, agents must coordinate their actions. If many the agents go to the same target, this may be an inefficient behavior and may result in a low reward. What we want to see is agents efficiently coordinating with each other to avoid missing a request. This may lead to behavior such as sending more agents to a far away request in order to search faster the target drone inside it, while only sending one agent to a closest request where time is more than enough for one detector drone to search the area. The last challenge is the **communication**. Since agents doesn't see each other, they must send messages to coordinate their actions. What could be interesting is to see how agents pay attention to each other during an episode using the attention network. Intuitively, we suppose agent should be communicating more with agents close to the target area and less with far away agents.

Chapter 4

Results

In this last chapter, we will go in details on how did we conduct our experiments, what was the results and what conclusion do we draw from this.

4.1 Training configuration

Training a reinforcement learning algorithm can be costly, both in time and in computational power. Finding the right hyperparameters can be time consuming due most RL methods being very sensible to the configuration of these values. It is then essential to be able to launch quickly experiments without them slowing down our work while running. To this end, we have at our disposal a server called *DL6* with 48 CPUs and 7 GPUs. Most of our experiences described below were training using this, with CUDA to speed up the training. We also use the library *multiprocessing* of Python to launch multiple instances of the environment and distribute it along the CPUs. Additionally, to run experiments on the environment Starcraft, we were constrained to use **Docker** due to the incompatibility of our OS with the API made by Blizzard.

4.2 Testing COMIX on a toy environment

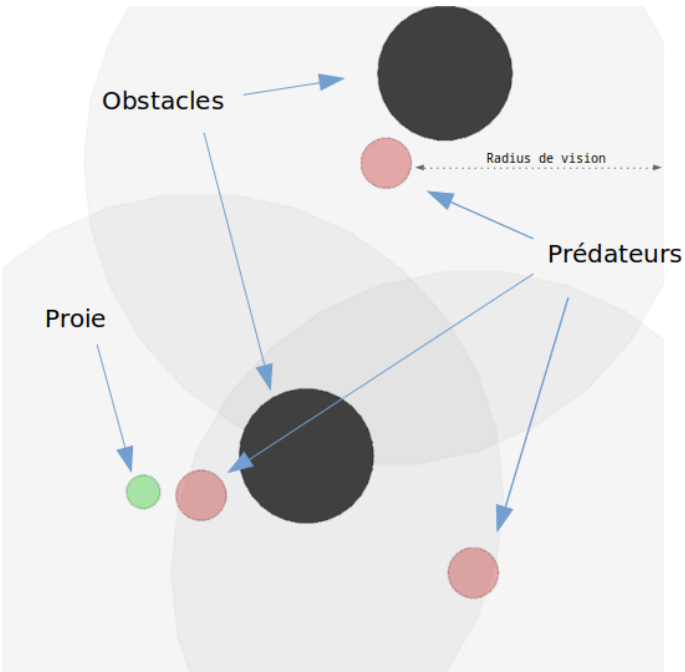

Figure 4.1: Predator Prey environment modified

We implemented a new algorithm for continuous action space environment in multi agent system called COMIX, and explained it in section 2.1. We tested it against the other implementation we have and compared it against MADDPG. We used the environment *MultiParticle* by OpenAI, and more precisely, the scenario of the prey-predator, where n number of agents need to chase a single prey.

We used a modified version of this environment where we added **partial observation** for the predators. Each predator have a circle of vision of radius r around them and can only observe the other predators and the prey if they are inside their vision range. This makes the task of chasing the prey more challenging.

Moreover, we hard-coded the behavior for the prey since we are only interested in the learning of the predators. We ran multiple experiments on 20 000 episodes using a learning rate $\alpha_a = 0.01$ for the actor and $\alpha_c = 0.001$ for the critic network of MADDPG. We chose $\gamma = 0.85$. For COMIX, we set $\alpha = 0.01$ for the Q network, the discount factor is $\gamma = 0.95$. The learning phase happen at the end of each episode, with a learning batch of size $m = 1024$. To help the agent explore, we added a constant Gaussian noise to the action space with $\sigma = 0.1$

4.2.1 Full vision

We first run an experiment with full vision and look at the number of time the predators hit the prey in the evaluation's runs.

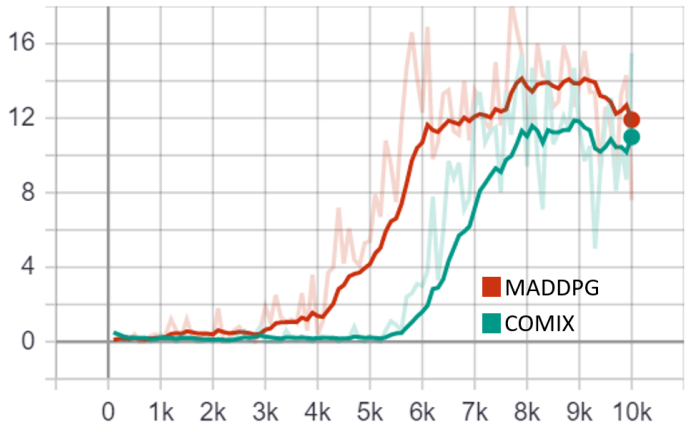

Figure 4.2: Number of collisions with full vision

According to the result in figure 4.2, MADDPG performed slightly better than COMIX when full vision is enabled. However, after 10 000 episodes, the mean number of collisions with prey is about the same, with a mean reward of $R = 118$ for MADDPG and $R = 110$ for COMIX. We however note that the actor-critic algorithm looked more sensible to the choice of hyperparameters than COMIX.

In terms of computational speed, COMIX is slightly slower due to the cross entropy method being computationally intensive. This may be an issue when dealing with many agents.

4.2.2 Partial vision

We now set the radius of vision for the agents to $r = 0.9$ as illustrated in figure 4.1 and retrain both algorithms using the same settings.

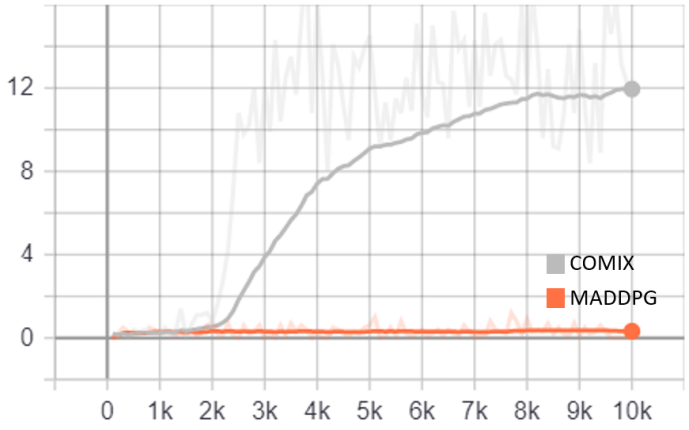

Figure 4.3: Number of collisions with small vision range

Surprisingly, COMIX was more robust to the limitation of vision than MADDPG. This was confirmed using different settings of hyperparameters, and with multiple instances of training. None of them succeeded to converge when vision was limited using MADDPG.

While we believe with more time and effort, it would be possible to find the right set of hyperparameters for MADDPG to make it converge to the optimal policy, this shows how COMIX seemed to be better choice when the vision of the environment is limited.

4.3 Swarm Defense

We test the robustness of the COMIX algorithm when applied to a more complex environment. Using two scenarios (2 defenders vs 2 attacks, 4 defenders vs 4 attackers), we compare our different implementations.

We train during x episodes and we freeze training every 100 episodes to evaluate. The model is saved every 1000 episodes. We choose $\gamma = 0.99$ for all algorithms, and train the model at the each of each episode with a learning batch of size $m = 1024$. We update the target model at each training with a soft update of the weights setting the parameter $\tau = 0.01$, with a learning rate of $\alpha = 0.01$.

After numerous attempts, we had issue stabilizing the training of COMIX or MADDPG on both scenarios. The reward collapsed every time after improving a bit, with no recovery. We tried many different settings for the hyper parameters, with no real improvements. Surprisingly, removing the mixer from COMIX resulted in much better result in this environment, with a much more stable training.

In figure 4.4, we compared our version of CO-VDN with CO-VDN with the communication network GA2Net. The goal was to observe if adding communication in an environment with full-vision could improve performance. Results showed however that this doesn't improve nor decreased the mean reward.

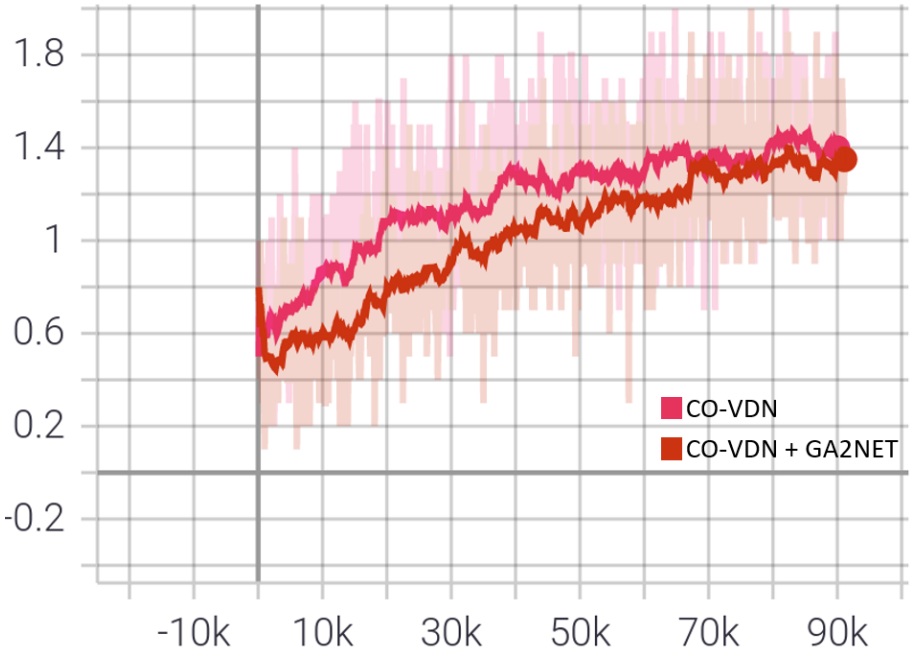

Figure 4.4: Mean number of enemy destroyed with scenario 4 attackers VS 4 defenders

4.4 Adding communication on Starcraft

In this section, we test the implementation of the communication algorithm GA2Net, combined with the algorithm VDN. We believe we can improve the mean win rate in some scenario by adding a communication process between the agent. With those experiences, we also want to see if adding communication can also result in overhead latency and slow down the training instead of improving it. The results of those experiences can give insights on whether or not we should consider implementing communication between agents in future use-cases.

In this environment, we use for all experiments a configuration of 10 000 training episodes, and do an evaluation every 1 000 episodes. We use a learning batch that contain the execution of 32 episodes, with $\gamma = 0.99$, and a batch replay of size $n = 5000$ samples. We set the learning α to 0.00005. The hidden layers are composed of 64 units, while the attention layers for the GA2Net is 32. We update the target network every 100 episodes. For the exploration, we add a decreasing probability of $0.05 < \epsilon < 1.$ to take a random action.

4.4.1 So Many Banelings

In this scenario of the SMAC environments, 7 zealots need to survive to 32 banelings. To do this, they must learn *to split* (getting away from each other) to avoid the banelings exploding and dealing splash damage.

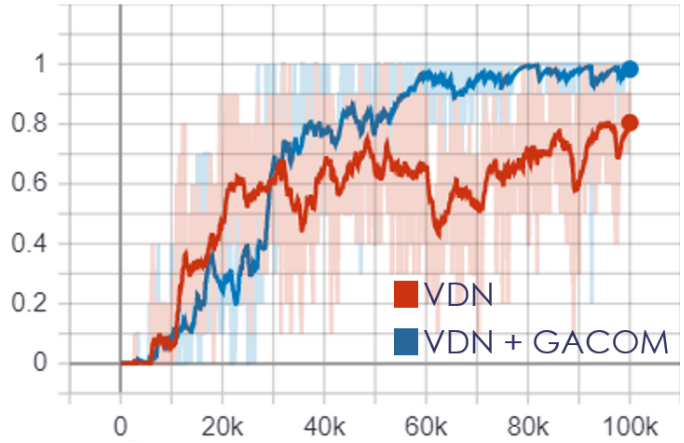

Figure 4.5: Win rate for VDN with and without communication on scenario *so many banelings*

The mean win rate of the agents when communication is enabled is around 0.98 while it is 0.80 when it is not. Moreover, the training is more stable when agents exchange messages.

We visualized the behavior of the agents, and observed their behavior with and without a communication network.

1. **Without communicating**, agents are staying close to each other, and are waiting for the pack of banelings to getting close before sending one unit towards them. This reduce efficiently the zone damage of the banelings but it is a risky tactic. If the baneling explode too close of the group, many agents may get hit. Coordination must be perfect to avoid damage.

2. **With communication**, agents are agreeing from the start to spread across the map. By doing so, they make sure the explosion of banelings will only hit one agent at a time, minimizing completely the total damage received. This strategy is more optimal than the previous one, which result in a higher win rate.

We investigate if the agents did use the attention network to get a sense of abstraction. To see how they relate to other agents, we look at the soft and hard weights $w_h^{i,j}$, $w_s^{i,j}$ that are output by GA2Net. Unfortunately, during all episode, at each timestep, all hard weights were equal to 1 and all soft weights were equals to each other. This means an agent paid equal attention to all other agents, and did not prioritize any communication. To make sure that attention network didn't play a role in the result, we train again our agents on the same scenario, but using this time ComNet, which is equivalent to GA2Net but without hard and soft attentions enabled. The result were similar to the one we had using GA2Net, with a faster training speed since we remove many layers in our architecture. This confirms that attention wasn't used effectively.

We hypothesize that agents did not use the attention because, due to how this scenario is build, it wasn't useful. We thought at first that agents would focused on the agent that get close to the enemy, but there is actually no specific reason to think that would help the training. To confirm this hypothesis and to make sure there is no error in our code, we decided to try a different scenario where paying attention to a specific group should be more rewarding.

4.4.2 Corridor

In this scenario, 6 zealots need to fight against a group of 24 zerglings on a map with a bottleneck on the middle. This task is much harder than the previous one for multiple reasons. The first reason is that, contrarily to the baneling's task, agents must choose specifically which unit to attack, while with baneling, it just had to move towards it for it to explode. Moreover, if an agent is surrounded and tries to attack a zergling which is close, but out of reach, it will fail and only receive damages. Finally, agents should coordinate their movements perfectly to avoid luring to many zerglings and get surrounded.

We trained VDN and VDN+GA2Net and looked at the result. None of the algorithm succeeded to solve the environment, which is no surprise since researchers from Oxford didn't succeeded themselves using QMIX. However, agents did improve their rewards during training with both algorithms. When we stopped the training at the end of 100 000 episodes, the rewards was still increasing. We believe that with more training time, agents may start solving this scenario.
We plot the mean attention given among agents and see if it correlates with what is happening on the map.

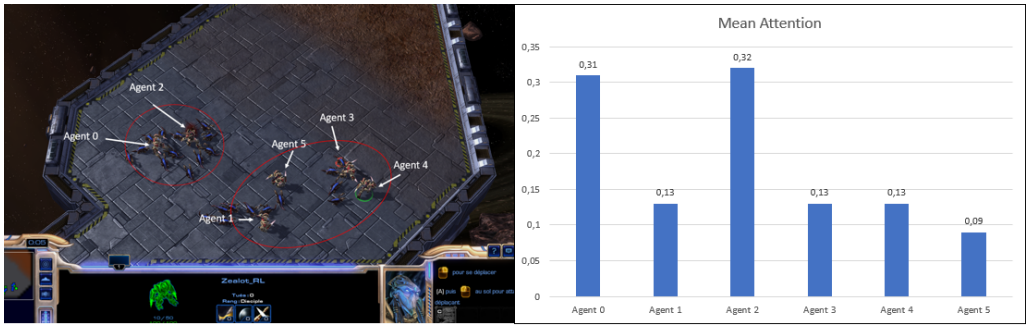

Figure 4.6: Attention given to agents on scenario *corridor*

What you see in figure 4.6 is a screenshot of an episode after training the GA2Net algorithm. Two of the agents are attracting most of the enemies, while the four other are dealing with a smaller group of units. We retrieve the weights of the communication network produced by the soft attention W_s, and we compute the average for each agent : $w_i = \frac{1}{N} \sum_{j \neq i}^{j} w_s^{i,j}$.

Interestingly, we see that our agents **tends to pay more attention to the group that is fighting the most number of zerglings.** Indeed, this fight is key for the success of the scenario. This show that agents learnt how to use the attention to prioritize critical communication from agents.

On the downside, when comparing the training curves of GA2Net+VDN with VDN, we observe that the reward is increasing slower when using communication. This is an issue, because using GA2Net has also for effect to increase training time (approximately 1.5x slower). This is probably due to the fact that the communication is creating overhead latency which makes the task of learning the optimal action harder. Looking at how agents are using the hard attention W_h, we observe that they weights are always equals to 1, meaning all agents always speak to all agents. This is not the behavior we are expecting, and this may be the cause of the latency.

Chapter 5

Further Improvements

5.1 Experimenting on Detector

Unfortunately, when we are writing these lines, we are still one month away from the end of the internship and we weren't able to conduct all the experiments we wanted to present in this report. We found that the first results we got in the environment we built, Detector, are promising. We hope to prove that the implementation of communication between agents can help the training in environments where information is key.

5.2 Testing noisy communication

Scaling reinforcement learning algorithm from a simulation to the real world is an hard challenge. There is often a huge gap between the two and most RL algorithms fail to adapt. When we implemented GA2Net, we naively stated that there is always perfect communication. This is however not reasonable if we want to adapt our agents to work in a real environment. For example, drones that would evolve in a urban environment might get behind a building where no communication can go through. How do we make sure the agent will still be able to coordinate with its allies if it is not able to send or receive messages ?

Setting an environment where communication could randomly be cut between agents would be an interesting approach to how robust those algorithms are. One could also imagine adding random noise during execution to the message sent to see how the behavior of the agents differs.

5.3 Implementing other communication networks

As for now, the approach we tried with GA2Net showed mixed result. Still, we are now able to understand better how communication could impact training, either beneficially or disadvantageously. We think using attention with communication may not be useful in most scenarios, but other communication algorithms may be. We mentioned in the introduction the algorithm NeurComm [5] which recently came out and was presented in ICLR 2020. It introduces a new protocol of communication, as well as a **discount spacial factor** for the reward in addition with the temporal one. According to the paper, this helped stabilizing MARL algorithms both with and without communication. We believe it is a promising approach and can be an pertinent choice for future projects.

Chapter 6

Personal Assessments

When I started searching for an internship, I had two objectives in mind. The first one was to have an experience closed to the research field. Because I have already worked for 2 years in a consulting company, I felt the need to see a different field, with different methods and different objectives. I was curious on how I would fit in the world of research and if I would find my place there. The second objective was to work in the domain that I am the most curious about, the one that actually made me interested in artificial intelligence : the field of reinforcement learning. My internship at Thales fitted these two objectives, so I am very glad I was able to work there.

As a research intern, I was given almost complete freedom to how I wanted to conduct my work. Completely autonomous, I worked most of the time on my own. It was much harder than I thought to work alone on a project, and the Covid-19 pandemic made it even harder. For the first month and half of my internship, I was working at home 4 days a week. It found it easy to get lost in my research if you are not enough structured and rigorous. However, once faced with the first difficulties, I believe I was able to overcome them and adapt to this change of pace.

Working in the field of reinforcement learning was both stimulating and fascinating. Most of the application are still at the early phase and a lot have still to be done. This is extremely motivating because I felt I was able to give my contribution and help the research moving forward. I really enjoyed working on different environments and experimenting, although lot of time can be wasted. Most experiments take a long time before even converging, so you need to think twice before launching a training.

With this internship, I developed skills on how to assimilate faster a research paper, how to conduct an experiment and construct a scientific approach. I believe I also improved my coding skills in Python, especially in how to keep it structured and cleaned. Those last months was also the chance for me to learn how to use distributed computing to optimize the computation speed. Finally, I have learnt a lot on reinforcement learning theory, and I understand better the different ramifications of this field.

With this experience now almost behind me, I am very grateful to my team at Thales for giving my this opportunity. I hope my work on multi agent reinforcement learning and the use of communication network will be helpful to them. If I am able to, I will try to find a job in RL because I believe this is what I want to do and what I want to work on for the following years. I also understood that I am a team player and I feel better at my place when working on a project with a team.

Bibliography

[1] Ian J. Goodfellow and Yoshua Bengio et al. Generative adversarial networks. June 2014.

[2] Ardi Tampuu and Tambet Matiisen. Multiagent cooperation and competition with deep reinforcement learning. 2015.

[3] Peter Sunehag. Value-decomposition networks for cooperative multi-agent learning. June 2017.

[4] Arthur Szlam Sainbayar Sukhbaatar. Learning multiagent communication with backpropagation. June 2016.

[5] Sandeep Chinchali Sachin Katti Tianshu Chu. Multi-agent reinforcement learning for networked system control. April 2020.

[6] Jiechuan Jiang. Learning attentional communication for multi-agent cooperation. November 2018.

[7] Dmitry Kalashnikov. Qt-opt: Scalable deep reinforcement learning for vision-based robotic manipulation. June 2018.

[8] Christian A. Schroeder de Witt Shariq Iqbal. Randomized entity-wise factorization for multi-agent reinforcement learning. June 2020.

[9] Deepak Pathak Wenlong Huang. One policy to control them all: Shared modular policies for agent-agnostic control. August 2020.

[10] Weixun Wang Yong Liu. Multi-agent game abstraction via graph attention neural network. November 2019.

[11] Ryan Lowe. Multi-agent actor-critic for mixed cooperative-competitive environments. June 2017.

[12] David Silver. Mastering the game of go with deep neural networks and tree search. June 2016.

[13] David Silver Oriol Vinyals. Grandmaster level in starcraft ii using multi-agent reinforcement learning. June 2019.