1. Environnement :

Derk's Gym est un jeu de style MOBA (Multiplayer Online Battle Arena) dans lequel deux équipes de trois agents (nommés derklings) s'affrontent avec pour objectif la destruction de la ou des structures de l'équipe ennemie et des adversaires tout en essayant de défendre leurs propres structures et de survivre. Son implémentation est simplifiée pour permettre la définition d'un environnement de Reinforcement Learning facilement manipulable. La mise en place des méthodes d'apprentissage se fait par l'intermédiaire de "Derk's Gym", un packet python exposant l'API de l'environnement gym sur lequel le jeu a été construit. Cet environnement date des années 2020 et 2021 et a été créée pour servir de support aux compétitions de Reinforcement Learning et en particulier à une compétition de la communauté AI Crowd datant de février 2021. L'environnement est compatible avec l'entraînement sur GPU (ce qui est nécessaire pour espérer obtenir des résultats en temps corrects) et en parallèle (sur plusieurs arènes en même temps).

La visualisation et l'exécution se fait par une Webapp de type WebGL2 implémentée par tous les navigateurs les plus utilisés aujourd'hui. L'utilisation de cet environnement est possible sur toutes les plateformes et OS disponibles et en cas de problèmes, des versions dockerisées sont disponibles et offrent toutes les fonctionnalités de base à condition d'avoir accès au GPU de la machine hôte.

1.1 Arène et principes du jeu :

L'arène est composée de deux statues (une pour chaque équipe) sur laquelle les deux équipes de trois agents se déplacent en essayant de détruire la statue de l'équipe adverse. Le terrain est accidenté et les agents peuvent rester bloqués face à une colline. Plus précisément, chaque statue est surélevée, empêchant les agents de l'atteindre s'ils ne se déplacent qu'en ligne droite (et donc d'attaquer la tour de près s'ils n'ont pas de moyens de le faire à distance). L'arène étant en hauteur, les bords sont des précipices d'où les agents peuvent tomber et mourir (ce qui ajoute des points au score de l'équipe adverse). Il est important de noter que les bords sont irréguliers ce qui joue un rôle dans les premières phases d'apprentissage où les agents apprennent à éviter de tomber.

Au début de la partie chaque agent reçoit de manière aléatoire une, deux ou trois armes (et parfois aucune) qu'il équipera à ses bras, sa queue ou sa tête. Cela aura une conséquence lors de l'apprentissage, comme cela sera décrit dans la seconde partie du rapport dédiée à l'apprentissage et aux expériences réalisées. Dans les paramètres de base de l'environnement, chaque partie dure 20 secondes. **Le score du jeu est dissocié de la fonction de récompense**. Il est fixe et est défini tel que la destruction d'une tour apporte **13 points**, l'élimination d'un agent ennemi apporte **4 points**. **Il n'y a pas de score négatif et les équipes ne perdent pas de points.**

Parmi les armes à disposition de chaque agent, on peut lister :

Name	Slot	Description
Talons	arms	Melee item dealing good, steady damge to a target.
BloodClaws	arms	Damage dealing melee item that also heals the equipper with each hit.
Cleavers	arms	Heave and powerful, but slow hitting melee item.
Cripplers	arms	Melee item that also cripple the opponent, making them move slower.
Pistol	arms	Ranged weapon. Pew pew!
Magnum	arms	Heavy ranged weapon that knocks the target back.
Blaster	arms	Heavy ranged weapon that deals massive damge.
FrogLegs	misc	Long strong legs, enabling the Derkling to quickly jump forward.
IronBubblegum	misc	Blows an iron-enforced bubble around a target, protecting them from damage.
HeliumBubblegum	misc	Blows a bubble filled with helium around a target, making them float up into the air.
Shell	misc	Increases the armor of a Derkling. Armor is further increased when they duck.
Trombone	misc	When the horn is blown, all enemies are forced to focus on the musician.
HealingGland	tail	Siphons hitpoints to the target.
VampireGland	tail	Drains a target of hitpoints and restores the casters hitpoints.
ParalyzingDart	tail	Launches a projectile at a target, dazing them for a short moment.

1.2 API exposée:

L'API exposée par "Derk's Gym" est similaire à toutes les API disponibles sur l'environnement gym de base. La principale différence est l'ajout de nombreux paramètres supplémentaires tels que les attributs associés aux agents (couleur, armes prédéfinies, fonction de reward spécifique, attributs physiques uniquement esthétiques ...). On retrouve donc une boucle d'apprentissage classique avec entre autre:

- un espace d'observation comportant **64** observations possibles dont les principales sont :
    - la capacité à agir : `Ability0Ready`

    - la distance et/ou l'angle à chacun des alliés ou de ses ennemis ou d'une statue (si l'observation est possible) `FriendStatueDistance` , `FriendStatueAngle`, `Friend1Distance` , `Friend1Angle` ...
    - **savoir s'il possède une des armes citées plus haut : `HasBloodClaws`, `HasCleavers` ...**
    - savoir si son point d'intérêt possède une des armes citées : `FocusHasBloodClaws`, `FocusHasCleavers`
    - si l'agent a un point d'intérêt : `HasFocus`
    - s'il est bloqué dans son mouvement : `Stuck`
    - ...
- un espace d'action :
    - `MoveX` : [-1;1] pour se déplacer en avant ou en arrière
    - `Rotate` : [-1;1] pour tourner à gauche ou à droite
    - `ChaseFocus` : 0 ou 1. Si égal à 1, l'agent se dirige vers son point d'intérêt. Des valeurs inférieures à 1 font un mix entre le résultat des deux paramètres précédents et de la direction de sont point d'intérêt (POI).
    - `CastingSlot` : 0 aucune arme utilisée; 1 à 3, l'agent utilise une de ses 3 armes à disposition (s'il en est équipé, sinon le résultat est équivalent à 0).
    - `ChangeFocus` : si 0, l'agent garde son POI, si 1, il fixe pour POI la statue de son équipe. Si 2 ou 3, son POI devient son équipier. Si 4, le POI est la statue ennemie et si 5 ou 7, le POI est un des deux ennemis. **Un agent peut avoir plusieurs observations (flèches blanches sur les vidéos de démonstration) mais un seul POI vers lequel il se dirigera et au niveau duquel il pourra potentiellement effectuer une action**.
- une fonction `step` (lien doc) prenant une liste d'actions en entrée (une pour chaque agent) et retourne le nouvel état.
- un logger de statistiques `team_stats` utilisé dans ce projet pour afficher les données et courbes d'intérêt sur `tensorboard` et regroupant les statistiques de chacune des deux équipes à l'épisode précédent.
- une propriété `total_reward` retournant toutes les récompenses accumulées au cours d'un épisode pour chaque agent.

Une API bas niveau est aussi disponible mais elle n'est utile que dans le cas où un agent est défini en tant que service sur un serveur ou dans le cas où plusieurs agents sont exécutés sur des machines différentes. Ce projet ne fait pas usage de ces fonctionnalités et il n'est pas nécessaire de les décrire en détail.

L'API est très riche et la documentation est détaillée. Je préfère ne dérouler que ce que j'ai utilisé afin de ne pas perdre en pertinence.

1.3 Fonction de récompense :

Field	Default value	Notes
damageEnemyStatue	0	par point de dégât
damageEnemyUnit	0	par point de dégât et par ennemi
killEnemyStatue	4	lors de la destruction de la statue ennemie
killEnemyUnit	1	lors de l'élimination d'un ennemi
healFriendlyStatue	0	par point de soin offert à la statue alliée
healTeammate1	0	par point de soin offert à l'agent allié

Field	Default value	Notes
healTeammate2	0	par point de soin offert à l'agent allié
timeSpentHomeBase	0	Mesuré toutes les 5 secondes
timeSpentHomeTerritory	0	Mesuré toutes les 5 secondes
timeSpentAwayTerritory	0	Mesuré toutes les 5 secondes
timeSpentAwayBase	0	Mesuré toutes les 5 secondes
damageTaken	0	par point de dégât subi par l'agent
friendlyFire	0	par point de dégât infligé à un allié
healEnemy	0	par point de soin offert à l'agent ennemi
fallDamageTaken	0	contribution unique lors du décès par chute de l'agent
statueDamageTaken	0	par point de dégât subi par la statue alliée (comptée pour tous les agents d'une équipe)
teamSpirit	0	Si égal à 1, toutes les récompenses sont moyennées pour tous les agents d'une équipe
timeScaling	1	Si 0, la récompense à l'étape précédente (exécution de la fonction `step`) n'est pas prise en compte

Les paramètres cités se différencient par leur mesure au cours du temps, le type d'action effectuées et de leurs résultats. Infliger des dommages à un agent ennemi équipé d'une Shell ne donnera pas la même récompense que s'il n'était pas protégé. Certains de ces paramètres offrent des **récompenses/pénalités par agent** (damageEnnemyStatue, damageTaken, fallDamagetaken, ...) tandis que d'autres induisent des **récompenses/pénalités pour tous les membres de l'équipe**. Le paramètre **teamSpirit** peut être intéressant car il permet de mettre en commun les récompenses des différents agents d'une équipe et d'induire des comportements plus homogènes (en particulier pour des stratégies défensives, comme cela sera montré dans la seconde partie).

1.4 Setup et exemple de boucle d'apprentissage :

Une classe `Agent` peut être redéfinie et enrichie en héritant ou en incluant une référence à la classe `derk3.agent.Agent` de base.

Un exemple de boucle d'apprentissage / exécution est défini ci-dessous :

```
with tqdm(range(config.episodes), total=config.episodes) as progress:
        for episode in progress:            ## iteration sur le nombre d'épisodes
            observation = env.reset()       ## récupération des observations
            trainer.reset()                 ## initialisation du module d'entraînement

            observations = [observation]    ## on met les observations dans une liste
            actions = []                    ## liste des actions choisies
            log_likelihoods = []            ## log de vraissemblance
            rewards = []                    ## récompenses lors de cet épisode

            while True:
                action, log_likelihood = trainer.act(observation)   ## récupération des
actions à réaliser
```

```
                observation, reward, done, _ = env.step(action)     ## réalisation des
actions

                observations.append(observation)                    ## màj des
observations
                actions.append(action)                  ## on garde les actions,
log_like et
                log_likelihoods.append(log_likelihood)  ## récompenses en mémoire
                rewards.append(reward)

                if all(done):                           ## stop si le critère d'arrêt
est atteint
                    break                               ## (20 secondes par défaut)
                ....
```

C'est une boucle gym classique avec `env`, une variable de type `DerkEnv` représentant l'environnement qui comporte les arènes et agents du processus d'entraînement ou de test et gérant l'affichage qui peut être annulé avec l'option `turbo_mode` à `True` .

1.5 Évaluation de l'environnement :

Derk's Gym est un environnement riche, rapide à utiliser, permettant une grande liberté dans la définition des algorithmes d'apprentissage tout en offrant de nombreux paramètres accessibles. Cependant il est important de noter certains bugs et effets indésirables dans la manière dont sont prises en compte certaines contributions à la fonction de récompense.

Lorsque deux agents accomplissent une action avec le même effet : endommager la statue ennemie et la détruire par exemple, ils auront tous les deux les récompenses liées aux dégâts effectués et la destruction de la statue. On peut remarquer la même chose lorsqu'il s'agit de soigner le même allié ou d'attaquer la même cible.

En conclusion on peut résumer les aspects positifs :

- Facile d'utilisation et documentation complète
- Multi-plateforme et adapté une large gamme de GPU
- Parallélisé pour l'entraînement
- Fonction de récompense prédéfinie complète et modifiable
- Visualisation peu gourmande en ressources et manipulable (stop, pause et affichage ou non des arènes)

Et les aspects négatifs:

- Problèmes de dédoublement dans la fonction de récompense
- Une très grande diversité d'observations (ce qui peut être un avantage pour certains mais difficile à gérer)
- L'aspect temps continu complique les déplacements et les agents peuvent facilement tourner sur eux mêmes (avoir une entropie forte sur les paramètres d'action `MoveX` et `Rotation` : `movex_entropy`)
- Idem pour les actions : parfois les agents sont très indécis et changent de POI très souvent (ce qui sera appelé `castslots_entropy` et `focus_entropy` dans la partie qui suit)

2. Apprentissage Multi-Agent :

La méthode d'apprentissage choisie dans ce projet est l'algorithme PPO (Proximal Policy Optimization) développé par OpenAI en 2017 (qui a aussi proposé une version GPU adpatée : PPO2) et connu pour ses bonnes performances et sa stabilité face aux approches antérieures (DQN et TRPO : Trust Region Policy Optimization). Ce choix été fait car il s'agit de l'algorithme de Reinforcement Learning que je maîtrise le mieux et que j'ai déjà eu à implémenter et tester dans d'autres projets qui étaient plus liés à la robotique et qui se basaient aussi sur Gym (récupération des observation, espaces d'action, définition de l'environnement et paramètres associés ...). PPO permet d'obtenir des améliorations stables dans les performances d'apprentissage tout en étant implémentable avec une difficulté moyenne. Le tuning des hyperparamètres est plus accessible ce qui permet de gagner un temps considérable en début de développement lorsqu'il faut mettre en place le modèle.

2.1 Proximal Policy Optimization : (Clipped Version)

La méthode PPO est une méthode On-Policy de type 'policy gradient' qui alterne entre un échantillonnage des données lors de l'intéraction avec l'environnement et l'optimisation d'une fonction 'auxiliaire' (surrogate objective function). Elle complémente les méthodes basées sur les Trust Regions et facilite l'implémentation. La principale différence par rapport à TRPO est au niveau de la fonction objective qui devient :

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right]$$

où :

$$\hat{E}_t : \text{operateur d'expectation}$$

$$r_t(\theta)\hat{A}_t : \text{policy gradient objective tel que } \; r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{Old}}(a_t|s_t)}$$

$$clip(r_t(\theta), 1-\epsilon, 1+\epsilon) : \text{la version clippée de } \; r_t(\theta)$$

$$\hat{A}_t : \text{estimation de la valeur relative de l'action choisie au temps t}$$

$$\epsilon : \text{hyperparametre entre } 0.1 \text{ et } 0.2$$

PPO oblige la modification de la valeur de la politique à être dans une fenêtre limitée et évite les changements de valeurs importants.

2.2 Description des comportements obtenus :

Par la modification de la fonction de récompense, deux familles de comportements ont été obtenues. Une famille de stratégies offensives où la destruction de la statue ennemie et des agents adverses est majoritaire et une famille où les agents défendent leur base avec de temps en temps un ou plusieurs agents qui va essayer d'aller endommager la statue des adversaires.

L'implémentation est faite en **PyTorch** et se base sur un modèle utilisé lors de la compétition AI Crowd de 2021 avec pour différence principale une approche où la fonction de récompense est différente pour les agents. L'initialisation et la classe Agent implémentée a donc été modifiée en conséquence et le modèle adapté. Les données d'intérêt ont été sauvegardées à l'aide de Tensorboard au niveau du script `trainer.py`. Ayant une capacité de **3 Gb** au niveau de ma carte graphique, le nombre d'épisodes effectués a été choisi afin de d'éviter les crashs entraînant l'annulation des résultats et avantages obtenus lors de l'apprentissage.

L'entraînement a été fait de la manière suivante. Un premier entraînement de **300 épisodes** (de 20 secondes chacun) sur **50 arènes en simultané** avec un checkpoint tous les **100 épisodes** pour sauvegarder la politique obtenue. Puis en repartant du checkpoint précédent, des entraînements de **100 à 200 épisodes** pour affiner les résultats et le comportement. A chaque partie (au niveau de chaque arène),

les 6 agents en 2 équipes de 3 s'affrontent. Chaque agent essaie d'optimiser ses actions face à l'équipe ennemie qui joue selon un modèle de performance moyenne obtenue lors de la dernière compétition . Les hyperparamètres utilisés sont :

```
"policy_learning_rate": 3e-4,
"value_learning_rate": 1e-3,
"arenas": 50,
"epochs": 10,
"minibatches": 1,
"discretization_bins": 9,
"epsilon": 0.2,
"gamma": 0.995,
"beta": 0.01,
"hidden_size": [[512, 256, 256], [512, 512, 256]],
"checkpoint_interval": 100
```

où `epsilon` est le paramètre décrit dans la description de la fonction objective, `arenas` le nombre d'arènes qui tournent en parallèle lors de l'apprentissage. `gamma` est le discount reward factor commun en RL, `hidden_size` la description du modèle utilisé.

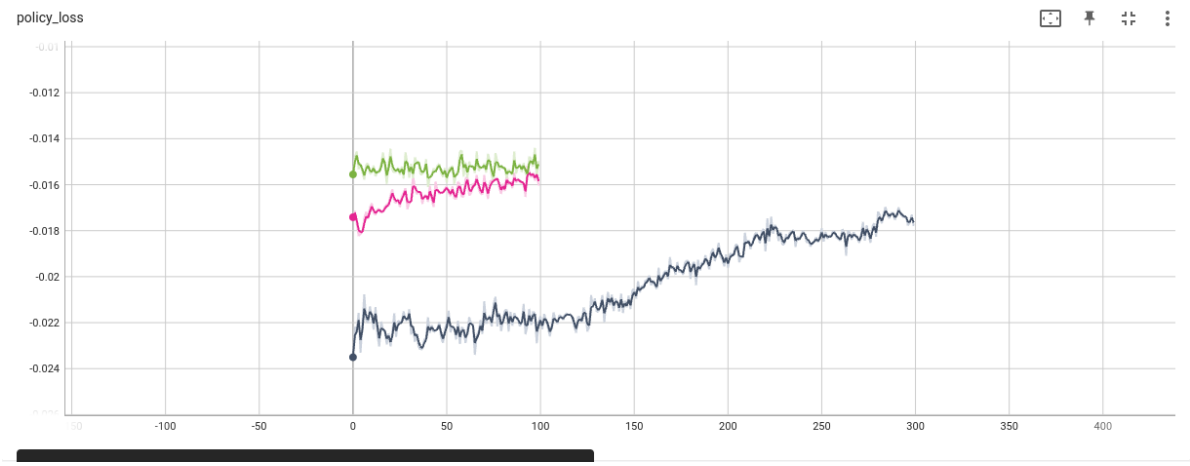
Pour les deux stratégies testées, la règle du jeu imposant l'assignation aléatoire des armes en début de partie a énormément d'influence sur le comportement obtenu. Cela est décrit dans la description des comportements au niveau du notebook `comportements.ipynb`

2.2.1 Stratégie offensive :

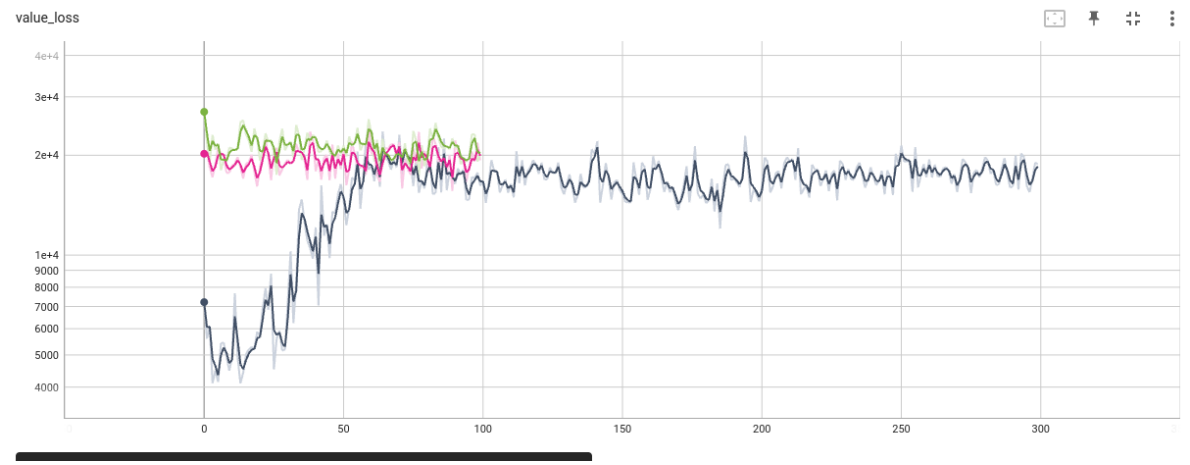
Pour la stratégie offensive, La fonction de reward a été configurée ainsi :

```
"damageEnemyStatue": 10,
"damageEnemyUnit": 5,
"killEnemyStatue": 30,
"killEnemyUnit": 15,
"healFriendlyStatue": 0.5,
"healTeammate1": 0.5,
"healTeammate2": 0.5,
"timeSpentHomeBase": 0,
"timeSpentHomeTerritory": 0,
"timeSpentAwayTerritory": 0,
"timeSpentAwayBase": 0,
"damageTaken": 0,
"friendlyFire": -5,
"healEnemy": -2.0,
"fallDamageTaken": -100,
"statueDamageTaken": 0,
"manualBonus": 0,
"victory": 5,
"loss": -5,
"tie": -1,
"teamSpirit": 0.0,
"timeScaling": 1.0,
```

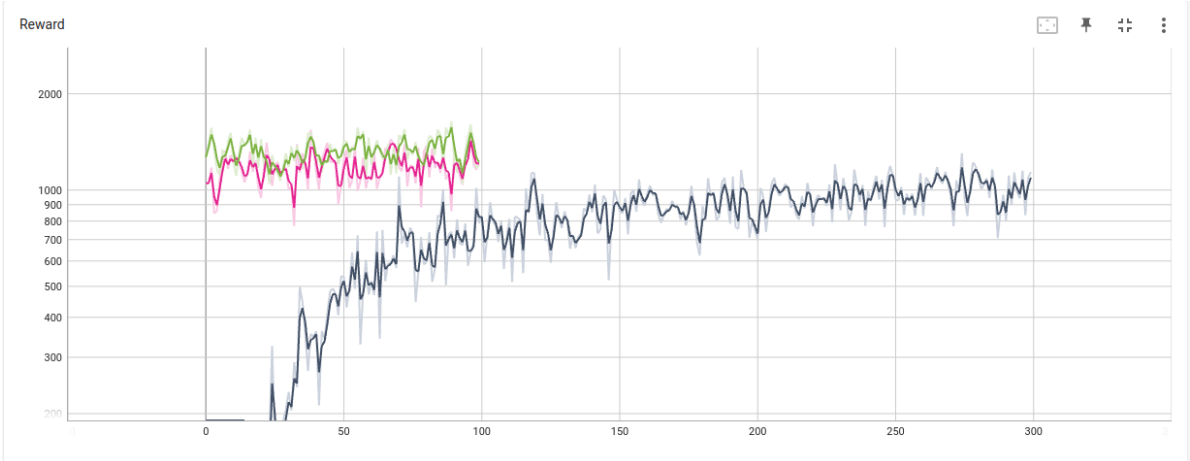
On offre plus de récompenses pour les actions offensives : destruction de la statue ennemie et des agents ennemis. On pénalise le tir ami, et on ne prend pas en compte la position de l'agent sur l'arène (proche de la base ou au niveau de la base ennemie) pour sa récompense. Les actions de soin sont récompensées pour le soin d'un allié et pénalisées plus fortement pour le soin d'un ennemi. La chute de l'arène entraîne la plus forte pénalité afin de pouvoir éviter au maximum cette éventualité dès le début d'apprentissage.



La méthode d'apprentissage mise en place a bien fonctionné comme le montre l'évolution de la policy loss sur les **300 premiers épisodes** et les **2 x 100 épisodes** réalisés à la suite. A partir de 450 épisodes, la policy loss évolue plus lentement et il est utile de regarder les autres paramètres enregistrés, en particulier ceux cités dans la présentation des espaces d'action (`movex_entropy` : entropie du paramètre `moveX` , `rotation_entropy` : entropie du paramètre `Rotation`, `focus_entropy` : entropie du paramètre `ChangeFocus`, `castslot_entropy` : entropie du paramètre `CastingSlot`). Cela permet d'avoir une idée de la précision des mouvements des agents et de la propreté des mouvements réalisés en plus de leur capacité à adopter des comportements leur permettant de jouer efficacement.



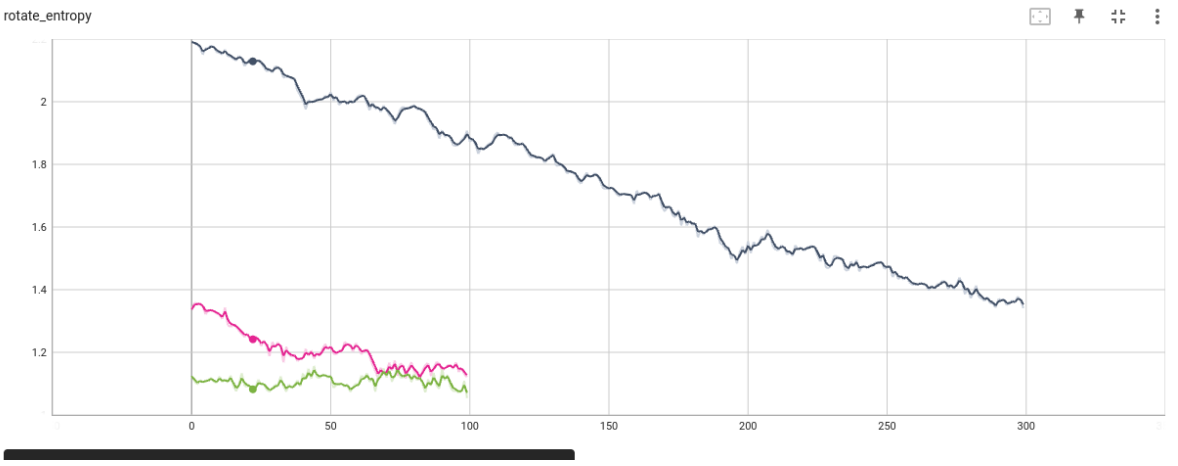
Contrairement à la policy loss, la value loss (moyenne par agent de la différence entre la valeur attendue et observée de la reward) évolue plus faiblement une fois le palier atteint lors des 300 premiers épisodes. Cette valeur se stabilise dès la première partie de l'apprentissage ce qui est en accord avec la stabilisation de la policy loss observée plus haut.



La reward quant à elle continue d'augmenter lors du troisième run d'apprentissage. Cette augmentation est beaucoup plus faible mais continuer l'apprentissage au delà du total des **1200 épisodes** effectués n'induit pas de changement significatif au niveau du comportement des agents. A partir de cette étape, il est intéressant de tester des changements dans la fonction de récompense en partant du checkpoint du modèle sauvegardé.

Évolution des entropies au cours de l'apprentissage :

Les graphes suivants montrent l'évolution de l'entropie des paramètres des agents moyennés par équipe.



Au delà de **400 épisodes**, les mesures d'entropie se stabilisent et ont été considérablement réduites. Les comportements impropres :

- L'agent tourne sur lui même sans arrêt
- L'agent bouge de manière aléatoire et finit par tomber de l'arène
- L'agent attaque ses alliés
- L'agent change de point d'intérêt sans arrêt sans bouger ou en ne bougeant que très peu lors de l'épisode.

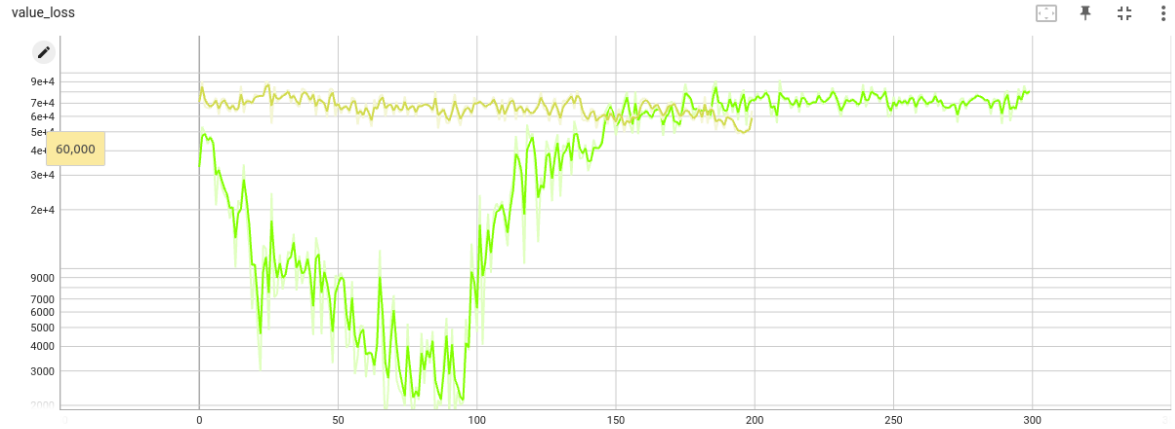
sont éliminés. On obtient alors les comportements qui sont décrits dans le notebook comportant les vidéos enregistrées lors des phases de test. Ces vidéos sont aussi disponibles au niveau de `/tensorboard_data_plots/offensive_run_obj.mp4` et `/tensorboard_data_plots/defensive.mp4`.

2.2.2 Stratégie défensive :

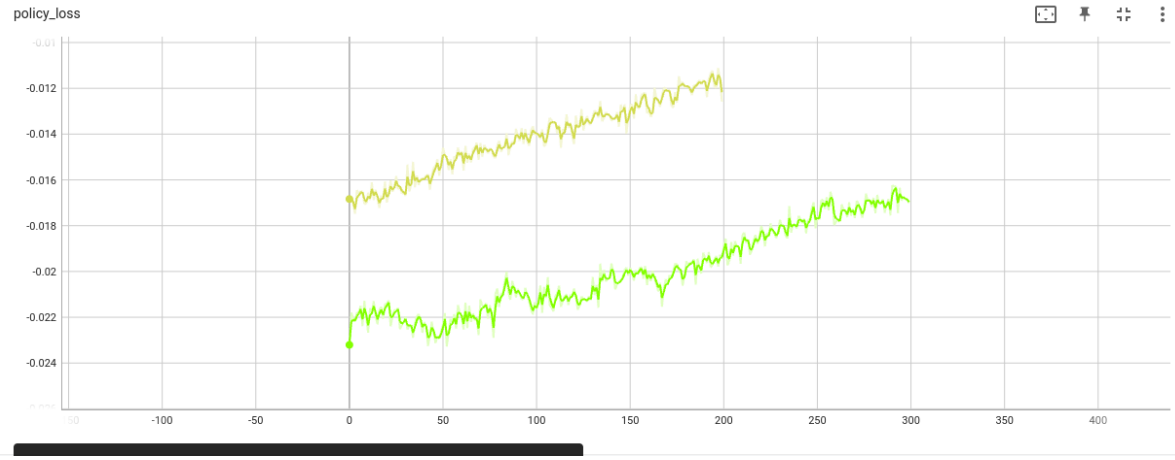
```
"damageEnemyStatue": 0.9,
"damageEnemyUnit": 0.5,
"killEnemyStatue": 3,
"killEnemyUnit": 2,
"healFriendlyStatue": 10,
"healTeammate1": 6,
"healTeammate2": 6,
"timeSpentHomeBase": 3,
"timeSpentHomeTerritory": 1,
"timeSpentAwayTerritory": 0,
"timeSpentAwayBase": 0,
"damageTaken": -1.5,
"friendlyFire": -5,
"healEnemy": -6.0,
"fallDamageTaken": -100,
"statueDamageTaken": -10,
"manualBonus": 0,
"victory": 5,
"loss": -5,
"tie": -1,
"teamSpirit": 0.0,
"timeScaling": 1.0,
```

Les récompenses associées aux dégâts infligés aux ennemis sont sensiblement réduits par rapport à la version précédente. Le temps passé à proximité de la statue de son équipe et autour (jusqu'au milieu de l'arène) est récompensé. Des pénalités sont appliquées en cas de dégâts subis et si la statue alliée subit des dégâts. La pénalité de `-100` associée à la chute de l'arène est conservée. Des récompenses plus conséquentes sont accordées pour le soin des unités alliées avec une préférence pour le soin de la statue. **Les récompenses associées aux dégâts infligés sont utiles car elle permettent de sortir d'une**

situation prévisible au vu de ces paramètres : les agents restent à leur base en se soignant les uns les autres et en essayant uniquement de survivre. Ne pas mettre `damageEnemyStatue` et `damageEnemyUnit` à 0 encourage un ou deux agents à aller marquer des points chez l'équipe adverse.



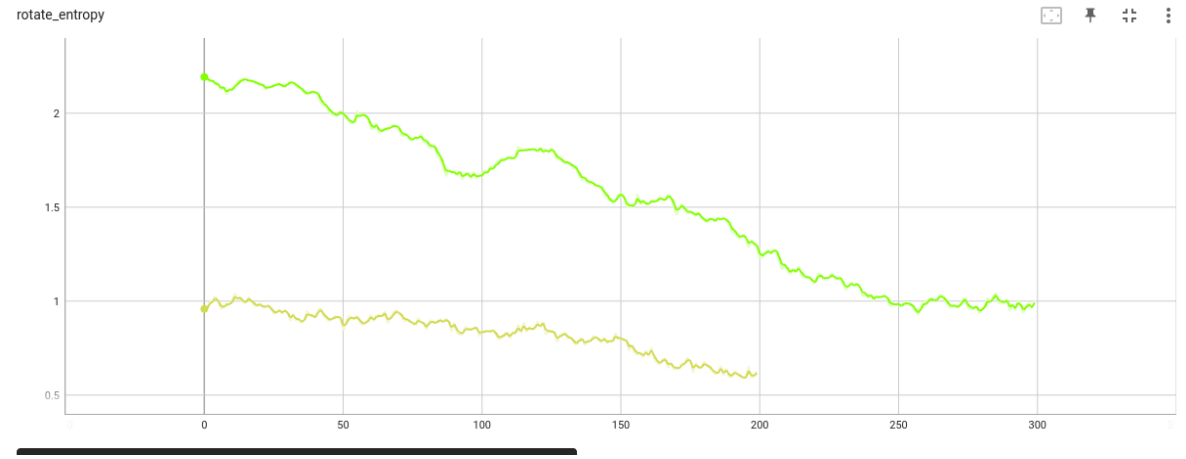
La value loss pour la stratégie défensive évolue différemment. On observe **qu'elle diminue sur les 100 premiers épisodes** avant de commencer à s'améliorer et d'atteindre un palier autour de l'épisode 200. Cette diminution est trompeuse et correspond aux cas où les agents sont statiques et tournent sur eux-mêmes avec des mouvements légers qui les font tomber. Durant cette phase, les agents attaquent parfois leur propre statue. Une fois ces comportements éliminés, la value loss se stabilise et le modèle a une meilleure idée de son environnement et de la reward qu'il peut obtenir.



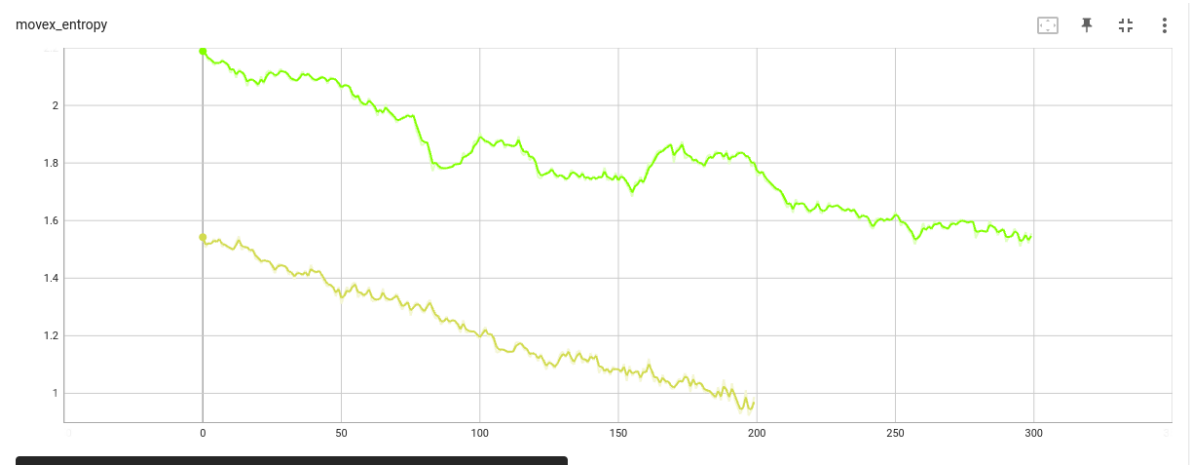
La policy loss évolue plus lentement dans la version défensive. L'apprentissage a été poursuivi sur **200 épisodes** en plus des **500** épisodes (**300 puis 200**) montrés ici et la policy loss s'est stabilisée à **-0.011**. Aucun changement au niveau des comportements n'a été observé à part des mouvements et trajectoires légèrement plus propres.

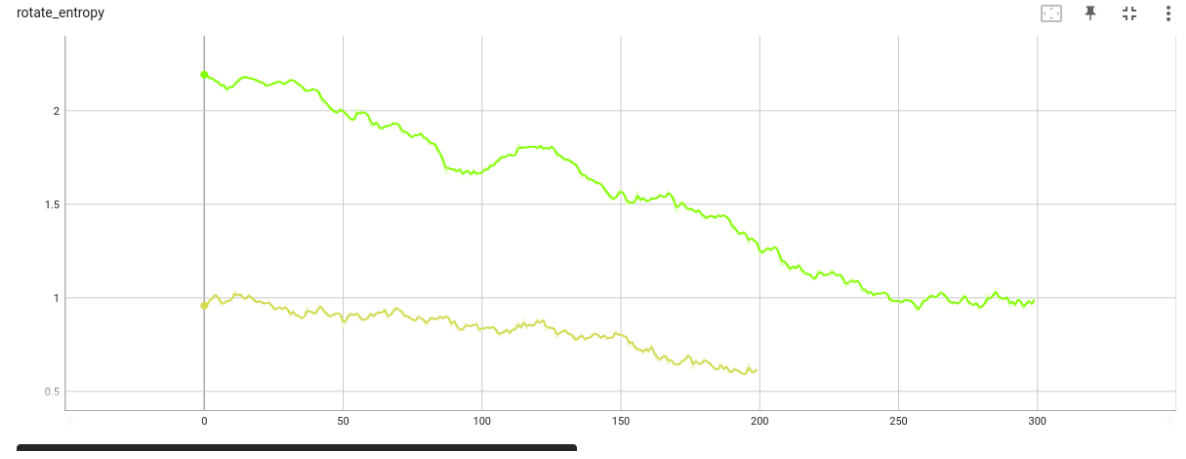
Évolution des entropies au cours de l'apprentissage :

Les graphes suivants montrent l'évolution de l'entropie des paramètres des agents moyennés par équipe.

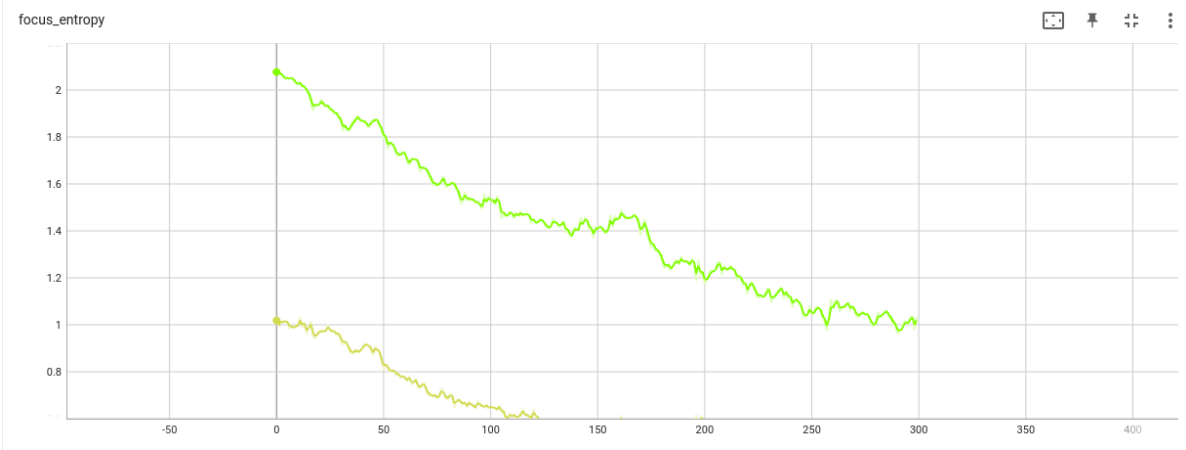

rotate_entropy
2
1.5
1
0.5
0
50
100
150
200
250
300

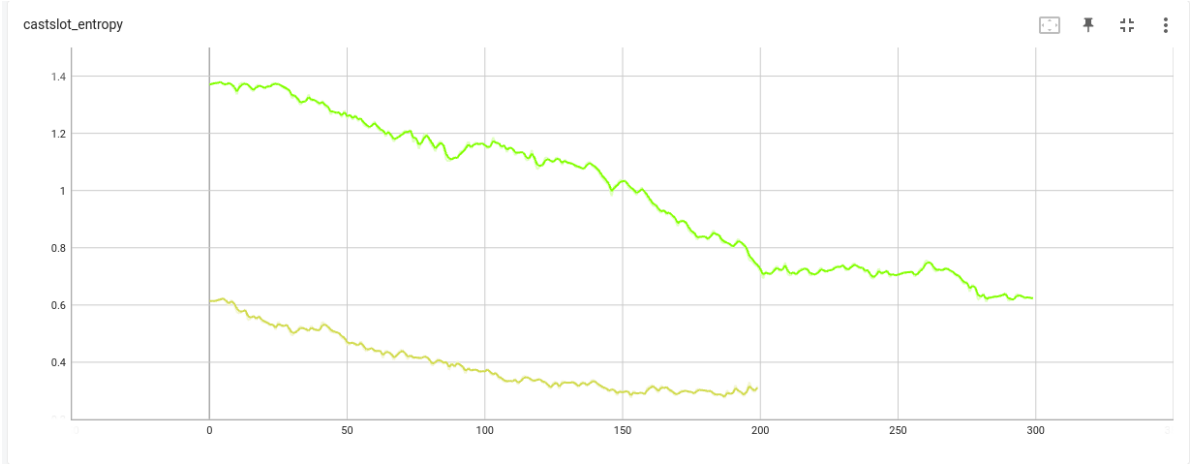
...


movex_entropy
2
1.8
1.6
1.4
1.2
1
0
50
100
150
200
250
300


rotate_entropy
2
1.5
1
0.5
0
50
100
150
200
250
300

...


focus_entropy
2
1.8
1.6
1.4
1.2
1
0.8
-50
0
50
100
150
200
250
300
350
400



La majorité des paramètres d'entropie se stabilisent aussi après **500 épisodes** à l'exception de la `focus_entropy`. Les agents étant encouragés à rester près de leur base, ils changent de point d'intérêt assez souvent pour fixer le ou les agents ennemis entrant dans leur base ou les alliés qui restent aussi à leur proximité. Cette augmentation du nombre de POI à leur voisinage explique cette difficulté à réduire cette entropie et est visible en phase de test.

3. Conclusion :

Ce projet a permis de mettre en place de manière ludique une méthode d'apprentissage par renforcement dans un cadre multi-agent et d'observer les résultats et comportements associés. Cet environnement, bien que comportant une forte part variable avec l'assignation aléatoire des armes en début de partie, a permis de montrer que PPO permet d'obtenir des performances intéressantes et l'émergence d'une diversité de comportements collectifs aboutissant à un objectif. Comme précisé dans le notebook, la stratégie défensive a abouti à un modèle ou les agents sont conscients de l'aspect offensif ou défensif de leur arme et de leur capacité à aller chez l'ennemi pour marquer des points.

Continuer l'apprentissage à partir des checkpoints réalisés en modifiant la fonction de récompense vers des stratégies mixtes est une piste d'ouverture intéressante. il serait aussi intéressant de tester si fixer les armes pour chaque équipe en début de partie peut permettre d'aboutir à des stratégies et comportements plus stables.

4. Bibliographie :

TutoMountRourke:Randomactionsfordr-derk's mutantbattlegrounds : https://github.com/MountRourke/Randos

Proximal Policy Optimization : https://openai.com/blog/openai-baselines-ppo/

Proximal Policy Optimization Algorithms : https://arxiv.org/abs/1707.06347

Dr-Derk reinforcement learning tutorial : https://github.com/alex-snd/derkAgent

OpenAI baselines : https://github.com/openai/baselines

Derk's Gym 1.1.1 Documentation : http://docs.gym.derkgame.com/index.html#

PyTorch : https://pytorch.org/

AI Crowd competition : https://www.aicrowd.com/challenges/dr-derks-mutant-battlegrounds/leaderboards