Étape 3: Robot-soccer sur Unity et MPO

Léo Chartrand 20 119 818
Jeremi Levesque 20 063 066
Marc-Antoine Bélisle 20 042 188

IFT608 - Planification en intelligence artificielle
Département d'informatique
Université de Sherbrooke

https://github.com/leochartrand/ml-agents

Abstract

Dans cette dernière étape, nous explorons un algorithme d'apprentissage *Policy Gradients* nommé Maximum a Posteriori Policy Optimization (MPO) qui surpasse l'algorithme de pointe PPO dans plusieurs environnements complexes [1]. Nous avons appliqué cet algorithme à l'environnement *SoccerTwos* de Unity ML Agents afin de pouvoir comparer la performance de notre implémentation par rapport aux algorithmes des étapes précédentes. Bien que la performance de l'agent MPO ne soit pas celle attendue, nous avons identifié plusieurs points à potentiellement prendre en considération qui pourraient expliquer une performance nettement inférieure à celle de PPO.

1 Introduction

Le soccer est un environnement de coalition où deux équipes d'agents s'affrontent, ce qui le rend particulièrement intéressant pour l'entraînement d'agents qui apprennent à coopérer et à développer une stratégie pour vaincre l'autre équipe. Ce sujet de recherche a des applications ludiques, mais principalement aussi des applications militaires et même de développement d'exosquelettes pour le déplacement de personnes handicapées [2]. L'environnement utilisé jusqu'à présent, *SoccerTwos* de ML Agents Toolkit, est un environnement idéal au développement de ce type d'agents puisqu'il représente cet environnement de manière simple, facilement manipulable et clairement interfacée pour le développement de nouveaux agents. ML Agents fournit quelques algorithmes déjà implémentés que nous avons utilisés et adaptés à notre environnement de soccer afin de les comparer dans les étapes précédentes.

Nous avons analysé l'implémentation et les résultats de l'algorithme de *Proximal Policy Optimization* (PPO) [3] dans les étapes précédentes et les résultats obtenus étaient comparables à l'algorithme multiagent POCA et ce qui donnait une équipe d'agent plutôt performante en deux contre deux. Le but de cette troisième et dernière étape du projet était de se diriger vers un autre algorithme de pointe plus performant que PPO afin de l'implémenter et d'analyser sa performance dans un contexte multiagent par rapport aux algorithmes des étapes précédentes. Muesli [4] nous semblait être un algorithme prometteur puisque c'est un algorithme qui a des performances similaires à celles de MuZero [5] qui a démontré des performances remarquables en apprentissage de politiques pour des environnements de grande complexité. Cet algorithme semblait être relativement complexe et risqué à implémenter dans le temps alloué pour cette dernière étape, donc nous avons décidé d'implémenter et d'analyser l'algorithme *Maximal a Posteriori Policy Optimization* (MPO) [1] qui est un des algorithmes qui sont utilisés comme fondation théorique derrière l'implémentation de Muesli et qui démontre tout de même, dans plusieurs environnements, des performances excédant celles de PPO.

2 Travaux Reliés

2.1 Méthodes

Alors que les algorithmes de *Policy Optimization* à région de confiance comme TRPO[6] et PPO[3] sont faciles à ajuster, offrent une réduction de la variance des gradients ainsi qu'une amélioration monotone robuste, ceux-ci sont par définition *on-policy* et sont donc peu efficaces quant à l'usage des échantillons. Ce défaut réduit d'ailleurs leur utilité pour les applications réelles comme en robotique où la génération d'échantillons peut être coûteuse.

À l'inverse, des algorithmes de type *value gradient* tels que DQN[7] et DDPG[8] sont *off-policy* et utilisent l'*experience replay*[9] pour réutiliser les expériences échantillonnées par les agents, faisant un usage des données beaucoup plus efficaces. Cette particularité rend d'ailleurs ces techniques beaucoup plus viables pour l'entraînement de système robotique[10]. Cependant, la complexité de l'environnement est proportionnelle à la difficulté de paramétrage de tels algorithmes ce qui les rend généralement peu efficaces des scénarios de haute dimensionnalité.

2.2 Maximum a Posteriori Policy Optimisation (MPO)

En 2018, une équipe de chercheurs chez DeepMind[1] a proposé un nouvel algorithme, appelé *Maximum a Posteriori Policy Optimisation* (MPO), qui tire profit des qualités de ces deux familles d'algorithmes. Cette technique se rapproche d'un algorithme espérance-maximisation, ce dernier pouvant être utilisé pour trouver des solutions Maximum a Posteriori pour des problèmes dont les *a prioris* sont paramétrés [11]. Sachant une trajectoire $\tau = \{(s_0, a_0)...(s_T, a_T)\}$ échantillonné par la politique π, on pose la formulation $p_\pi(\tau) = p(s_0) \prod_{t>0} p(s_{t+1}|s_t, a_t)\pi(a_t|s_t)$ ainsi que le retour $\mathbb{E}_\tau \left[\sum_{t>0} \gamma^t r_t\right]$. Le problème d'apprentissage peut alors être posé comme un problème d'inférence variationnelle pour lequel une limite inférieure sur la vraisemblance de l'optimalité de π peut être définie à l'aide de la distribution auxiliaire q[1], telle que $q_\pi(\tau) = p(s_0) \prod_{t>0} p(s_{t+1}|s_t, a_t)q(a_t|s_t)$, ce qui donne cet objectif:

$$\mathcal{J}(q, \theta) = \mathbb{E}_q \left[\sum_{t=0}^{\infty} \gamma^t \left[r_t - \alpha \mathrm{KL} \left(q(a|s_t) || \pi(a|s_t, \theta) \right) \right] \right] + \log p(\theta).$$

L'addition du terme *a priori* $\log p(\theta)$ découle d'une formulation du problème comme une estimation de maximum a-posteriori et sert de terme régularisateur. L'algorithme EM alterne entre une optimisation de deux distributions q et π. Ainsi, l'objectif $\mathcal{J}$ est maximisé lors de chaque étape en fonction de leur distribution respective.

L'étape E consiste à évaluer l'espérance de la vraisemblance de l'optimalité de π en maximisant $\mathcal{J}$ selon la distribution q. Compte tenu de la divergence KL, cela revient à utiliser $q(\tau)$ pour approximer la distribution réelle des trajectoires τ générées par π. La fonction action-valeur $Q_\theta(s, a)$ est estimée à partir d'échantillons Monte-Carlo de manière *off-policy* via l'*experience replay*, et ensuite régularisée telle qu'on peut reformuler l'objectif comme suit:

$$\begin{aligned} \underset{q}{\text{maximiser}} \;\; \bar{\mathcal{J}}_s(q, \theta) = \underset{q}{\text{maximiser}} \;\; \hat{\mathbb{E}} \left[\mathbb{E}_{q(a|s)} \left[Q_\theta(s, a) \right] \right] \\ \text{sujet à} \;\; \hat{\mathbb{E}} \left[\mathrm{KL}(q(a|s), \pi(a|s, \theta)) \right] < \epsilon. \end{aligned}$$

La formule énoncée ci-haut rappelle la fonction objectif subrogée de TRPO[6], la différence étant qu'au lieu de comparer la politique courante avec celle d'une itération précédente afin de former une limite supérieure quant à l'ascente de gradient, la divergence KL est inversée pour poser une limite inférieure en fonction de q. En somme, étant donné l'objectif régularisé et un *batch* d'expériences échantillonnées, on obtient $q = \text{argmax}\bar{\mathcal{J}}(q, \theta)$.

L'étape M consiste ensuite à optimiser $\mathcal{J}$ en fonction de θ, ce qui donne l'objectif suivant:

$$\mathcal{J}(q, \theta) = \mathbb{E}_q \left[\mathbb{E}_q(a|s) \left[\log \pi(a|s, \theta) \right] \right] + \log p(\theta),$$

[1] À noter ici que q n'a rien à voir avec les Q-valeurs.

où on cherche à maximiser θ. Ce problème se réduit à résoudre

$$\begin{aligned} \underset{\theta}{\text{maximiser}} \ & \hat{\mathbb{E}}\left[\mathbb{E}_{q(a|s)}\left[\log \pi(a|s,\theta)\right]\right] \\ \text{sujet à} \ & \hat{\mathbb{E}}\left[\mathrm{KL}(\pi(a|s,\theta_t), \pi(a|s,\theta))\right] < \epsilon. \end{aligned}$$

Les auteurs notent qu'en pratique, la divergence KL aide à stabiliser l'apprentissage. C'est donc cette étape qui porte concrètement sur l'optimisation de la politique π.

2.3 Relative Entropy Regularized Policy Iteration

L'article de *Relative Entropy Regularized Policy Iteration* [12] détaille un algorithme d'apprentissage *actor-critic* qui est une extension de l'algorithme MPO par les mêmes auteurs. Cet algorithme peut aussi être vu comme une extension de CMA-ES qui est un algorithme que nous ne détaillerons pas dans cette étape. Cette extension implémente MPO de manière légèrement plus simple qui offre une liberté d'implémentation différente pour chaque étape. MPO étendu se résume essentiellement en 3 étapes importantes: 1) l'évaluation de la politique en estimant paramétriquement une valeur d'état-action 2) amélioration de la politique en estimant une politique locale non paramétrée 3) généralisation en adaptant une politique paramétrée.

2.3.1 Étape 1: Évaluation de la politique

Cette étape concerne principalement l'approximation d'une fonction Q qui permet d'évaluer la politique. Plusieurs implémentations d'algorithmes d'apprentissages d'une fonction Q peuvent être utilisées dans cette étape pourvu que la valeur de la politique approximée soit appropriée et relativement précise dans le contexte d'utilisation. Généralement, on minimise la différence temporelle au carrée en utilisant deux fonctions paramétriques: une fonction dont les paramètres sont modifiés au cours de l'apprentissage et l'autre qui est une copie de la première dont les paramètres sont fixés et qui recopie les paramètres de la première fonction à un nombre d'itérations donné.

$$\arg\min_{\phi}(r_t + \gamma Q_{\phi'}^{(k-1)}(s_{t+1}, a_{t+1}) - Q_{\phi}^{k})^2$$

Q_ϕ dénote la fonction paramétrique alors que $Q_{\phi'}$ dénote la fonction d'estimation dont les paramètres sont fixés à un nombre d'itérations donné.

2.3.2 Étape 2: Recherche des poids d'actions

Le but dans cette étape est de réajuster les probabilités des actions de sorte que les actions permettant de maximiser la somme espérée de toutes les actions dans chaque état. On veut ainsi augmenter la probabilité qu'une bonne action soit choisie. On échantillonne du *replay buffer* une série d'états ainsi qu'une quantité N d'actions dans chaque état. En calculant les valeurs Q pour chaque paire état-action, on peut ainsi augmenter les probabilités des actions résultant en une valeur Q plus élevée. Ces probabilités modifiées sont dénotées q_{ij} où i est l'action dans l'état j et elles forment la politique non paramétrique améliorée d'un échantillon du *replay buffer*. Comme l'étape précédente de l'algorithme, il est possible d'implémenter presque n'importe quelle méthode permettant de calculer les poids des actions en utilisant des transformations qui conservent le rang des valeurs Q pourvu que les probabilités des actions pour chaque état soient positives et somment à 1.

Dans notre implémentation, nous effectuons une recherche des poids d'actions en utilisant une transformation exponentielle des valeurs Q. On peut obtenir les points en optimisant directement l'assignation des probabilités des actions, mais il serait souhaitable de contraindre le changement en résolvant l'objectif de régularisation KL suivant:

$$\begin{aligned} q_{ij} &= \text{argmax}_{q(a_i|s_j)} \sum_{j}^{K} \sum_{i}^{N} q(a_i|s_j) Q^{\pi^{(k)}}(s_j, a_i) \\ \text{sujet à} \ & \frac{1}{K} \sum_{j}^{K} \sum_{i}^{N} q(a_i|s_j) \log \frac{q(a_i|s_j)}{1/N} < \epsilon, \quad \forall_j \sum_{i}^{N} q(a_i|s_j) = 1. \end{aligned}$$

La première contrainte est celle qui permet de limiter l'entropie relative moyenne, et donc l'update maximale qui peut être faite sur la distribution de probabilités. Si une action a une valeur très élevée, on pourrait être tenté de faire une mise à jour pour mettre la plus grande probabilité possible de sorte que seulement cette action soit prise dans cet état. Cependant, à cause notamment du dilemme d'exploitation exploration, on contraint la mise à jour des poids pour effectuer des pas pessimistes vers une solution. Enfin, la deuxième contrainte nécessite que toutes les valeurs Q pour chaque action i dans un état j somment à 1 afin d'être utilisées comme probabilités pour les nouveaux paramètres. Bien que cette équation semble complexe à implémenter, sa solution est obtenue en *closed form* en effectuant le softmax des valeurs Q:

$$q_{ij} = q(a_i, s_j) = \frac{\exp\left(Q^{\pi}(s_j, a_i)/\eta\right)}{\sum_i \exp(Q^{\pi(k)}(s_j, a_i)/\eta)}$$

où la valeur du paramètre de température η peut être trouvé en effectuant quelques itérations de descente du gradient:

$$\eta = \text{argmin}_{\eta} \eta\epsilon + \eta \sum_{j}^{K} \frac{1}{K} \log \left(\sum_{i}^{N} \frac{1}{N} \exp\left(\frac{Q(s_j, a_i)}{\eta} \right) \right).$$

2.3.3 Étape 3: Amélioration de la politique

Pour améliorer la politique, nous résolvons un problème d'apprentissage supervisé pondéré afin de généraliser les valeurs de notre échantillon sur l'espace d'état et d'action. Afin d'éviter un surapprentissage sur l'échantillon de l'étape 2, on impose une contrainte de divergence additionnelle qui peut être résolue en transformant à l'aide de la Relaxation Lagrangienne ce qui donne l'équation suivante:

$$\max_{\theta} \min_{\alpha>0} L(\theta, \eta) = \sum_j \sum_i q_{ij} \log \pi_\theta(a_i|s_j) + \alpha \left(\epsilon_\pi - \sum_j^K \frac{1}{K} \text{KL}\left(\pi^{(k)}(a|s_j) || \pi_\theta(a|s_j) \right) \right)$$

En pratique, il suffit de faire une itération de descente du gradient tant pour la maximisation (maximisation extérieure) de θ que pour l'optimisation des multiplicateurs de Lagrange (minimisation intérieure) sur chaque échantillon de données afin de satisfaire les contraintes durant tout l'entraînement.

3 Méthodologie

L'évaluation de l'algorithme MPO a été réalisée dans le même environnement que lors de la dernière étape, c'est-à-dire l'environnement *SoccerTwos* offert par le projet ML-Agents de Unity dans la *Release 20*. Afin de pouvoir comparer les performances de MPO avec les méthodes de référence, nous avons utilisé les mêmes paramètres d'environnement, les mêmes récompenses et le même $seed = 5$. Tout comme à la dernière étape, nous avons utilisé le module Tensorboard pour pouvoir générer les courbes des graphiques utilisés pour la comparaison des algorithmes. Les résultats obtenus lors de cette évaluation sont disponibles sur le GitHub du projet.

3.1 Méthodes de référence

Pour évaluer les performances de MPO, nous avons comparé cet algorithme aux performances de POCA, PPO et A2C. Nous n'avons pas inclus l'algorithme DQN, présenté lors de la dernière étape dans cette comparaison, car leurs performances étaient nettement inférieures à celles de MA-POCA et PPO et le temps d'entraînement pour DQN était trop grand pour les ressources disponibles. Comme lors des étapes précédentes, nous avons utilisé la métrique de performance Elo pour évaluer les différents algorithmes.

Dans l'environnement ML-Agents, l'algorithme MPO n'était pas implémenté. Nous avons donc procédé à son implémentation en utilisant le module *ml-agents-trainer-plugin*, qui permet d'intégrer

notre propre algorithme au sein de l'environnement. Pour ce faire, nous nous sommes inspirés de l'implémentation de daisatojp[2], basée sur l'article "Relative Entropy Regularized Policy Iteration"[12]. Nous avons également utilisé les mêmes hyperparamètres que ceux proposés par daisatojp, compte tenu de la complexité de notre environnement et du temps d'entraînement nécessaire pour rechercher les meilleurs hyperparamètres. De plus, l'algorithme en lui-même ne devrait pas nécessiter une grande recherche d'hyperparamètres puisqu'il est prévu pour donner de bonnes performances avec une grande étendue d'hyperparamètres.

L'architecture des critiques paramétrés est essentiellement la même que celle implémentée pour DQN lors de l'étape 2, permettant de supporter plusieurs branches ayant chacune leur propre espace d'actions discret. Cette restructuration consiste à l'ajout de la classe `BranchValueNetwork` dans la librairie ML-Agents, allant plus loin que le simple ajout d'algorithme à l'aide du *ml-agents-trainer-plugin*. Par ailleurs, plusieurs autres modifications ont du être apportées pour pouvoir généraliser au-delà des implémentations PPO et POCA fournies par défaut.

Étant donné que les algorithmes DQN et A2C ont nécessité un temps d'entraînement considérable lors des étapes précédentes, nous avons décidé de louer un serveur chez Google Cloud, équipé d'une carte graphique Tesla V100-SXM2-16GB, afin d'accélérer le processus d'entraînement pour l'algorithme MPO.

4 Expérimentations

Durant la majorité de notre expérimentation, nous en avons profité pour continuer l'entraînement de l'algorithme A2C qui a été présenté et comparé à l'étape 2 afin de pouvoir présenter une mesure comparative plus représentative avec un entraînement "complet" de 50 millions d'étapes. L'entraînement de MPO dans l'environnement de soccer nécessite un temps d'entraînement moyen supérieur aux autres algorithmes pour un même nombre d'étapes donné. En guise de comparaison, nous avons atteint 500 000 étapes après 4,1 heures d'entraînement alors qu'A2C, qui était considéré significativement plus lent que PPO et POCA, avait atteint un même nombre d'étapes en 0,75 heure (voir le Tableau 1).

Temps d'entraînement nécessaire pour 500 000 étapes				
MA-POCA	PPO	A2C	DQN	**MPO**
$\sim$ 11 min	$\sim$ 7 min	$\sim$ 45 min	$\sim$ 120 min	$\sim$ **240 min**

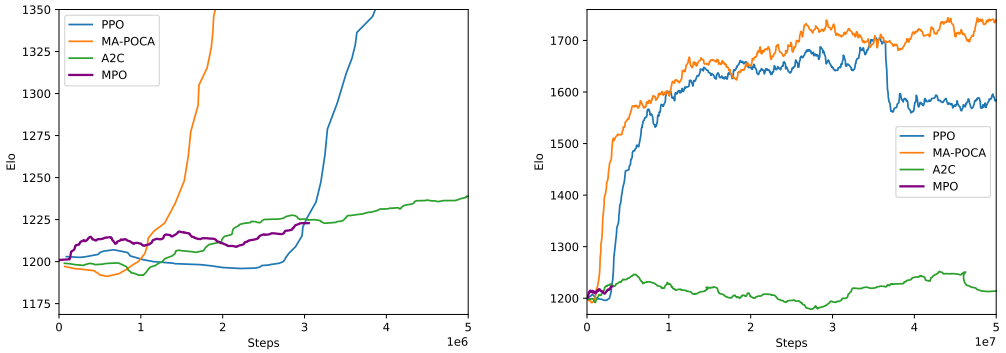
Tableau 1: Temps d'entraînement comparatif des 500 000 premières étapes d'entraînement seulement. L'entraînement total résultant de 50 millions d'étapes (sauf pour MPO qui n'a pas atteint ce nombre d'étapes dans le temps alloué) est similaire en terme de proportion de différence de temps entre les algorithmes. MPO a été entraîné sur un serveur de calcul avec GPU, donc, si l'entraînement avait été fait sur le même matériel que les autres algorithmes, la différence de temps d'entraînement aurait été potentiellement plus grande.

4.1 Résultats

Après 24 heures consécutives sur près de 3 millions d'étapes d'entraînement effectuées, nous obtenons des agents MPO qui ont un Elo final de 1223 ainsi qu'un Elo maximal de 1225. Selon notre mesure de performance, on peut voir que l'algorithme semble performer mieux que tous les autres algorithmes durant le premier million d'étapes d'entraînement, et demeure supérieur à A2C durant tout l'entraînement. Cependant, les performances de PPO et MA-POCA viennent rapidement excéder les capacités de l'agent MPO après 3M et 1M d'étapes respectivement.

Lorsqu'on analyse le comportement des différents agents en action dans l'environnement, on peut tout d'abord rapidement voir que PPO et MA-POCA ont bel et bien appris une politique qui semble être raisonnable. On décrit une politique comme étant raisonnable si elle résulte en une stratégie qui permet de marquer des buts sans que ça soit par un mouvement aléatoire du joueur. Bien que MPO semble avoir un elo supérieur à A2C en moyenne, le comportement des agents en action semble être

[2]https://github.com/daisatojp/mpo



(a) Échelle restreinte à 5 millions d'étapes.

(b) Échelle complète de 50 millions d'étapes.

Figure 1: Perspectives de l'apprentissage de MPO durant son entraînement en comparaison avec les algorithmes des étapes précédentes.

un comportement aléatoire qui ne semble pas raisonnable pour un contexte de soccer. Les joueurs ne semblent pas être incités à entrer en contact avec le ballon et certains se retrouvent fréquemment coincés dans le coin d'un des buts, ce qui n'est certainement pas des comportements désirés au soccer.

Comme nous n'avons pas effectué le même nombre d'étapes que PPO et MA-POCA, la comparaison des résultats reste relativement incertaine, puisque comme l'entraînement de l'agent MPO se limite à 3M d'étapes par faute de temps, un apprentissage non négligeable est toujours possible. Par contre, nous suspectons que même avec un entraînement complet, l'amélioration des performances au fil de l'entraînement serait similaire à A2C puisque dans certains entraînements précédents, les agents développent une politique qui ne leur permettent pas de récolter des récompenses (i.e. restent coincé dans le but ou dans le coin de l'aire de jeu). Ainsi, nous avons essayé d'identifier des goulots d'étranglement dans notre implémentation afin d'accélérer l'entraînement des agents, mais les quelques parties identifiées et modifiées n'ont pas vraiment accéléré l'exécution de l'entraînement de notre agent au final.

Comme expliqué dans les deux premières étapes, MA-POCA est un algorithme qui semble avoir été développé spécialement pour un contexte multiagent et son intégration à ML Agents a été plusieurs fois optimisées puisque c'est l'algorithme de démonstration principal pour cet environnement. On ne s'attendait donc pas à ce que MPO obtienne des résultats supérieurs à MA-POCA. Cependant, les résultats obtenus ne sont pas ceux attendus puisque nous étions à la recherche d'un algorithme qui aurait été capable de surpasser PPO par rapport à notre mesure de performance. Nous avons révisé notre implémentation de MPO, les paramètres de l'environnement ainsi que les hyperparamètres de l'entraînement plusieurs fois, mais notre agent MPO n'augmentait à peine son Elo moyen de quelques unités.

4.2 Risques et points faibles

Selon l'article de référence [1], MPO est particulièrement adapté pour des environnements de contrôle continus. C'est un algorithme qui n'a pas atteint l'état de l'art en termes de performances sur des environnements à domaine d'action discret, mais qui demeure compétitif en réutilisant les mêmes paramètres que pour un espace continu. Ça pourrait donc, être possiblement une cause de la faible performance de notre agent MPO, puisque non seulement l'environnement possède un espace d'action discret, mais nous mettons à l'épreuve l'agent dans un contexte multiagent avec des récompenses clairsemées, ce qui le met particulièrement à l'épreuve.

De plus, bien que l'environnement de Unity ML Agents est facile d'utilisation, ce n'est peut-être pas le meilleur environnement de développement d'algorithmes pour un court projet de soccer. L'implémentation du module d'entraînement d'agents est assez complexe, sa documentation n'est pas exhaustive et détaille peu la hiérarchie des définitions d'objets dans l'interface de ML Agents. Cela rend l'implémentation d'algorithmes particulièrement complexe puisque toutes les parties du module doivent être maîtrisées afin de bien comprendre ce qui se passe dans le code et la courbe de

compréhension de cette implémentation est abrupte. De plus, l'environnement de simulation pourrait être simplifié, ce qui pourrait potentiellement accélérer légèrement le temps d'entraînement. Un environnement en 2 dimensions pour modéliser cet environnement de soccer suffirait dans ce contexte au lieu d'avoir une modélisation 3D non nécessaire à l'apprentissage des agents.

En ce qui concerne la modification des hyperparamètres, il est important de souligner que la recherche des meilleurs hyperparamètres est un processus long et fastidieux. En effet, puisqu'un entraînement nécessite généralement 2 à 3 millions d'étapes pour déterminer si l'algorithme est performant ou non, cela engendre un temps considérable avant de pouvoir évaluer la satisfiabilité des hyperparamètres utilisés. Ce facteur peut expliquer en partie pourquoi nous n'avons pas été en mesure d'obtenir un agent avec de bonnes performances puisque notre exploration des hyperparamètres fût limitée par le temps alloué pour ce projet.

Un autre problème auquel nous avons fait face est que les grandes lignes de notre reproduction de MPO sont basées sur l'implémentation de l'utilisateur GitHub daisatojp[3], qui au préalable nous a sauvé beaucoup de temps étant donné l'opacité des articles par (Abdolmaleki *et al.*, 2018a[1], Abdolmaleki *et al.*, 2018b[12]) quant à une implémentation concrète de MPO. Cependant ce code de référence contient des erreurs que nous avons remarquées trop tard, soit entre autres l'usage d'une étape M alors que q est paramétrée. Cette erreur nous a poussés à utiliser plus de réseaux neuronaux qu'il en était nécessaire en plus d'ajouter des étapes excessives à l'algorithme d'optimisation, ce qui est en grande partie la source de la lenteur d'apprentissage.

La distribution actuelle des récompenses dans notre environnement pourrait être améliorée pour favoriser l'apprentissage des agents et ainsi augmenter leurs performances. Pour l'instant, les récompenses sont attribuées de la manière suivante: lorsqu'une équipe marque un but, elle reçoit une récompense de (1 - temps écoulé depuis le début de la partie) et l'autre équipe reçoit une récompense de -1. De plus, à la fin de chaque épisode, chaque agent reçoit des récompenses basées sur le nombre de fois qu'il a touché le ballon, soit 0,2 * nombre de contacts avec le ballon. Il est important de souligner que nous n'avons pas modifié les récompenses de base et que celles-ci sont conçues pour être adaptées à l'algorithme MA-POCA et non pour MPO. Cette différence peut avoir un impact sur les performances de notre agent MPO, car les récompenses optimales pour l'un ne sont pas nécessairement les meilleures pour l'autre. Il serait donc pertinent d'explorer des ajustements aux récompenses pour mieux les adapter à l'algorithme MPO et potentiellement améliorer les performances de notre agent dans cet environnement.

Il est possible que cette distribution des récompenses ne soit pas optimale pour encourager les comportements souhaités chez les agents. Afin d'améliorer la distribution des récompenses, plusieurs approches pourraient être envisagées:

- Récompenses intermédiaires: Plutôt que de récompenser uniquement les buts et le nombre de contacts avec le ballon, on pourrait introduire des récompenses intermédiaires pour des actions spécifiques, telles que des passes réussies, des interceptions ou des tirs cadrés. Ces récompenses pourraient encourager des comportements plus coopératifs et stratégiques entre les agents.
- Récompenses basées sur la proximité: On pourrait également envisager d'attribuer des récompenses en fonction de la proximité du ballon par rapport au but adverse. Cela encouragerait les agents à progresser vers le but adverse et à maintenir la possession du ballon dans des zones stratégiques du terrain.
- Récompenses différenciées pour les rôles: Il serait également possible d'attribuer des récompenses spécifiques en fonction des rôles des agents sur le terrain (attaquants, défenseurs, gardiens de but). Par exemple, les défenseurs pourraient recevoir des récompenses pour les interceptions réussies, tandis que les attaquants pourraient être récompensés pour les occasions de but créées.

5 Conclusion

En conclusion, nous avons exploré l'implémentation et l'analyse de l'algorithme Maximal a Posteriori Policy Optimization (MPO) dans l'environnement de ML agent. Bien que cet algorithme ait

[3]https://github.com/daisatojp/mpo

montré des performances prometteuses dans d'autres environnements, il n'a pas réussi à surpasser les performances de l'algorithme PPO. Malgré le peu d'entrainement, les agents MPO ont affiché un comportement apparemment aléatoire, ne parvenant pas à développer une stratégie de jeu efficace. Bien que des améliorations potentielles pourraient être apportées à l'implémentation et aux hyperparamètres de l'algorithme, les résultats obtenus suggèrent que MPO n'est pas l'algorithme le plus adapté pour ce contexte spécifique.

À l'avenir, il pourrait être intéressant d'explorer plus en détail les particularités de MPO. Par exemple, il serait possible d'utiliser une fonction de *clip* pour substituer la divergence KL comme de TRPO à PPO, permettant de simplifier l'implémentation et possiblement accroître la performance. C'est ce que fais d'ailleurs Muesli[4] par DeepMind, qui performe compétitivement dans plusieurs environnements et rivalise même MuZero[13]. Il serait alors pertinent d'implémenter un tel algorithme pour évaluer son potentiel dans le domaine du soccer multiagent.

Finalement, une perspective future porte sur l'utilisation du *framework* ML-Agents quant à l'implémentation de nouveaux algorithmes et la pertinence de la plateforme pour le développement RL. Alors que d'autres logiciels comme MuJoCo[14] sont plus adaptés pour le développement d'algorithmes de contrôle et sont davantage reconnus par la communauté, ML-Agents se démarque par son accessibilité et la facilité d'utilisation permettant de créer de nouveaux environnements. L'exploitation de Unity comme moteur de jeu permet de créer des tâches complexes, pouvant d'ailleurs inclure des dynamiques de coopération et compétition pour des systèmes multi-agents. Selon nos observations, le module *ml-agents-trainer-plugin* semble être spécifiquement adapté pour PPO et POCA. Alors qu'il s'agit d'un travail en cours *open-source*, il serait pertinent d'apporter des améliorations à la plateforme pour faciliter l'implémentation d'une multitude d'algorithmes de natures distinctes, permettant de faciliter la tâche pour de prochains projets de ce genre ainsi que de contribuer à la communauté.

References

[1] Abbas Abdolmaleki et al. "Maximum a posteriori policy optimisation". In: *arXiv preprint arXiv:1806.06920* (2018).

[2] Luiz A Celiberto Junior and Jackson Paul Matsuura. "Robotic Soccer: the Gateway for Powerful Robotic Applications." In: *ICINCO-RA (2)*. 2008, pp. 287–293.

[3] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[4] Matteo Hessel et al. "Muesli: Combining improvements in policy optimization". In: *International conference on machine learning*. PMLR. 2021, pp. 4214–4226.

[5] Julian Schrittwieser et al. "Mastering Atari, Go, chess and shogi by planning with a learned model". In: *Nature* 588 (2020), pp. 604–609.

[6] John Schulman et al. *Trust Region Policy Optimization*. 2017. arXiv: 1502.05477 [cs.LG].

[7] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518 (2015), pp. 529–533.

[8] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. 2019. arXiv: 1509.02971 [cs.LG].

[9] Longxin Lin. "Self-improving reactive agents based on reinforcement learning, planning and teaching". In: *Machine Learning* 8 (1992), pp. 293–321.

[10] Shixiang Gu et al. *Deep Reinforcement Learning for Robotic Manipulation with Asynchronous Off-Policy Updates*. 2016. arXiv: 1610.00633 [cs.RO].

[11] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[12] Abbas Abdolmaleki et al. "Relative entropy regularized policy iteration". In: *arXiv preprint arXiv:1812.02256* (2018).

[13] Julian Schrittwieser et al. "Mastering Atari, Go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (Dec. 2020), pp. 604–609. DOI: 10.1038/s41586-020-03051-4. URL: https://doi.org/10.1038%2Fs41586-020-03051-4.

[14] Emanuel Todorov, Tom Erez, and Yuval Tassa. "MuJoCo: A physics engine for model-based control". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.