Présentation de mi-parcours de PFE

Apprentissage par renforcement multi-agents

David Albert

INSA Rouen

14 janvier 2019

1. Aspect théorique
   - Le cadre
   - Prérequis
   - Multi-Agent Acteur-Critique

2. Aspect pratique
   - Apprentissage pour système d'agents mixtes
   - UML

3. Axes d'orientation possibles

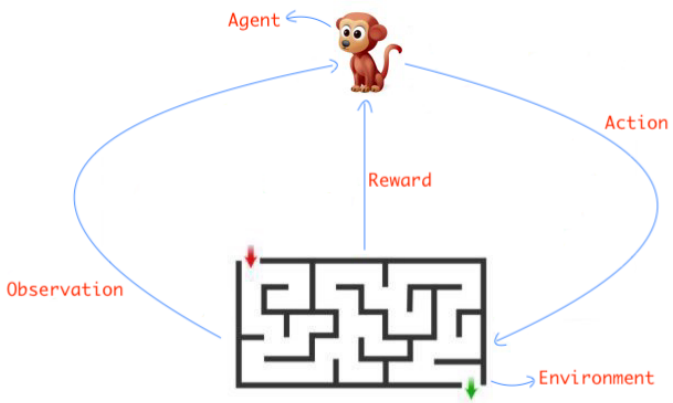
1. Aspect théorique
   - Le cadre
   - Prérequis
   - Multi-Agent Acteur-Critique
2. Aspect pratique
   - Apprentissage pour système d'agents mixtes
   - UML
3. Axes d'orientation possibles

Le cadre : Apprentisage par renforcement multi-agents

- **Motivation**: De nombreuses applications impliquent l'intéraction de plusieurs agents:
  - Navigation multi-robot (ex: voiture autonome)
  - Analyse des dilèmmes sociaux (ex: Sequential Social Dilemnas)
  - Intéragir de manière utile avec l'homme
  - Jeux multi-joueurs (ex: football)
- **Objectif**: Apprendre un comportement pour chaque agent permettant un comportement **global optimal**.
- **Difficultés**: Les méthodes de RL mono-agent s'adaptent mal au cas multi-agents.
  - Environnement non-stationaire $\rightarrow$ Apprentisage très instable

- **Types d'agents**:
  - Agents indépendants: but ne sont pas liés
  - Agents collaboratifs: difficile de modéliser les bénéfices propres à chacun
  - Agents compétitifs: buts opposés

- **La base**: Apprentisage par renforcement mono-agent



Definition

Un **processus de décision markovien** (MDPs) est une framework mathématique permettant de décrire un agent évoluant dans un environnement.

Definition

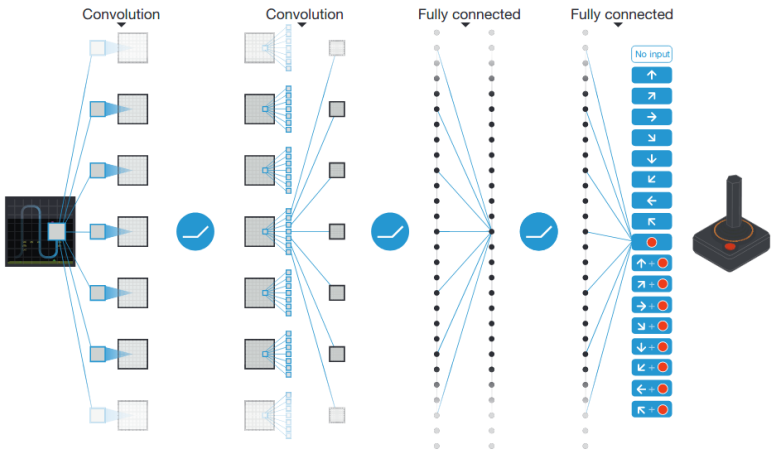
Les **jeux stochastiques** (ou de Markov) sont une extension des processus de décision markovien (MDPs) pour le cas de N agents.
Un jeu markovien est décrit par un tuple $\{\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{O}, \mathbf{r}, \rho, \mathbf{o}\}$:

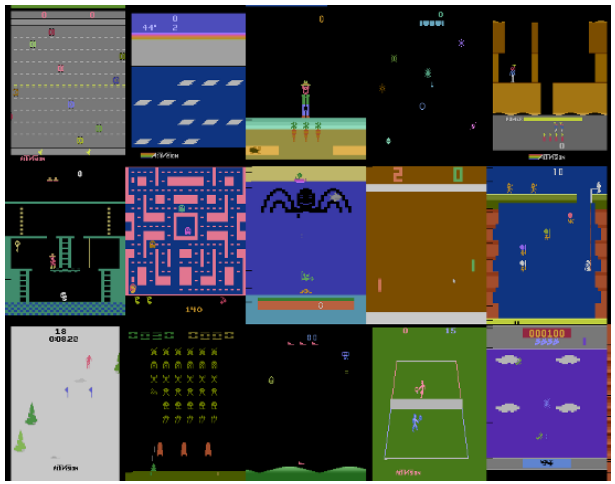
- $\mathcal{S}$: espace d'états
- $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_N\}$: ensemble d'espaces d'actions
- $\mathcal{O} = \{\mathcal{O}_1, \mathcal{O}_2, ..., \mathcal{O}_N\}$: ensemble d'espaces d'observations
- $\mathcal{T} : \mathcal{S} \times \mathcal{A}_1 \times ... \times \mathcal{A}_N \to \mathcal{S}$: fonction de transition
- N fonctions de récompense r_i, un état initial ρ et N fonctions d'observation o_i

Definition: S'il existe au moins un agent i tel que l'observation o_i n'est pas injective alors on se situe dans un jeu de Markov partiellement observable.

Approche 1: Estimation de la fonction de valeur

- Approximation itérativement de la fonction de valeur d'action optimale q_*
  - $\rightarrow$ $q_*(s,a) = \max_\pi \mathbb{E}_\pi \{G_t | S_t = s, A_t = a\}$
    - Cas 1 : Espace d'état fini (Q-learning)
      - Update Q by $Q(S_t, A_t) = (1 - \alpha_t) Q(S_t, A_t) + \alpha_t G_t$
    - Cas 2 : Espace d'état "infini" (DQN)
      - Utilise réseau de neurones pour approximer q_*
      - Utilise *buffer d'expérience replay*
      - Utilise *target network*
      - Minimiser $\mathcal{L}(\theta) = \mathbb{E}_{s,a,r,s'} \{(Q_*(s,a|\theta) - y)^2\}$ *where* $y = r + \gamma \max_{a'} \bar{Q}_*(s', a')$





Cas multi-agent: Environnement non-stationnaire du point de vu de chaque agent.

Approche 2: Optimiser directement la politique π

Pourquoi ?: Apprendre des politiques stochastiques

$\rightarrow$ Mieux quand partiellement observable

Méthode: Maximiser $J(\theta) = \mathbb{E}_{\pi_\theta}\{G_1\}$

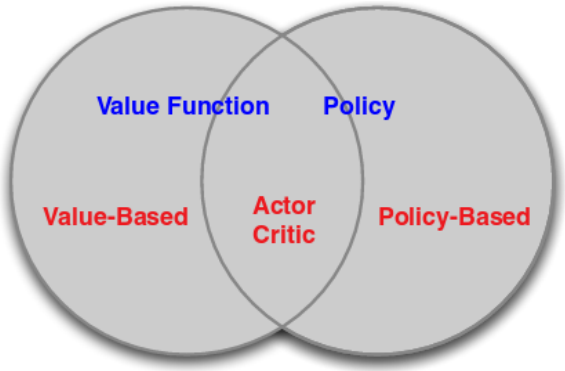
$\rightarrow$ $\nabla_\theta J(\theta) = \mathbb{E}_{s\sim p^\pi,\ a\sim\pi_\theta}\{q_\pi(s,a)\ \nabla_\theta log\ \pi_\theta(a|s)\}$ (policy gradient theorem)

$\rightarrow$ REINFORCE et Actor-Critic algorithmes

Cas déterministe: Deterministic Policy Gradient (**DPG**)

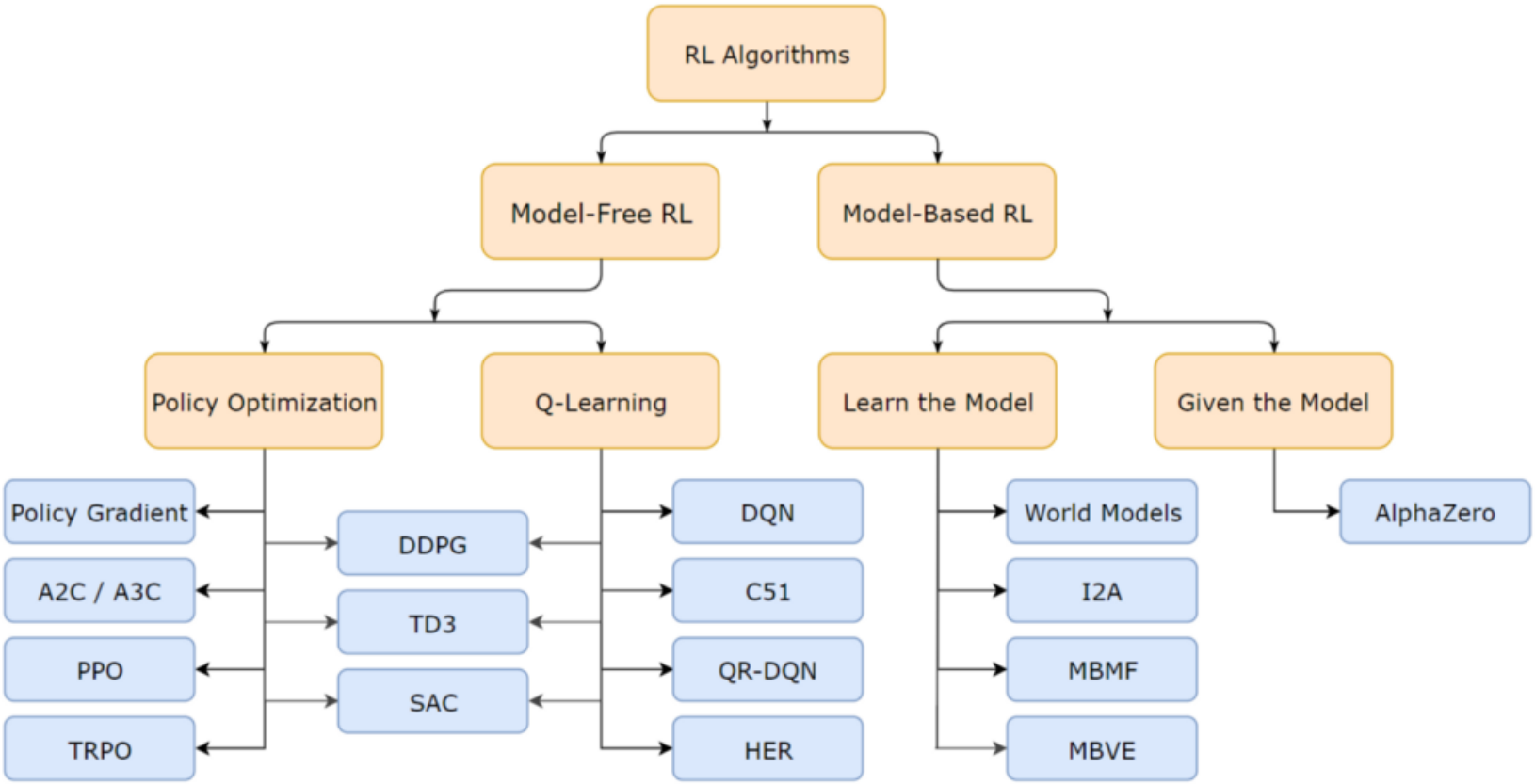
$\rightarrow$ $\nabla_\theta J(\theta) = \mathbb{E}_{s\sim p^\mu}\left\{\nabla_\theta \mu_\theta(a|s)\ \nabla_a q_\mu(s,a)|_{a=\mu_\theta(s)}\right\}$

$\rightarrow$ Espace d'action $\mathcal{A}$ doit être continue



Cas multi-agent: Variance extrêmement élevée. Impossible d'utiliser un point de référence à cause de la non-stationnarité.

Approche 1: Estimation de la fonction de valeur
Approche 2: Optimiser directement la politique π
Approche 3: Utilisation d'un modèle de l'environnement



Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments

Lowe et al. [2018]

Definition

L'apprentissage est dit **décentralisé** si l'agent n'a accès qu'à une information locale, issue de ses capteurs.

Le principe : Adapter les méthodes **Actor-Critic** (DDPG) pour faire face aux problèmes de non-stationnarité de l'environnement et de variances élevée.

Idée : *Si on connait les actions prises par tous les agents l'environnement devient stationnaire même si les politiques changent.*

- $p(s'|s, a_1, ..., a_N, \pi_1, ..., \pi_N) = p(s'|s, a_1, ..., a_N) = p(s'|s, a_1, ..., a_N, \pi'_1, ..., \pi'_N)$

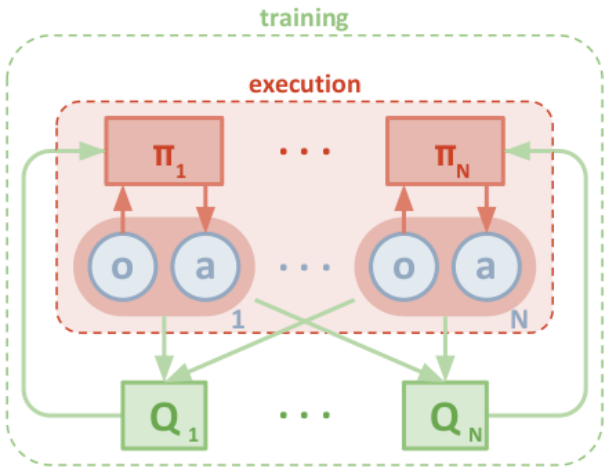
Solution proposée : Apprentisage centralisée et politique déterministe

→ Permet une **exécution décentralisée**

→ L'**acteur** n'utilise que l'information locale

→ Le **critique** utilise les états/actions de tous les agents

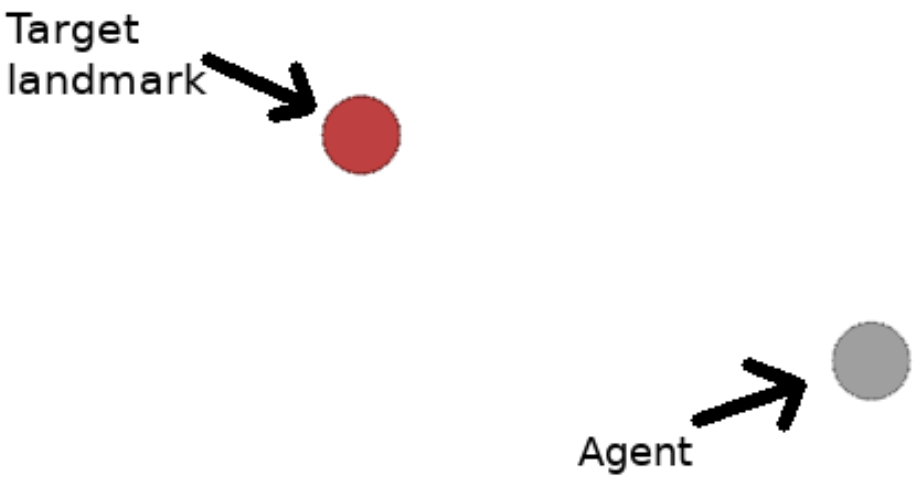
→ Entraîne K sous-politiques pour chaque agent ($K = 3$)



Environnement MADDPG

L'environnement

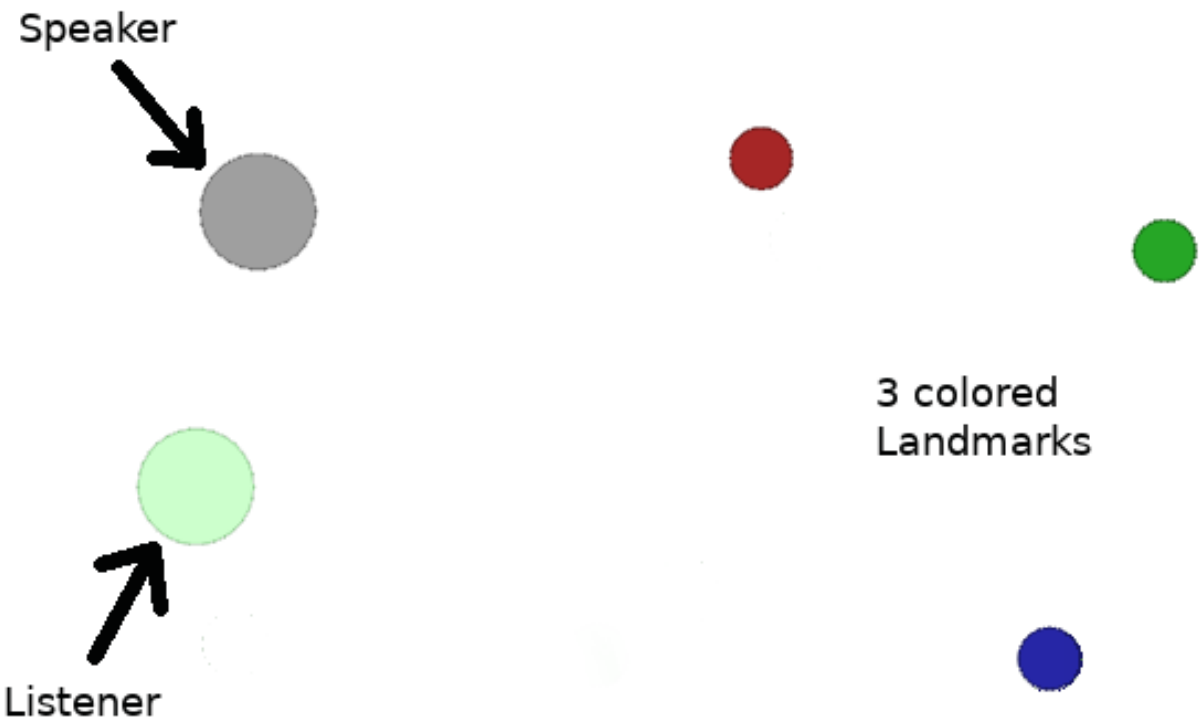
- Simple



Environnement MADDPG

L'environnement

- Simple
- Speaker-Listener

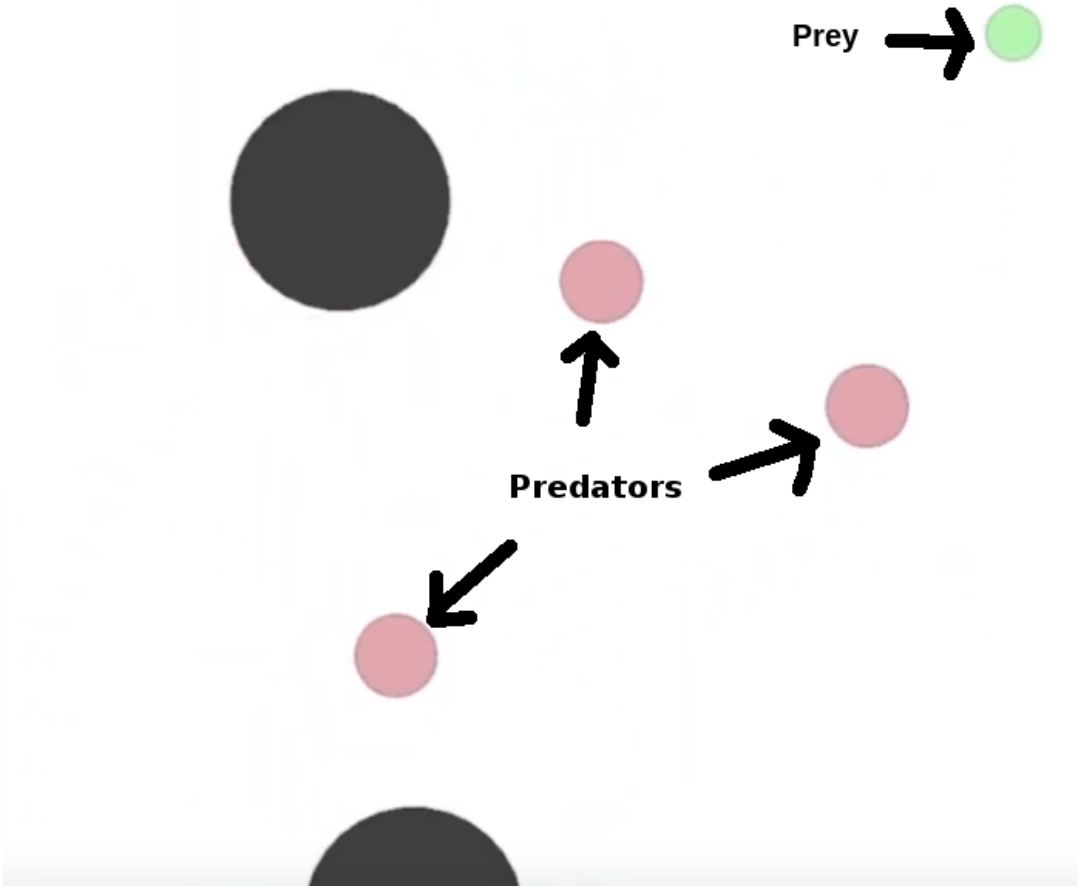


https://www.youtube.com/watch?v=qAUf9zOM70M

Environnement MADDPG

L'environnement

- Simple
- Speaker-Listener
- Prey-Predator



https://www.youtube.com/watch?v=sSltKKwCXbM

1. Aspect théorique
   - Le cadre
   - Prérequis
   - Multi-Agent Acteur-Critique

2. Aspect pratique
   - Apprentissage pour système d'agents mixtes
   - UML

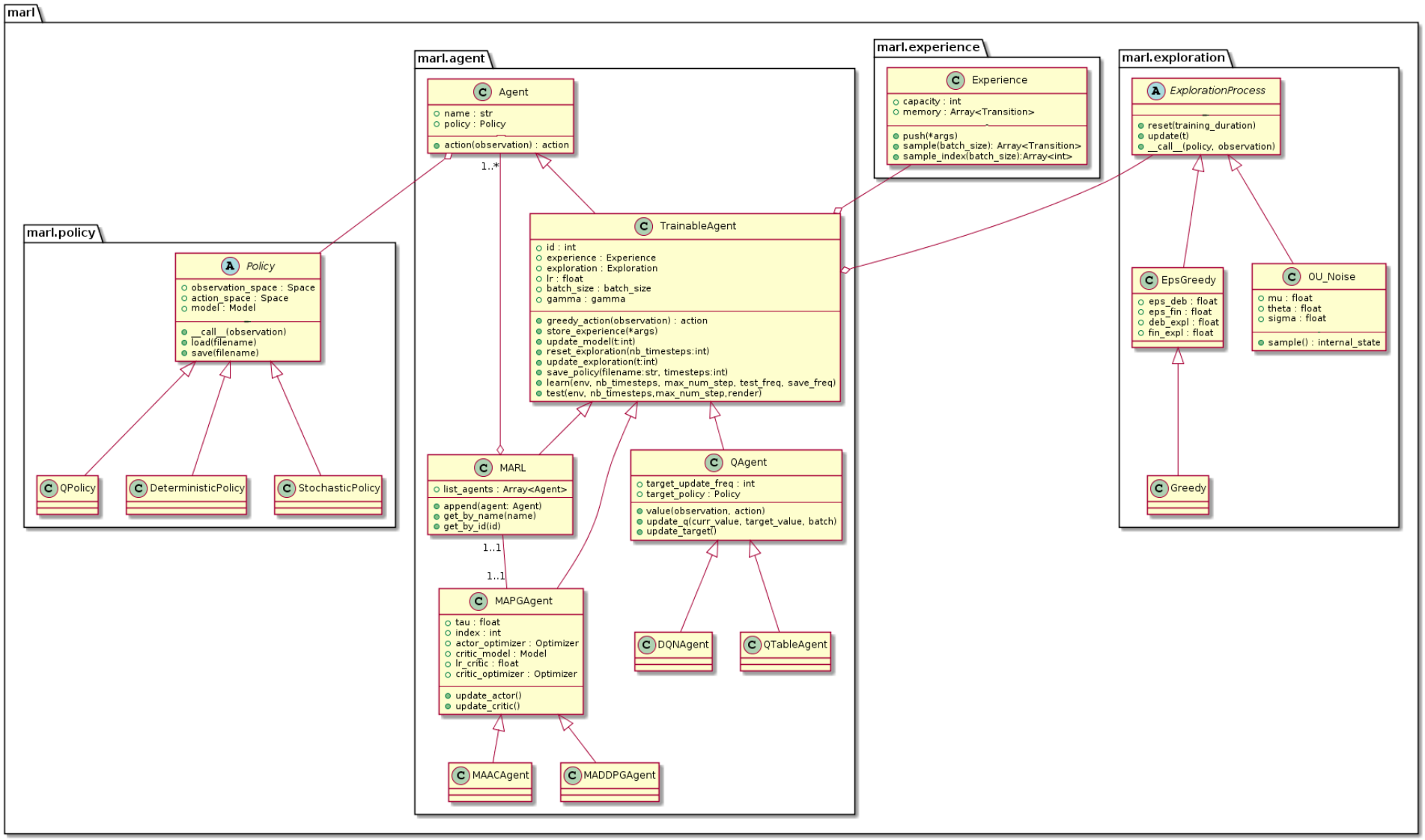
3. Axes d'orientation possibles

Apprentissage pour système d'agents mixtes

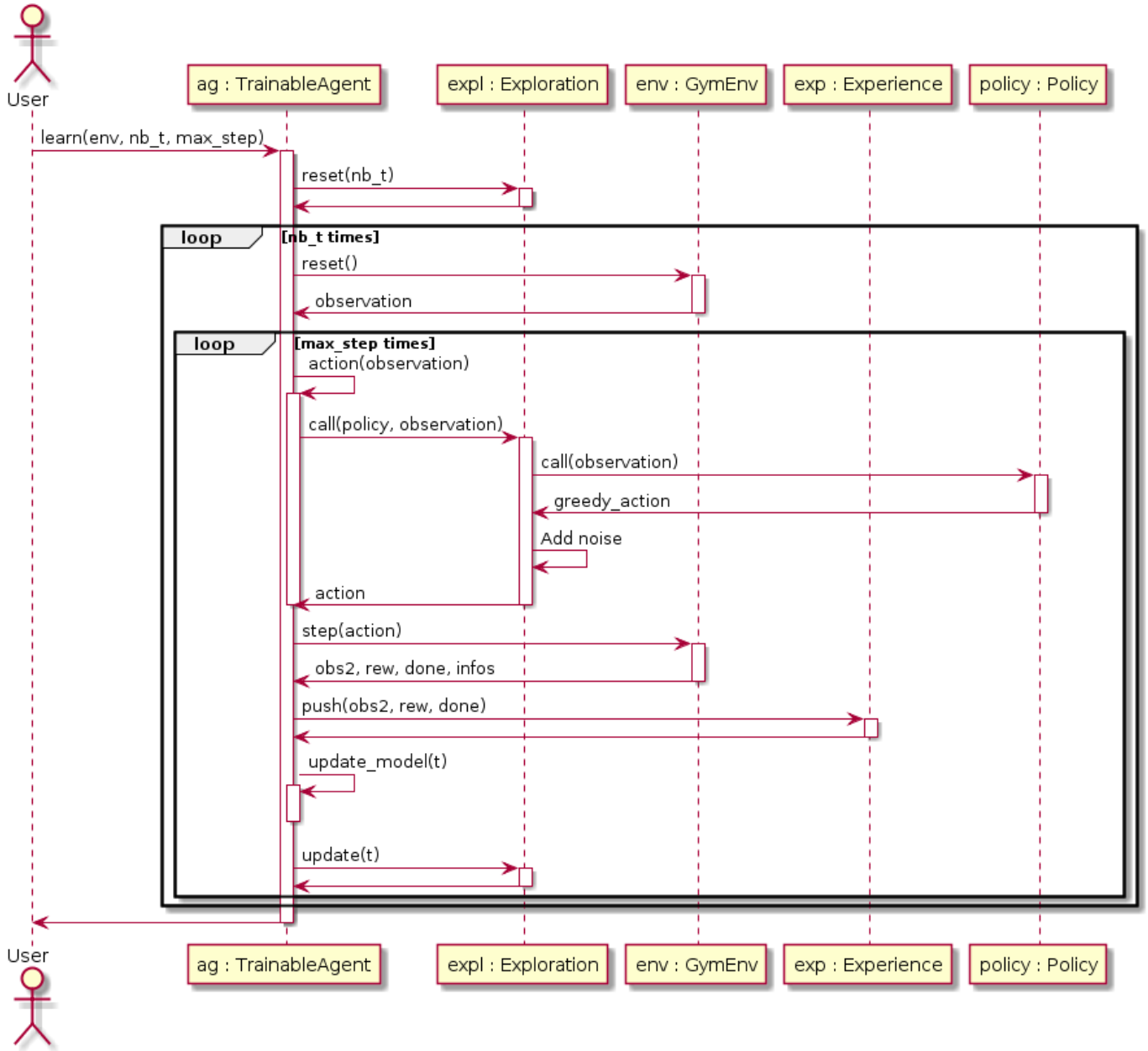
L'idée : Développer une bibliothèque python permettant d'entraîner différent agents avec des méthodes d'apprentissage différentes qui leur sont propre. L'environnement utilisé doit implémenter les fonctions des environnements Gym.
Exemple d'utilisation :

```python
from marl.model import MlpNet, GumbelMlpNet
from marl.agent import DQNAgent, MADDPGAgent
from marl import MARL

# Declare your Gym environment
env = make_env("env_name")

obs_space_0 = env.observation_space[0]
act_space_0 = env.action_space[0]
obs_space_1 = env.observation_space[1]
act_space_1 = env.action_space[1]

# Decalre an agent trained following DQN algorithm
first_agent = DQNAgent( model='MlpNet', observation_space=obs_space_0,
                        action_space=act_space_0, name="DQN_Agent")

# Declare an agent trained following MADDPG algorithm
n_inputs_critic =  obs_space_0.shape[0] + obs_space_1.shape[0] + act_space_0.n
                + act_space_1.n

critic = MlpNet(n_inputs_critic,1)

ag2 = MADDPGAgent(critic_model=critic, actor_model='GumbelMlpNet',
                    observation_space=obs_space_1, action_space=act_space_1,
                    index=1, name="MADDPG_Agent")

# Declare the multi-agent system ()
marl_sys = MARL(agents_list=[ag1, ag2], name='Two-Agent-RL')

# Multi-Agent algorithms as MADDPG usually need to have access to ohter agents attributed
# --> Centralized Training
ag2.set_mas(marl_sys)

# Train simultaneously all agents for 25 000 timesteps
marl_sys.learn(env, nb_timesteps=25000)
```

UML - Diagramme de Classe



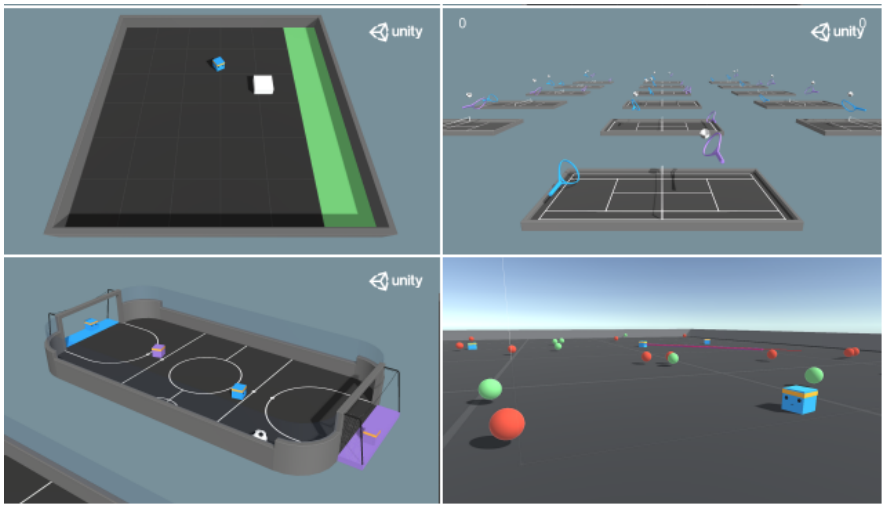
UML - Diagramme de séquence



1. Aspect théorique
   - Le cadre
   - Prérequis
   - Multi-Agent Acteur-Critique
2. Aspect pratique
   - Apprentissage pour système d'agents mixtes
   - UML
3. Axes d'orientation possibles

Orientation du projet

- Orientations possibles/envisagées
  - Continuer développement de la librairie python
  - Tenter d'obtenir des résultats intéressant avec l'algo MADDPG
  - Développer un environnement 3D simple sous Unity et appliquer une méthode d'apprentissage adaptée



  - Etudier méthodes récentes d'apprentissage de communication
  - Lien avec Hierarchical Learning ?