UNIVERSITÉ DE SHERBROOKE

FACULTÉ DES SCIENCES

Implémentation de QMIX et MADDPG dans l'environnement SoccerTwos de Unity

Cours : IFT608-702 - Planification en intelligence artificielle

Elaboré par :

Mohammed Benabbassi - CIP : benm1005

Elio Torquet - CIP : tore1101

Farouk Oussaada - CIP : ousf2802

M. J. Friedman Couthon - CIP : coum3028

Abdou Rahime Daouda - CIP : daoa2504

Sous la supervision de :
Pr. Mohamed Mehdi Najjar

Session d'Hiver 2024

Table des matières

Table des figures

INTRODUCTION

L'intelligence artificielle (IA) a connu des avancées importantes ces dernières années, propulsées par des progrès significatifs dans le domaine de l'apprentissage par renforcement (RL). Le RL est un paradigme d'apprentissage machine dans lequel des agents apprennent à effectuer des tâches en interaction avec avec leur environnement et cherchant à maximiser une récompense cumulative [17]. Cela met en lumière comment les agents peuvent apprendre des stratégies optimales à partir de l'expérience plutôt que d'être explicitement programmés pour une tâche spécifique.

Parallèlement, le domaine des systèmes multi-agents (MAS) explore les interactions entre agents autonomes pouvant coopérer, compétir, ou une combinaison des deux, pour atteindre leurs objectifs. Ces systèmes sont particulièrement pertinents pour modéliser des situations complexes dans des domaines variés tels que la logistique, la robotique, ou encore les jeux vidéo [15]. Cela souligne l'importance de la coordination et de la coopération entre agents pour résoudre des tâches collectivement.

La collaboration et la compétition entre agents dans un cadre multi-agents présentent des défis uniques en termes de planification, de communication, et d'apprentissage. La collaboration requiert des mécanismes permettant aux agents de partager des informations et de prendre des décisions qui bénéficient au collectif, tandis que la compétition nécessite des stratégies permettant aux agents d'optimiser leurs propres intérêts, parfois au détriment des autres [18]. Cela illustre comment des agents peuvent apprendre à collaborer ou à compétitionner dans des environnements complexes à travers le RL, en utilisant l'exemple de jeux vidéo comme plateforme d'expérimentation.

Dans le cadre du cours IFT608-702 - Planification en intelligence artificielle, il nous était demandé d'entraîner des agents, en choisissant les modèles que l'on souhaitait, dans un environnement bien défini. Nous avons décidé d'opter pour **Unity**, qui offre une multitude d'environnements avec interface graphique, plus précisément l'environnement **Soccer two**. Pour entraîner les agents, nous utiliseront deux algorithmes, soient QMIX et MA-DDPG.

ENVIRONNEMENT ET TECHNOLOGIES

1.1 Environnement Unity : Soccer Two

Notre environnement, conçu sur **Unity** [6], simule un match de soccer opposant **deux équipes** de **deux agents** chacune, incarnant ainsi un contexte à la fois compétitif et collaboratif. Chaque équipe poursuit un double objectif : marquer un but le plus rapidement possible tout en s'efforçant de ne pas encaisser de buts.

Pour naviguer dans cet environnement dynamique et prendre des décisions stratégiques, les agents sont équipés de capteurs sous forme de **"lasers"** (voir figure 1.1) placés à l'avant et à l'arrière, leur permettant de détecter leur environnement immédiat ce qui rend l'environnement partiellement observable d'un point de vue des agents. Les actions disponibles pour chaque agent incluent le déplacement en avant/arrière, sur les côtés, ainsi que les rotations, offrant ainsi une gamme de manœuvres pour interagir avec l'environnement et les autres agents. L'espace d'action est multi-discret : une action d'un agent est alors composée de 3 vecteurs *one-hot* qui représentent chacun un type d'action respectivement.

Ce cadre met en valeur l'importance de la stratégie et de la coopération entre les agents de la même équipe pour atteindre les objectifs fixés. Chaque agent doit à la fois adapter ses actions au comportement de son équipier et aux tactiques de ses adversaires.

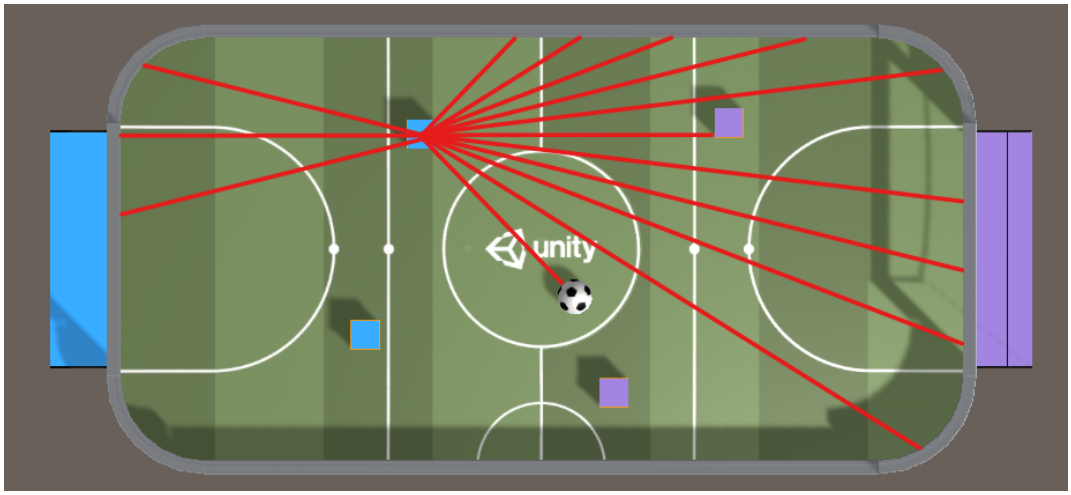


Figure 1.1 – **Environnement SoccerTwo**. En rouge, les *raycasts* constituants les observations de l'agent dans l'environnement par défaut.

1.2 Technologies explorées

Dans le cadre de ce projet, nous avons exploré diverses technologies, chacune présentant ses propres avantages et inconvénients. Après une analyse approfondie, nous avons sélectionné la technologie qui correspondait le mieux à nos contraintes de temps et à notre niveau d'expérience.

1.2.1 ML-agents Trainer/Optimizer

La première technologie que nous avons exploré pour entraîner nos agents dans l'environnement **Soccer Twos** est le plugin **Unity ML-Agents Custom Trainers** [6] qui permet de mettre d'entraîner des algorithmes d'apprentissage par renforcement personnalisés. Ce système de plugins extensible permet de définir de nous modèles d'entraînement basés sur l'API de haut niveau du package ML-Agents. Il propose d'étendre des classes d'entraînement, d'optimiseur, d'hyperparamètres et en particulier pour notre environnement des algorithmes déjà implémentés comme MA-POCA [1]. Pour ajouter un nouveau modèle personnalisé à ML-Agents, on devrait créer un package Python organisé comme à la figure 1.2.

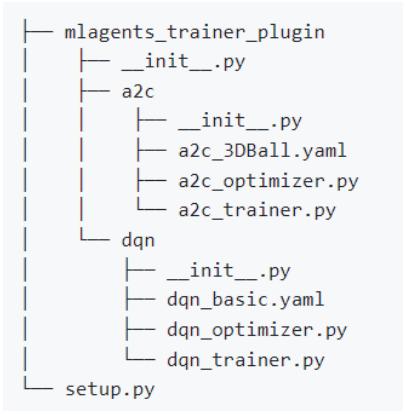


FIGURE 1.2 – Exemple de package Ml-Agents incluant les implémentations des algorithmes A2C[8] et DQN[9]

La mise en place d'un tel package se fait en 4 étapes. La première étape consiste à créer la classe du **trainer** dans un environnement virtuel, étendant les classes *OnPolicyTrainer* ou *OffPolicyTrainer* (selon le type d'algorithme) et implémentant des méthodes pour créer des politiques et faire de l'optimisation, ainsi que la pipeline de traitement pour la mise à jour du modèle. Ensuite, dans une seconde étape, une classe d'optimiseur (**optimizer**) personnalisée est implémentée, prenant une politique fournie par le **trainer** et un dictionnaire de paramètres d'optimisation pour mettre à jour le modèle et calculer les pertes. Dans une troisième étape, on intègre le modèle au système de plugins via un fichier *setup.py* pour enfin dans une quatrième étape installer et commencer l'entrainement tout en spécifiant dans le fichier *yaml* la configuration du processus d'entraînement.

Le plugin ML-Agent vient avec plusieurs avantages. Son intégration profonde avec Unity permet une interaction et une utilisation transparentes au sein de l'environnement Unity permettant une certaine efficacité. Il est optimisé pour Unity et donc garantit de tirer le meilleur parti de Unity

pendant toute l'implémentation. Aussi, il a l'avanatage de prendre en charge les entrées multi-modales (capteurs visuels, audio, etc.), offrant ainsi un outil complet et polyvalent pour diverses applications et type d'environnements. De plus, le plugin est livré avec des exemples et des tutoriels pour différents algorithmes, ce qui le rend accessible.

Bien que puissant, le plugin ML-Agent présente certaines limitations. Sa forte dépendance à Unity est un important inconvénient pour nous qui n'avons pas une grande expérience avec la plateforme. Aussi, le processus de configuration et de mise en œuvre est quelque peu complexe, posant pas mal de défis lorsqu'on veut implémenter un nouvel algorithme. Le débogage est aussi difficile en raison des subtilités du plugin de l'environnement Unity, et la relative faible utilisation de Unity par rapport à d'autres projet open source comme Gymnasium [22] qui peuvent sembler plus flexible.

Compte tenu des contraintes de temps, nous avons donc explorer d'autres solutions dont celle sur HuggingFace [2].

1.2.2 HuggingFace

Cette plateforme facilite l'accès à des modèles d'IA avancés, leur partage et la collaboration sur divers projets d'IA. Plusieurs cours sont d'ailleurs offerts pour ceux qui veulent s'initier au RL. Voici une évaluation des points forts et des limites liés à son utilisation dans le cadre de notre projet.

Points forts

1. **Ressources pédagogiques pour les novices en RL :** La plateforme met à disposition des ressources pédagogiques qui rendent l'apprentissage par renforcement plus accessible aux débutants, contribuant à rendre les technologies d'IA avancées plus atteignables.

2. **Documentation détaillée :** Le succès de HuggingFace tient aussi à la qualité supérieure de sa documentation, qui est complète et bien organisée, ce qui simplifie la recherche d'informations et la compréhension des différents modèles et API pour les utilisateurs.

3. **Beaucoup d'implémentations de modèles et algorithmes disponibles :** La bibliothèque de modèles pré-entraînés et d'algorithmes offerte par la plateforme permet aux utilisateurs d'économiser un temps non négligeable dans la phase de développement.

4. **Évaluation et comparaison des modèles via des challenges :** La plateforme organise des défis et compétitions où les utilisateurs peuvent mesurer les performances de leurs modèles face à ceux d'autres développeurs.

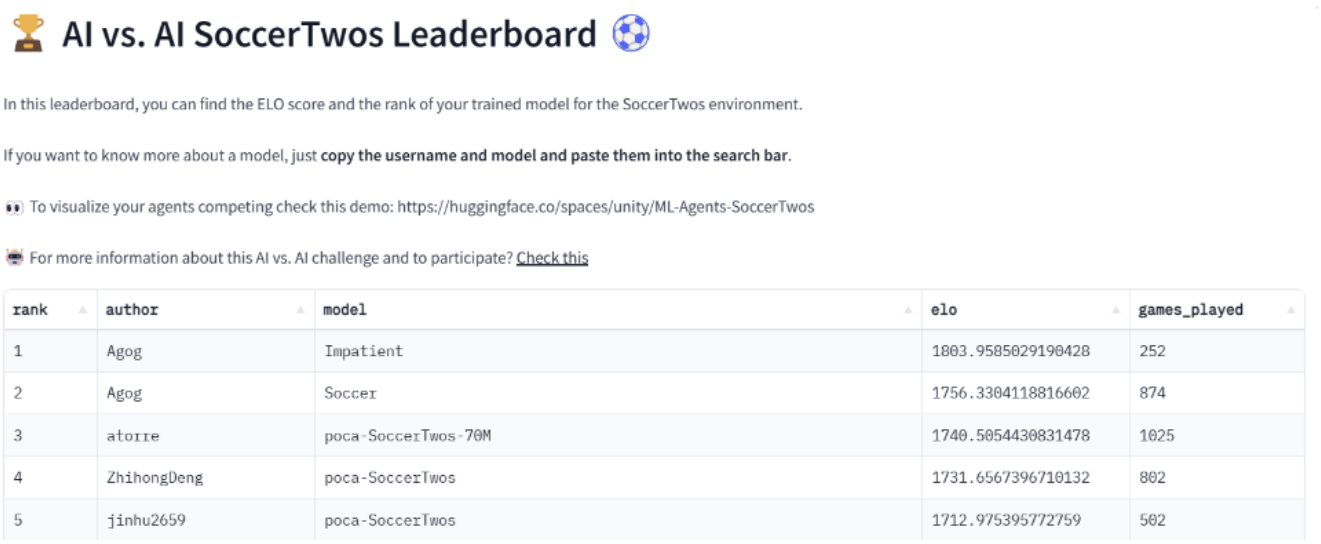


FIGURE 1.3 – **Leaderboard** pour l'environnement SoccerTwo.

Limites

1. **Restrictions sur la personnalisation des espaces d'observations et d'actions :** Ces contraintes peuvent limiter l'adaptabilité des modèles à des cas d'usage spécifiques, restreignant la capacité des utilisateurs à explorer de nouvelles stratégies par exemple pour leurs agents.

2. **Difficulté à intégrer des *trainers* personnalisés :** Malgré la variété d'outils et de modèles prêts à l'utilisation proposés par HugginFace, l'intégration ou le développement de *trainers* sur mesure pour répondre à des besoins particuliers n'est pas pris en charge pour le moment.

Étant donné les limitations majeures que présentait l'environnement sur HuggingFace, nous avons été contraints d'utiliser le framework PettingZoo.

1.2.3 PettingZoo

C'est la technologie pour laquelle nous avons décidé d'opter. Nous avons trouvé qu'elle comblait parfaitement nos besoins, qu'elle offrait des solutions aux problématiques soulevées pour les deux précédentes technologies.

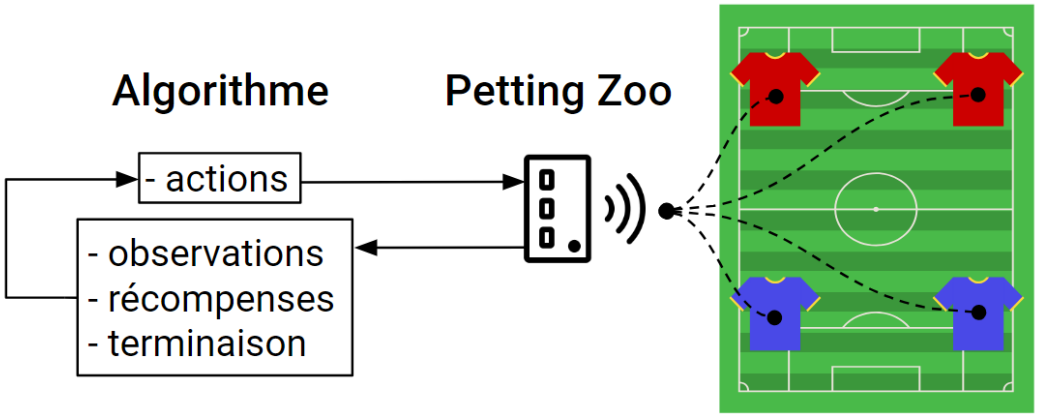


FIGURE 1.4 – **Schéma de l'utilisation de PettingZoo.** Les actions à faire vont être envoyées à l'environnement à l'aide de PettingZoo, qui va aussi renvoyer les observations, les récompenses et les terminaisons au modèle.

Avantages

1. **Interface de contrôle standardisée pour du RL Multi-Agents :** PettingZoo [21] fournit une interface unifiée pour le développement de systèmes d'apprentissage par renforcement multi-agents, facilitant l'interaction et la coordination entre agents. Cette standardisation simplifie considérablement le processus de développement en offrant une base commune pour la construction et l'expérimentation de divers scénarios coopératifs ou compétitifs.

2. **Fonctionnement de l'interface (méthode *step(action)*) :** L'interface de PettingZoo, en particulier la méthode *step(action)*, permet une interaction intuitive avec l'environnement multi-agents. Chaque appel à cette méthode fait progresser l'environnement d'un pas de temps, prenant en compte les actions de chaque agent et retournant leur nouvel état. Cela rend le cycle d'apprentissage plus clair et plus facile à gérer.

3. **Intégration avec Unity via un *Wrapper* :** La capacité de s'intégrer facilement avec Unity grâce à un *Wrapper* permet d'exploiter les capacités graphiques et de simulation de Unity. Le *wrapper* agit comme un pont entre les environnements de simulation 3D de Unity et le cadre d'apprentissage par renforcement de PettingZoo.

4. **Implémentation plus conventionnelle des algos :** PettingZoo favorise une approche plus conventionnelle dans l'implémentation des algorithmes d'apprentissage par renforcement, ce qui peut faciliter la compréhension, l'adaptation et l'amélioration des méthodes existantes. Ainsi, il est plus facile, surtout dans le cadre d'un cours introductif, de mettre le projet en application.

5. **Beaucoup plus documenté et utilisé :** La richesse de la documentation de PettingZoo et son adoption croissante par la communauté de RL garantissent un soutien robuste pour les développeurs. L'accès facile à des exemples, tutoriels et forums de discussion peut permettre de résoudre plus aisément les problèmes techniques rencontrés.

Chapitre 2

État de l'art

Dans le domaine de l'apprentissage par renforcement (RL), en particulier dans l'apprentissage par renforcement multi-agents (MARL), plusieurs cadres et algorithmes ont été développés pour répondre aux complexités de l'entraînement et de l'exécution de politiques dans des environnements avec plusieurs agents en interaction. L'Entraînement Centralisé et Exécution Décentralisée ou en anglais *Centralized Training and Decentralized Execution* (*CTDE*) est un cadre qui a gagné du terrain dans MARL pour sa capacité à exploiter les informations d'état globales pour l'entraînement tout en permettant aux agents de prendre des décisions basées sur les politiques locales pendant l'exécution [25].

Les méthodes basées sur la valeur (*value-based methods*) tels que QMIX (*Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning*) [13], visent à estimer la valeur d'entreprendre une certaine action dans un état donné. Ces algorithmes cherchent la fonction de valeur Q optimale qui maximise la récompense cumulée attendue. QMIX que nous allons présenter plus loin est un algorithme de type *off-policy* qui utilise un réseau critique centralisé pour estimer ces valeurs Q conjointes et une politique décentralisée pour la sélection des actions pendant l'exécution. Il a été démontré qu'il fonctionne bien dans des environnements compétitifs et collaboratifs mais peut rencontrer des difficultés dans les environnements multi-agents où la coordination simultanée est cruciale, car il peut échouer lorsque l'action optimale d'un agent dépend des actions concurrentes des autres [14].

En revanche, quant aux méthodes *policy gradient* comme MA-DDPG (Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments)[7], optimisent directement la politique elle-même en utilisant le gradient de performance de la politique pour mettre à jour les paramètres de la politique. MA-DDPG [7] utilise aussi un réseau critique centralisé pour estimer les valeurs Q pour tous les agents et des réseaux d'acteurs décentralisés pour chaque agent. Même si cet algorithme surpasse les algorithmes RL traditionnels sur une variété d'environnements multi-agents coopératifs et compétitifs, il reste sujet à une augmentation linéaire de l'espace d'entrée de la fonction Q avec un nombre d'agents croissant [7].

2.1 Entraînement centralisé, exécution décentralisée

Le principe de l'entraînement centralisé et d'exécution décentralisée (CTDE) dans l'apprentissage par renforcement multi-agents (MARL) est une approche stratégique qui exploite les atouts des méthodologies centralisées et décentralisées pour optimiser les processus d'apprentissage et d'exécution d'environnement multi-agents. Cette approche est particulièrement pertinente dans les scénarios où le nombre d'agents est important, ce qui rend impossible la centralisation du processus de prise de décision.

En MARL coopératif, une tâche peut être décrite comme un Dec-POMDP [11] constitué d'un tuple $G = \langle S, U, P, r, Z, O, n, \gamma \rangle$ avec $s \in S$ l'état de l'environnement. À chaque pas de temps, chaque agent $a \in A \equiv \{1, ..., n\}$ choisit une action $u^a \in U$, formant une action conjointe $u \in U \equiv U^n$. Une transition sur l'environnement est donc réalisée selon la fonction de transition d'état $P(s'|s, u) : S \times U \times S \to [0, 1]$. Tous les agents partagent la même fonction de récompense $r(s, u) : S \times U \to \mathbb{R}$ et $\gamma \in [0, 1)$ est un facteur d'escompte. Considérons un scénario partiellement observable dans lequel chaque agent a des observations individuelles $z \in Z$ selon la fonction d'observation $O(s, a) : S \times A \to Z$. Chaque agent garde un historique action-observation $\tau^a \in T \equiv (Z \times U)^*$, à laquelle il conditionne une politique stochastique $\pi^a(u^a|\tau^a) : T \times U \to [0, 1]$. La politique commune π a une fonction action-valeur conjointe : $Q^\pi(s_t, u_t) = \mathbb{E}_{s_{t+1:\infty}, u_{t+1:\infty}}[R_t|s_t, u_t]$, où $R_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$ est la récompense individuelle totale attendue.

Entraînement centralisé
L'entraînement centralisé implique l'utilisation d'un modèle centralisé pour apprendre des informations sur l'état global de l'environnement. C'est à dire que l'algorithme d'apprentissage a accès à tous les historiques d'observation d'action locaux τ et à l'ensemble des états S. Cette approche d'agrégation des informations de tous les agents permet l'apprentissage de politiques plus complexes et nuancées. La formation centralisée est bénéfique pour capturer la dynamique globale et les dépendances entre les agents, qui sont cruciales pour une coordination multi-agent efficace. Cependant, cela nécessite des ressources informatiques importantes et peut être difficile à adapter à un grand nombre d'agents [23].

Exécution décentralisée
L'exécution décentralisée, quant à elle, implique de permettre à chaque agent de prendre des décisions basées sur ses observations locales et une politique décentralisée. C'est-à-dire que la politique apprise de chaque agent ne peut dépendre que de sa propre histoire d'observation d'action τ^a. Cette approche est plus efficace en termes de ressources informatiques et peut être plus flexible en termes d'exécution. L'exécution décentralisée permet aux agents d'opérer de manière indépendante tout en bénéficiant des connaissances collectives acquises lors de la phase de formation centralisée. Cependant, cela peut conduire à des politiques sous-optimales si les agents ne communiquent pas ou ne coordonnent pas efficacement leurs actions [23].

2.2 MADDPG

MADDPG est un algorithme de type *policy gradient* polyvalent, applicable non seulement aux environnements coopératifs (avec communication explicites ou non), mais également aux environnements compétitifs. Pour mettre au points cette méthode, les auteurs de MA-DDPG [7] ont du respecter certaines contraintes. Ils partent du principe que les politiques s'appuient uniquement sur des informations locales, que le modèle n'est pas différentiable de la dynamique de l'environnement (contrairement aux études précédentes) et qu'il n'y a pas de structure particulière concernant la communication entre agents (canal de communication non différenciable). MA-DDPG est une approche CTDE : pendant l'entraînement, le critique a accès à des informations supplémentaires que l'agent ne connaîtras pas au cours des inférences. C'est une extension simple des méthodes de type *actor-critic*, dans laquelle chaque critique a une vue globale de l'environnement et de tous les agents

qui s'y trouve. Un aperçu de l'architecture multi-agents de MA-DDPG est présentée à la figure 2.1.

Sachant un environnement N agents et l'ensemble de leur politiques $\pi = \{\pi_1, \ldots, \pi_N\}$ paramétrisées par $\theta = \{\theta_1, \ldots, \theta_N\}$, le gradient des récompenses attendues $J(\theta_i) = \mathbb{E}[R_i]$ pour un agent i est donné par :

$$\nabla_{\theta_i} J(\theta_i) = \mathbb{E}_{s\sim p^\mu, a_i\sim\pi_i}[\nabla_{\theta_i} \log \pi_i(a_i|o_i) Q^{\pi_i}(x, a_1, \ldots, a_N)].$$

avec $Q^{\pi_i}(x, a_1, \ldots, a_N)$ la Q-fonction centralisée (car il prend les actions a_i de tous les agents) pour l'agent i et $x = (o_1, \ldots, o_N)$ les observations de tous les agents. Chaque Q^{π_i} est appris séparément permettant d'inclure ici l'aspect compétitif. Si les politiques sont déterministes, on peut considérer N politiques continues $\mu_i = \mu_{\theta_i}$ et récrire le gradient comme :

$$\nabla_{\theta_i} J(\mu_i) = \mathbb{E}_{x,a\sim D}[\nabla_{\theta_i} \mu_i(a_i|o_i) \nabla_{a_i} Q^{\mu_i}(x, a_1, \ldots, a_N)|_{a_i=\mu_i(o_i)}],$$

Le tampon de relecture de l'expérience (*experience replay buffer*) D contient les tuples $(x, x_0, a_1, \ldots, a_N, r_1, \ldots,$ enrégistrant les expérience de tous les agents. La Q-fonction centralisée est mise à jour avec :

$$L(\theta_i) = \mathbb{E}_{x,a,r,x'}[(Q_i^\mu(x, a_1, \ldots, a_N) - y)^2],$$

$$y = r_i + \gamma Q_i^\mu(x_0, a'_1, \ldots, a'_N)\|_{a'_j=\mu'_j(o_j)},$$

où $\mu' = \{\mu_{\theta'_1}, \ldots, \mu_{\theta'_N}\}$ est l'ensemble des politiques avec des paramètres retardés de θ'_i. La critique centralisée avec des politiques déterministes fonctionne bien en pratique [7].
De prime abord, MA-DDPG [7] nécessite une connaissance des politiques des autres agents pour mettre à jour la Q-fonction. Les auteurs proposent d'assouplir cette hypothèse en permettant aux agents d'apprendre les politiques d'autres agents à partir d'observations. Il suffit donc que chaque agent i garde en plus une approximation $\hat{\mu}_{ji}^\phi$ de paramètre ϕ de la vraie politique μ_j de l'agent j, On peut donc mettre à jour la Q-fonction avec :

$$L(\phi_{ji}) = -\mathbb{E}_{o_j,a_j}[\log \hat{\mu}_{ji}(a_j|o_j) + \lambda H(\hat{\mu}_{ji})],$$

$$\hat{y} = r_i + \gamma Q_i^{\mu'}(x_0, \hat{\mu'}_{1i}(o_1), \ldots, \mu'_i(o_i), \ldots, \hat{\mu'}_{Ni}(o_N)),$$

où H est l'entropie de la distribution de la politique, $\hat{\mu}_i^{'j}$ désigne le réseau cible pour la politique approximée $\hat{\mu}_{ji}$.
Enfin, pour surmonter le problème de non-stationnarité en environnement compétitif où les agents sur-apprennent sur les comportements de leurs adversaires et obtenir des politiques moins changeantes et plus robustes, les auteurs [7] proposent d'entraîner une collection de K différentes sous-politiques, en sélectionnant aléatoirement à chaque épisode, une sous-politique à exécuter par chaque agent. Si on suppose que cette politique μ_i est un ensemble de K différentes de sous-politiques $\mu_i^{(k)}$, pour l'agent i, on maximise :

$$J_e(\mu_i) = \mathbb{E}_{k\sim\text{unif}(1,K), s\sim p^\mu, a\sim\mu_i^{(k)}}[R_i(s, a)].$$

.

Et comme différentes sous-politiques seront exécutée à différents épisode, on maintient le *replay buffer* $D_i^{(k)}$ pour chaque sous-politique $\mu_i^{(k)}$ et on dérive le gradient de l'ensemble des objectifs par

rapport à $\theta_i^{(k)}$ comme suit :

$$\nabla_{\theta_i^{(k)}} J_e(\mu_i) = \frac{1}{K}\mathbb{E}_{x,a\sim D_i^{(k)}}\left[\nabla_{\theta_i^{(k)}}\mu_i^{(k)}(a_i|o_i)\nabla_{a_i}Q^{\mu_i}(x,a_1,\ldots,a_N)|_{a_i=\mu_i^{(k)}(o_i)}\right].$$

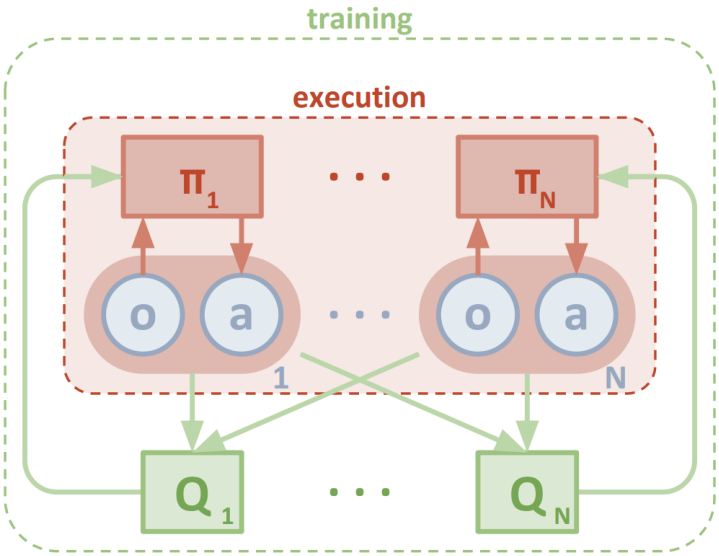


FIGURE 2.1 – Approche multi-agents décentralisée pour l'acteur, centralisée pour le critique.

2.3 QMix

QMIX [13] comme VDN (Value Decomposition Networks) [16] se situe à mi-chemin entre du IQL (Independent Q-learning)[19] (on traduit le problème multi-agent en un ensemble de problèmes simultanés à agent unique) et du Q-learning centralisé comme vu précédemment avec MA-DDPG. L'idée ici est d'effectuer un *argmax* global sur la Q-fonction centralisée Q_{tot}, qui donne le même résultat qu'un ensemble d'opérations argmax individuelles effectuées sur chaque Q-fonction individuelle Q_a :

$$\text{argmax}_u Q_{\text{tot}}(\tau,u) = \begin{pmatrix}\text{argmax}_{u_1} Q_1(\tau_1,u_1),\\ \cdots \\ \text{argmax}_{u_n} Q_n(\tau_n,u_n)\end{pmatrix}$$

Une contrainte supplémentaire sur sur la relation entre Q_{tot} et chaque Qa est donnée par :

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

Et pour appliquer celle-ci, QMIX représente Q_{tot} avec une architecture composée de réseaux d'agents, d'un réseau de *mixture* et d'un ensemble d'hyper-réseaux. La figure 2.2 illustre la configuration globale de QMIX : dans la partie de gauche, les hyper-réseaux en rouge produisent les pondérations et les biais pour "mixer" les couches de réseau en bleu ; au milieu on a l'architecture globale du QMIX , et en droite la structure interne du réseau d'agents. La fonction $Q_a(\tau_a,u_a)$

de chaque agent a est représenté par un réseau d'agents qui n'est qu'un DRQN (Deep Recurrent Qnetworks) [3] qui reçoit l'observation individuelle actuelle $o_{a,t}$ et la dernière action u^a_{t-1} en entrée à chaque pas de temps (voir figure 2.2, partie verte).

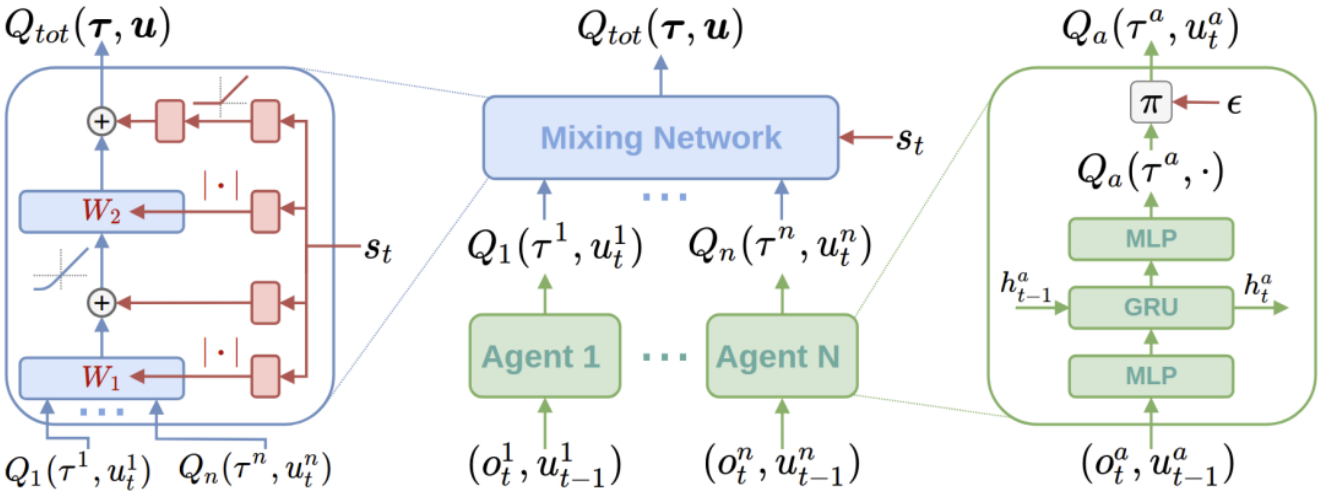


FIGURE 2.2 – Structure du réseau de mixture. En rouge, les hyper-networks qui produisent les poids et les biais pour les couches du réseau de mixture illustrées en bleu. Au milieu, l'architecture globale QMIX. En vert, structure du réseau d'agent.

QMIX utilise un réseau de *mixture*, c'est un réseau de neurones entraîné sur les sorties du réseau d'agents, les "mixe" de manière monotone et produit les Q_{tot} (voir figure 2.2, à gauche). Les poids du réseau de mixture sont produits par des *hyper-networks* séparés composés d'une couche linéaire + fonction d'activation absolue et entraîné sur un état s. Cette façon de procéder permet à l'*hyper-network* de conditionner les poids du réseau monotone sur s de manière arbitraire, intégrant ainsi l'état complet s dans les estimations conjointes de la valeur d'action.
Notons aussi, que QMIX est entraîné de bout-en-bout pour minimiser les la perte :

$$L(\theta) = \sum_{i=1}^{b} \left(y_{\text{tot},i} - Q_{\text{tot}}(\tau, u, s; \theta)\right)^2,$$

où b est la taille du batch de transition échantillonné à partir du *replay buffer*, $y_{\text{tot}} = r + \gamma \max_{u_0} Q_{\text{tot}}(\tau_0, u_0, s_0; \theta^-)$, et θ^- sont les paramètres de réseaux cible comme dans DQN. Enfin, la maximisation de Q_{tot} se fait dans un temps linéaire au nombre d'agents.

Chapitre 3

Méthodologies

Cette section a pour objectif de présenter la façon dont nous avons utilisé les ressources à notre disposition, les modifications que nous y avons apportées et les initiatives qui ont été entreprises en vue d'obtenir de meilleurs résultats.

3.1 Modifications apportées à l'environnement

Dans un souci de performances et de compatibilité avec les algorithmes que nous avons choisi, plusieurs modifications ont été apportées à l'environnement.

3.1.1 Observations

Le système par défaut se compose de multiples *raycasts* permettant de simuler de façon réaliste la vision partielle d'un agent, résultant en un vecteur de 336 observations. Il ne nous a pas semblé réaliste de conserver un système aussi coûteux dans le contexte de notre travail. En effet, les algorithmes d'apprentissage que nous avons choisis sont assez complexes et, à notre connaissance, non implémentés pour résoudre le problème de SoccerTwos. Nous avons donc opté pour un système très simplifié nous permettant de tester, corriger et améliorer notre travail grâce à des tests plus rapides. Les observations de nos agents sont donc totales et se composent de 8 vecteurs à 2 dimensions, représentant respectivement la vélocité de l'agent, la position de son équipier, de ses adversaires, des goals, de la balle, ainsi que la vélocité de cette dernière.

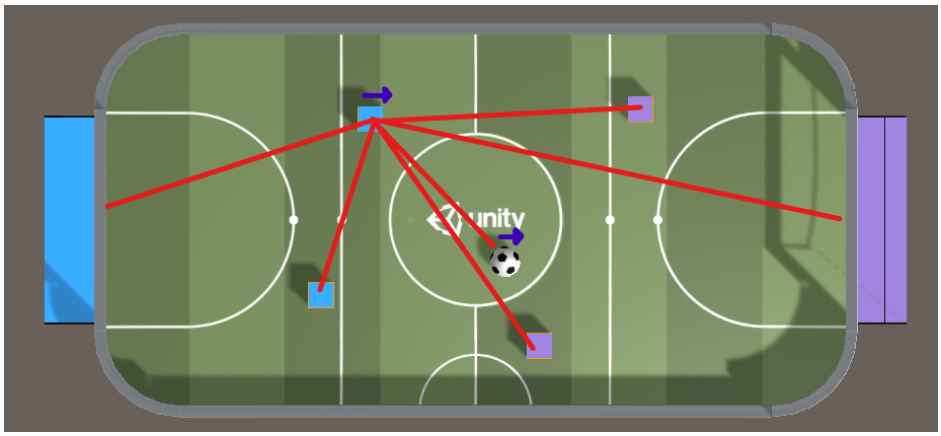


Figure 3.1 – **Environnement SoccerTwo**. En rouge, les positions relatives et en bleu les vélocités observés par l'agent.

3.1.2 Actions

L'espace d'actions des agents fut aussi modifié, mais uniquement pour les agents MA-DDPG. L'espace d'action par défaut et un espace multi-discret alors que MA-DDPG s'adapte plus simplement à des espaces continus. Bien que la *Gumbel-Softmax* [7] est une solution à ce problème, nous avons décidé d'adapter l'environnement pour l'algorithme plutôt que l'inverse. L'espace d'actions des agents MA-DDPG se compose donc de 3 réels bornés entre -1 et 1 représentant : le déplacement longitudinal (z), le déplacement latéral (x) et la rotation (y).

3.1.3 Récompenses

Dans l'environnement d'origine, la récompense (respectivement la pénalité) distribuée aux agents correspond aux buts marqués (respectivement aux buts encaissés). Cependant, cela donne peu d'indices aux agents concernant le comportement attendu. Pour les guider et les aider à découvrir leur objectif principal, nous avons ajouté un bonus et un malus intermédiaire :

- Si un agent tape la balle en direction du but adverse il reçoit une récompense : $0.2 \times$ *ball touch*.
- En revanche si il tape la balle dans en direction de son propre but il reçoit un petit malus : $0.01 \times$ *ball touch*.

Où *ball touch* est une valeur qui décroît au fil de l'entraînement, pour donner plus d'importance à l'objectif final.

3.2 Adaptation PettingZoo

Pour établir la communication entre nos algorithmes et l'environnement nous avons utilisé le Wrapper PettingZoo fournit par Unity [20]. Cependant, les implémentations sur lesquelles nous nous sommes basés n'étaient pas spécifiquement conçues pour PettingZoo ou Unity. Même si elles utilisaient une interface de communication standard très proche de PettingZoo, ce qui a d'ailleurs facilité l'adaptation, plusieurs ajustements ont été nécessaires. Nous avons donc rajouté une interface qui remplit principalement 2 rôles.

Le premier rôle est de formater les données envoyées et reçues par l'algorithme, par exemple, lors de l'exécution de multiples instances de l'environnement nous nous assurons que les actions sont envoyées au bons environnements et que les observations reçues sont regroupées de manière à faciliter et optimiser les calculs ultérieurs. Le second est de gérer des canaux (*channels*) pour communiquer directement avec l'environnement et ainsi modifier ses propriétés depuis l'algorithme.

Enfin, notre interface apporte aussi des corrections au Wrapper de Unity : lors de l'envoi d'actions continues, les valeurs étaient transformés en entiers et certaines valeurs étaient dupliquées.

3.3 Algorithme policy/value-gradient : MADDPG

Nous avons opté pour MADDPG [7], un algorithme qui n'était pas préalablement implémenté dans Unity ML-Agents. La décision d'adopter MADDPG a été guidée par son adéquation avec les caractéristiques de l'environnement Soccer Twos, qui nécessite un algorithme capable de prendre en compte la collaboration et la compétition. Notre adaptation s'appuie sur le dépôt GitHub [4] qui propose une implémentation de l'algorithme pour une interface Gymnasium. Nous disposons donc d'une base solide à laquelle nous avons ajouté de multiples modifications, pour que le script réponde à parfaitement aux spécificités de notre environnement.

Lors de notre mise en œuvre de MADDPG, nous avons pu utiliser des éléments clés tels que le Replay Buffer. Essentiel pour l'apprentissage *off-policy*, il sauvegarde un certain nombre d'itérations dans lesquelles on piochera aléatoirement lors de la phase d'update. L'exploration est introduite par le procédé d'*Ornstein-Uhlenbeck* aussi appelé *OUNoise*, un processus qui ajoute une certaine quantité de bruit aléatoire aux actions prises par l'agent pour l'aider à découvrir de nouvelles stratégies. Cette composante aléatoire est cruciale au début de l'apprentissage, et nous avons implémenté un mécanisme de décroissance de ce bruit au fil du temps pour favoriser l'exploitation des stratégies apprises à mesure que l'agent apprend.

Afin d'ajouter la gestion des espaces d'actions discrets, nous avons recours à la *Gumbel-Softmax* [7], une méthode qui transforme les sortie du réseau de neurones en probabilités sur l'ensemble des actions possibles. En réalisant cette transformation, on répond à 2 problèmes posés par les actions discrètes. D'abord, les sorties du réseau deviennent différenciables, ce qui est indispensable pour calculer la perte et réaliser la rétro-propagation. Aussi, il ajoute de la stochasticité sur le choix des actions, et donc de l'exploration.

3.4 Algorithme action/state-value : QMIX

Contrairement à des algorithmes classiquement utilisés dans Unity ML-Agents tels que MA-POCA [1], DQN[9] et autres, nous avons opté pour QMIX, tout comme MA-DDPG, n'était pas initialement implémenté dans cet environnement ni sur les autres environnements Unity. Ce choix a introduit des défis supplémentaires en raison de l'absence d'une base d'implémentation directe dans Unity ML-Agents. Néanmoins, l'utilisation de PettingZoo a facilité l'adaptation de QMIX en nous permettant de nous inspirer d'un dépôt GitHub [24] existant qui intègre QMIX avec PettingZoo pour des environnements multi-agents. Notre choix d'approche a été guidé par les particularités de notre environnement Soccer Twos, qui implique à la fois collaboration et compétition. QMIX s'est donc révélé particulièrement approprié grâce à sa capacité à gérer de manière efficace les dynamiques complexes entre les agents dans des situations multi-agents.

Notre point de départ a été l'article sur QMIX [13] ainsi que le dépôt GitHub [24]. Cela nous a fourni une base solide pour implémenter, réutiliser et adapter divers éléments en vue d'optimiser les performances de l'apprentissage par renforcement dans notre environnement spécifique. Les mécanismes d'exploitation essentiels à l'apprentissage profond, tels que la sauvegarde et le chargement

des modèles, ainsi que la parallélisation des calculs via CUDA, ont été directement réemployés pour accélérer le processus d'entraînement. La communication stratégique entre les agents et le réseau de mixture a également été un élément clé réutilisé, assurant une coopération et une compétition efficaces au sein de notre environnement.

L'adaptation de QMIX à notre environnement a nécessité des modifications significatives, notamment le remplacement des perceptrons multicouches (MLP) par des réseaux de neurones récurrents (RNN [5]). Les RNNs offrent des avantages considérables par rapport aux MLP dans notre contexte, notamment leur capacité à mémoriser et traiter les séquences d'actions précédentes [10], ce qui est crucial dans un environnement dynamique multi-agents. De plus, l'initialisation des poids des réseaux neuronaux et l'ajustement de l'architecture du réseau de mixture ont été adaptés pour répondre spécifiquement aux exigences de notre étude.

En ce qui concerne la gestion des actions multi-discrètes, notre environnement Unity contient trois actions discrètes, codées sous forme de vecteur "*one-hot*" : une pour avancer, une pour se déplacer latéralement et une pour la rotation. Cette particularité a nécessité une adaptation de QMIX pour traiter efficacement cet ensemble d'actions multi-discrètes. Nous avons considéré toutes les combinaisons possibles des trois actions et utilisé l'*argmax* comme sortie pour décider de l'action à entreprendre. Cette adaptation a permis une intégration fluide des actions spécifiques à notre environnement dans Unity dans le cadre de l'algorithme QMIX.

Enfin, nos contributions ont également inclus l'intégration à l'environnement Unity via PettingZoo pour une communication efficace, l'implémentation d'une paramétrisation et la généralisation via des fichiers *yaml*, et enfin le développement d'un *ReplayBuffer* adapté pour les RNN afin de gérer les dépendances temporelles. Un effort considérable de refactoring et de nettoyage du code a été réalisé pour assurer une base de code maintenable et extensible.

3.5 Optimisations & améliorations générales

3.5.1 Instances multiples de l'environnement

La première optimisation fut de gérer plusieurs instances de l'environnement en parallèle. Grâce à un exécutable, on lance plusieurs environnements avec des "seeds" aléatoire différentes. Du côté du modèle, il y a toujours seulement 4 agents, mais un même agent prend des décisions pour un joueur de chaque environnement. Grâce à ce système, couplé à l'utilisation du multi-processing, on exploite de façon optimale les ressources matérielles pour multiplier la quantité d'expérience produite.

3.5.2 Parallelisation des calculs

La phase la plus coûteuse en temps et en ressource est certainement la phase de mise à jour du modèle : on envoie de grand batch de données dans les réseaux de neurones et on calcule les pertes pour mettre à jours les poids. C'est d'autant plus vrai pour notre environnement car il s'agit de mettre à jour 4 *actors* et 4 *critics*. Pour améliorer le temps de calcul et l'utilisation des ressources

nous utilisons Torch [12], une bibliothèque pour les calculs et l'apprentissage automatique. Si l'infrastructure le permet, les calculs seront réalisés de façon optimisée, sur le GPU lors des phases de mise à jour du modèle.

3.5.3 Sauvegarde et chargement des modèles

Nous avons utilisé un système de sauvegardes régulières des modèles pendant l'entraînement. Le temps d'entraînement pouvant être très long il est important de garder un historique de l'évolution du modèle : si les agents évoluent trop longtemps vers un comportement sous optimal, on reprendra l'entraînement depuis une version antérieur en essayant de corriger l'erreur.

3.5.4 Paramétrisation via yaml

Pour faciliter au maximum le lancement de l'entraînement et des tests nous avons regroupé l'ensemble des paramètres modifiables dans un fichier *yaml*. On peut y spécifier quasiment l'intégralité des hyperparamètres du modèle et de l'environnement.

3.5.5 Scaling des paramètres

Grace à l'interface que nous avons mis en place par dessus le Wrapper PettingZoo, nous sommes capables de mettre à jour les paramètres de l'environnement au fil de l'entraînement. Nous l'utilisons notamment pour décrémenter les récompenses liées aux touches de balles. Pour faciliter le scaling d'un nouveau paramètre, nous avons abstrait cette fonctionnalité : il faut uniquement préciser les valeurs initiales/finales et l'intervalle sur laquelle le paramètre évolue.

Experimentations

4.1 Conditions expérimentales

Nos expérimentations ont été conduites sur une configuration matérielle uniforme pour les deux algorithmes, à savoir une carte graphique NVIDIA RTX 3050 Ti, un processeur Intel i7-11800H à 8 cœurs (16 threads) et une mémoire vive de 32 Go. Nous avons aussi envisagé l'utilisation d'une machine virtuelle sur Google Cloud pour accélérer l'entraînement, mais cette option sera discutée plus en détail dans la section dédiée aux difficultés rencontrées au cours de notre projet.

4.1.1 Environnement d'expérimentation

Nous avons utilisé la version $2022.3.4f1$ de Unity et la release 21 de ML-Agents pour nos expérimentations et la version Python 3.10.12. Des modifications spécifiques ont été apportées à l'environnement Unity Soccer Two, comme discuté précédemment dans la section méthodologie, incluant l'adaptation des paramètres de l'environnement et la personnalisation des scénarios d'interaction agent-environnement pour répondre à nos besoins d'expérimentation.

4.1.2 Hyperparamètres et configuration d'entraînement

Pour l'entraînement de MADDPG et QMIX, nous avons défini un ensemble d'hyperparamètres clés sans recourir à des méthodes de recherche d'hyperparamètres avancées, principalement en raison des coûts élevés en ressources et du temps d'entraînement considérable. Les hyperparamètres incluaient :

Hyperparamètre	**MA-DDPG**	**QMix**
Nombre de couches cachées	2	2
Taille des couches cachées	128	128
Taux d'apprentissage (*actor*)	3e-4	1e-4
Taux d'apprentissage (*critic*)	1e-3	1e-3
Tau (*target update rate*)	1e-3	1e-3
Gamma (*discount factor*)	0.9	0.95
Taille des batch	1024	1024
Taille du *replay buffer*	10e5	10e5
Itération par *update*	512	512
Évolution du bruit	$10.0 \rightarrow 0.0$	$10.0 \rightarrow 0.0$

4.1.3 Durée d'entraînement et répartition des mises à jour

L'entraînement pour chaque algorithme a duré approximativement **31 heures** pour un total de **25 mille épisodes**. Les modèles furent mis à jour toutes les 500 itérations, avec des batches de 1000 itérations, assurant une répartition uniforme et un volume substantiel de données pour l'apprentissage.

4.1.4 Mesures de performance et critères d'évaluation

La performance des algorithmes a été évaluée selon plusieurs critères, incluant la récompense cumulative, la longueur des épisodes, et le score ELO. Ces mesures nous ont permis de quantifier l'efficacité et la progression de l'apprentissage au cours des sessions d'entraînement, en offrant une vue d'ensemble des capacités développées par les agents.

4.2 Stratégies d'entraînement

Lors des premiers entraînements nous avons remarqué qu'une équipe pouvait plus ou moins prendre le dessus sur l'autre, ce qui pénalise et freine l'entraînement des deux équipes. En effet, pour les agents de l'équipe dominée, le jeu devient plus difficile, ne leurs laissant pas le temps d'explorer des stratégies pour apprendre. D'un autre côté, l'équipe dominante gagne plus facilement et donc voit ses agents s'améliorer faiblement. C'est un problème connu dans les environnements compétitifs : le résultat d'une action dépend à la fois de l'environnement et du niveau de l'adversaire.

La solution que nous avons trouvé pour QMix est le self-play : le modèle affronte une version anterieur de lui-même, régulièrement mise à jour. Cette stratégie d'entraînement permet aux agents de s'adapter de manière dynamique à des adversaires en évolution, favorisant ainsi une amélioration continue et la découverte de stratégies compétitives plus robustes.

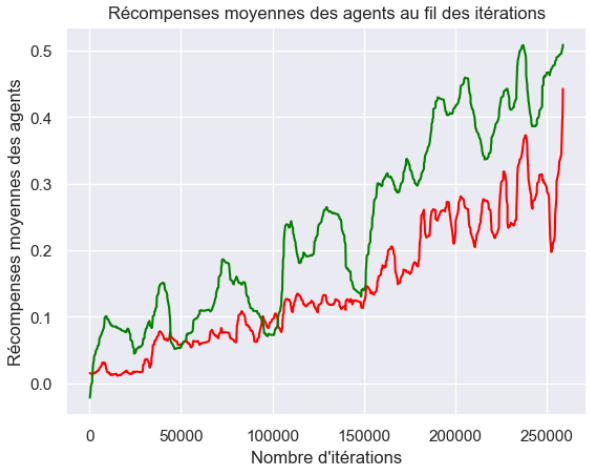
Pour MADDPG, qui s'adapte mieux aux environnements compétitifs, nous avons mis en place un mélange régulier des équipes. Comme toutes les observations du modèle sont relatives, la mise en place de ce mélange à été facile. Cela à permis d'obtenir une évolution plus stable et équitable des politiques et des performances.

Résultats

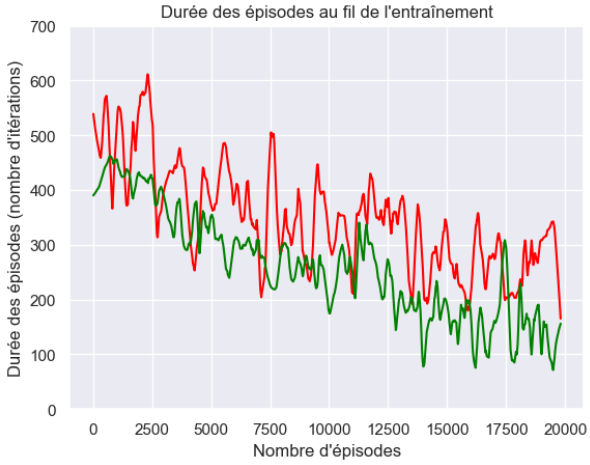
L'entraînement des agents MA-DDPG et QMIX sur 20 000 épisodes de 1000 itérations a produit des résultats très prometteurs. Même si nous pensons que les paramètres des modèles ne sont pas optimaux, il est clair que nos agents sont capables d'apprendre et de s'améliorer.

(a) Évolution des pertes de MADDPG. Critic loss (en orange) et Actor loss (en bleu).

(b) Évolution de la *agent loss* de Qmix au fil des itérations



(c) Récompenses moyennes cumulées. En rouge MADDPG et en vert Qmix



(d) Durée des épisodes. En rouge MADDPG et en vert Qmix

Figure 5.1 – Statistiques de l'entraînement au fils des itérations

5.1 La perte/le coût

En premier lieu, on peut remarquer que la *loss* du réseau *critic* 5.1a se stabilise et que celle du réseau *actor* diminue au fil de l'entraînement. C'est un signe que la politique de l'agent s'améliore. Cependant, après 15 millions d'itérations, la *loss* de l'*actor* remonte : on peut attribuer cette baisse de performance à la façon dont nous distribuons les récompenses. En effet, à ce stade, les récompenses données pour toucher la balle sont nulles et donc, l'agent doit réussir à se concentrer sur l'objectif final, il est donc possible que l'agent se soit trop appuyé sur cette récompense. En observant le comportement d'un modèle sauvegardé à cette étape, nous avons pu observer que, effectivement, les agents avaient déviés de leur objectif et collaborés tous ensemble pour maximiser leur touches de balles. Dans le même ordre d'idée, il est observable que la valeur de *loss* de l'agent, illustrée en bleu sur le graphique 5.1b, décroît de manière stable au fur et à mesure des itérations, ce qui peut être interprété comme une amélioration de l'estimation de la valeur des états par le réseau de l'agent. Cette diminution constante suggère que l'agent QMix affine son estimation de la valeur de l'état au cours de l'entraînement, ce qui est un indicateur d'un apprentissage efficace.

5.2 Récompenses moyennes cumulées

Par ailleurs, on observe que les récompenses moyennes cumulées par tous les agents 5.1c augmentent continuellement ce qui dénote aussi une amélioration de la politique. Il serait intéressant de voir la proportion de récompenses attribués pour des buts et celles attribuées pour des touches de balles. L'accroissement soutenu de la courbe verte 5.1c associée à QMix indique une amélioration continue des politiques des agents. Ce progrès illustre une capacité croissante des agents à collaborer et à maximiser la récompense collective. Cependant, la présence de variations suggère que les agents ont encore des difficultés à consolider leurs stratégies sur certains épisodes. On pense que l'attribution des récompenses joue un rôle sur cet aspect, et on suggère explorer un décroissement des récompenses de touches de balles au pendant l'entraînement. Une hypothèse alternative pour expliquer ces fluctuations pourrait être une spécialisation excessive des agents dans des rôles ou des stratégies spécifiques, limitant leur flexibilité et adaptabilité. Les agents pourraient alors exceller dans certains scénarios mais peiner à ajuster leur comportement dans des contextes variables ou inattendus, conduisant à une performance inégale. Pour contrer cela, il serait pertinent d'introduire une régularisation dans l'apprentissage ou d'incorporer un entraînement sur des scénarios plus diversifiés.

5.3 Durée des épisodes

Finalement, la durée des épisodes de QMix et MADDPG, bien qu'elle présente une variabilité marquée comme le montre la figure 5.1d, montre une tendance à la baisse sur le long terme. Cette réduction indique que les agents apprennent à atteindre leurs objectifs, en l'occurrence marquer des buts, plus efficacement et rapidement. Malgré cette instabilité, la tendance générale vers des épisodes plus courts peut être interprétée comme une progression dans la coordination et l'efficacité des agents.

Chapitre 6

Perspectives futures

A ce jour, le travail réalisé se concentre principalement sur l'intégration de nouveaux algorithmes pour l'environnement de Ml-agents soccerTwo. Il reste encore plusieurs points que nous aimerions approfondir, nous les présentons dans cette section.

6.1 Étude des performances et recherche de paramètres

Les expérimentations réalisées, bien qu'elles ne soient pas nombreuses, révèlent plusieurs points intéressants : nos implémentations sont fonctionnelles et le choix des paramètres d'entraînement est cruciale. Nous avons relevé par exemple que des récompenses mal calibrées peuvent écarter les agents de leur but ou qu'insister sur l'exploration permettait aux agents de s'améliorer plus rapidement.
Dans ce contexte il semble indispensable de réaliser une étude approfondie pour trouver les paramètres optimaux. L'acquisition d'une machine virtuelle serait nécessaire pour cette étape.

6.2 Robustesse et adaptabilité des algorithmes

Comme nous l'avons détaillé dans la section 3.1 (modifications apportés à l'environnement) plusieurs modifications ont été apportés pour faciliter le processus d'implémentation et de test. Actuellement l'observabilité de l'environnement est total, or les modèles que nous avons choisi sont reconnus pour leur capacité à évoluer dans un environnement à l'observabilité partielle. Entraîner nos agents dans un environnement partiellement observable nous permettrait à la fois de valider leur robustesse face à différents environnements et aussi de les comparer avec les autres modèles disponibles dans Ml-agents (par exemple MA-POCA).

6.3 Portabilité de nos implémentations

L'implémentation que nous avons développé est déjà quasiment indépendante de l'environnement SoccerTwos. Nous aimerions donc la tester sur d'autres environnements multi-agents similaires pour observer ses performances.

CONCLUSION

6.4 Bénéfices

Dans le cadre de notre projet, les choix algorithmiques ont constitué un pilier fondamental de notre réussite, offrant une adaptabilité et une pertinence face aux exigences de l'environnement étudié. L'architecture modulaire de l'implémentation, facilitée par PettingZoo, a permis une certaine adaptabilité, potentiellement transposable à d'autres environnements Unity ou PettingZoo sans modifications majeures. La documentation élaborée pour naviguer dans les complexités de ML-Agents s'est avérée être un atout, simplifiant le processus d'intégration pour l'équipe et pouvant servir de référence pour des initiatives similaires (prochains étudiants du cours IFT608-702). Une compréhension approfondie de ML-Agents a été cruciale pour adapter les algorithmes à l'environnement, optimisant ainsi les performances. L'adoption de la parallélisation des calculs sur GPU, le lancement de plusieurs instances et l'utilisation de Torch ont contribué à réduire le temps d'entraînement, témoignant de l'efficacité de ces technologies dans l'accélération du développement des modèles.

6.5 Difficultés

Cependant, le projet a rencontré des obstacles significatifs, principalement dus à la complexité intrinsèque de ML-Agents et à un manque de documentation pour certains de ses aspects, notamment pour son installation et pour l'ajout d'un *trainer/optimizer* personnalisé. Les algorithmes, bien que avancés, ont requis un investissement important en termes de temps et de ressources pour l'entraînement, rendu compliqué par une documentation limitée et une faible disponibilité d'exemples pratiques. Une approche progressive, commençant par des environnements plus simples (Atari de Gymnasium par exemple), aurait peut-être permi une meilleure familiarisation avec les principes de l'apprentissage par renforcement avant de s'attaquer à une configuration multi-agents, compétitive et collaborative. Enfin, l'accès à des ressources matérielles, telles que des machines virtuelles sur Google Cloud et Azure, a présenté des défis, soulignant l'importance de l'accès à des infrastructures de calcul adéquates.

Bibliographie

[1] Andrew Cohen et al. "On the Use and Misuse of Absorbing States in Multi-agent Reinforcement Learning". In : *RL in Games Workshop AAAI 2022* (2022). URL : http://aaai-rlg.mlanctot.info/papers/AAAI22-RLG_paper_32.pdf.

[2] Hugging Face. *Hands-on*. https://huggingface.co/learn/deep-rl-course/unit7/hands-on. 2024.

[3] Matthew Hausknecht et Peter Stone. "Deep recurrent q-learning for partially observable mdps". In : *2015 aaai fall symposium series*. 2015.

[4] Shariq Iqbal. *MADDPG-PyTorch*. https://github.com/shariqiqbal2810/maddpg-pytorch. 2018.

[5] M I Jordan. "Serial order : a parallel distributed processing approach. Technical report, June 1985-March 1986". In : (1986). URL : https://www.osti.gov/biblio/6910294.

[6] Arthur Juliani et al. "Unity : A general platform for intelligent agents". In : *arXiv preprint arXiv :1809.02627* (2020). URL : https://arxiv.org/pdf/1809.02627.pdf.

[7] Ryan Lowe et al. "Multi-agent actor-critic for mixed cooperative-competitive environments". In : *Advances in neural information processing systems* 30 (2017).

[8] Volodymyr Mnih et al. "Asynchronous methods for deep reinforcement learning". In : *International conference on machine learning*. PMLR. 2016, p. 1928-1937.

[9] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In : *arXiv preprint arXiv :1312.5602* (2013).

[10] Steven Morad et al. "POPGym : Benchmarking Partially Observable Reinforcement Learning". In : *The Eleventh International Conference on Learning Representations*. 2023. URL : https://openreview.net/forum?id=chDrutUTs0K.

[11] Frans A Oliehoek, Christopher Amato et al. *A concise introduction to decentralized POMDPs*. T. 1. Springer, 2016.

[12] Adam Paszke et al. "PyTorch : An Imperative Style, High-Performance Deep Learning Library". In : *Advances in Neural Information Processing Systems 32*. Sous la dir. de H. Wallach et al. Curran Associates, Inc., 2019, p. 8024-8035. URL : http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[13] Tabish Rashid et al. "Monotonic value function factorisation for deep multi-agent reinforcement learning". In : *Journal of Machine Learning Research* 21.178 (2020), p. 1-51.

[14] Tabish Rashid et al. "Weighted qmix : Expanding monotonic value function factorisation for deep multi-agent reinforcement learning". In : *Advances in neural information processing systems* 33 (2020), p. 10199-10210.

[15] Peter Stone et Manuela Veloso. "Multiagent Systems : A Survey from a Machine Learning Perspective". In : *Autonomous Robots* 8.3 (2000), p. 345-383.

[16] Peter Sunehag et al. "Value-decomposition networks for cooperative multi-agent learning". In : *arXiv preprint arXiv :1706.05296* (2017).

[17] Richard S. Sutton et Andrew G. Barto. *Reinforcement Learning : An Introduction*. 2018.

[18] Ardi Tampuu et al. "Multiagent cooperation and competition with deep reinforcement learning". In : *PLOS ONE* 12.4 (2017). URL : https://doi.org/10.1371/journal.pone.0172395.

[19] Ming Tan. "Multi-agent reinforcement learning : Independent vs. cooperative agents". In : *Proceedings of the tenth international conference on machine learning*. 1993, p. 330-337.

[20] Unity Technologies. *ML-Agents : Python PettingZoo API*. https://github.com/Unity-Technologies/ml-agents/blob/develop/docs/Python-PettingZoo-API.md. 2023.

[21] J. K Terry et al. "PettingZoo : Gym for Multi-Agent Reinforcement Learning". In : *arXiv preprint arXiv :2009.14471* (2020).

[22] Mark Towers et al. *Gymnasium*. Mars 2023. DOI : 10.5281/zenodo.8127026. URL : https://zenodo.org/record/8127025 (visité le 08/07/2023).

[23] Jiangxing Wang, Deheng Ye et Zongqing Lu. "More centralized training, still decentralized execution : Multi-agent conditional policy factorization". In : *arXiv preprint arXiv :2209.12681* (2022).

[24] Chen Yang, Tianyu Zhang et DetectiveCode. *PyTorch implements multi-agent reinforcement learning algorithms, including QMIX, Independent PPO, Centralized PPO, Grid Wise Control, Grid Wise Control+PPO, Grid Wise Control+DDPG*. https://github.com/yangchen1997/Multi-Agent-Reinforcement-Learning. 2024.

[25] Yihe Zhou et al. "Is centralized training with decentralized execution framework centralized enough for marl ?" In : *arXiv preprint arXiv :2305.17352* (2023).