Paul Malet
IASD apprentissage
2024-2025

Étude et résolution de Lunar Lander avec PPO

Table des matières

Introduction

Pour ce projet d'apprentissage par renforcement, j'ai choisi de travailler sur Lunar Lander, un jeu que j'avais exploré auparavant pour apprendre à coder en Python. Lors de cette première expérience, j'avais conçu une stratégie basée sur des actions discrètes, choisies de manière déterministe en fonction de la position, de la vitesse et de la zone d'atterrissage cible. Dans ce projet, mon objectif est de résoudre ce problème dans un environnement continu à l'aide d'un algorithme d'IA.

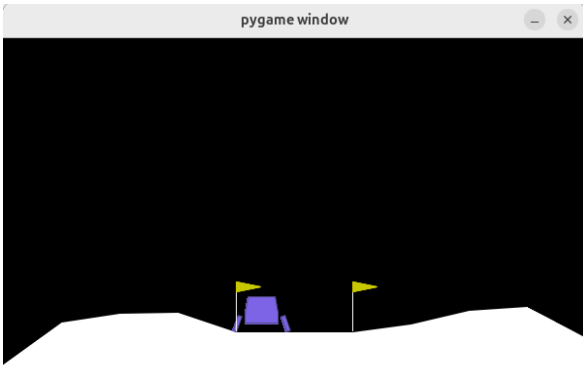
J'ai choisi d'utiliser Proximal Policy Optimization (PPO) plutôt que REINFORCE pour sa robustesse et sa plus grande efficacité. Afin de gagner du temps sur la phase de développement, j'ai décidé de m'appuyer sur des implémentations open source existantes. J'ai notamment utilisé celles des projets GitHub pytorch_minimal_ppo de Barhate Nikhil et V-MPO_Lunarlander de l'utilisateur YYCAAA. Ces implémentations ont été modifiées pour s'adapter à mon problème et à mes objectifs spécifiques. Mon travail a ensuite consisté à optimiser les paramètres de l'algorithme pour résoudre Lunar Lander de manière plus efficace. Le répertoire Github contenant mon code et des agents entraînés se trouve ici.

Dans ce rapport, je vais tout d'abord présenter les premières améliorations que j'ai apporté aux implémentations de PPO trouvées sur GitHub, ensuite je vais présenter les paramètres du modèle, puis j'expliquerai la manière dont j'ai procédé pour la recherche des paramètres, et enfin j'exposerai l'entraînement et les résultat d'un agent dans deux environnements différents.

I. Améliorations du code

Les implémentations récupérées sur GitHub étant anciennes, plusieurs ajustements ont été nécessaires pour garantir leur compatibilité avec les versions actuelles des bibliothèques, telles que Gymnasium et CUDA. Ces modifications ont permis de rétablir le bon fonctionnement du code et d'optimiser la vitesse d'entraînement de l'agent.

Afin d'améliorer la lisibilité et la modularité, j'ai restructuré le code en classes, fonctions et fichiers distincts. En particulier, j'ai séparé les phases d'entraînement et de test de l'agent, qui peuvent être lancées par une ligne de commande, avec les paramètres voulus. J'ai également ajouté des visualisations périodiques des performances de l'agent : tous les 500 épisodes, une fenêtre vidéo montre le comportement de l'agent pour observer ses progrès.



Pour suivre les performances au fil de l'entraînement, j'ai intégré une collecte de l'historique des récompenses et des durées des épisodes. Ces données sont ensuite tracées sous forme de moyennes mobiles sur 20 épisodes, ce qui réduit l'impact du bruit dans les résultats.

Enfin, j'ai ajouté une heuristique pour réduire progressivement l'écart type de la distribution des actions à mesure que l'agent s'améliore. Cette stratégie favorise l'exploration initiale tout en permettant à l'agent de mieux exploiter ses connaissances dans les phases avancées de l'apprentissage.

II. Présentation des paramètres

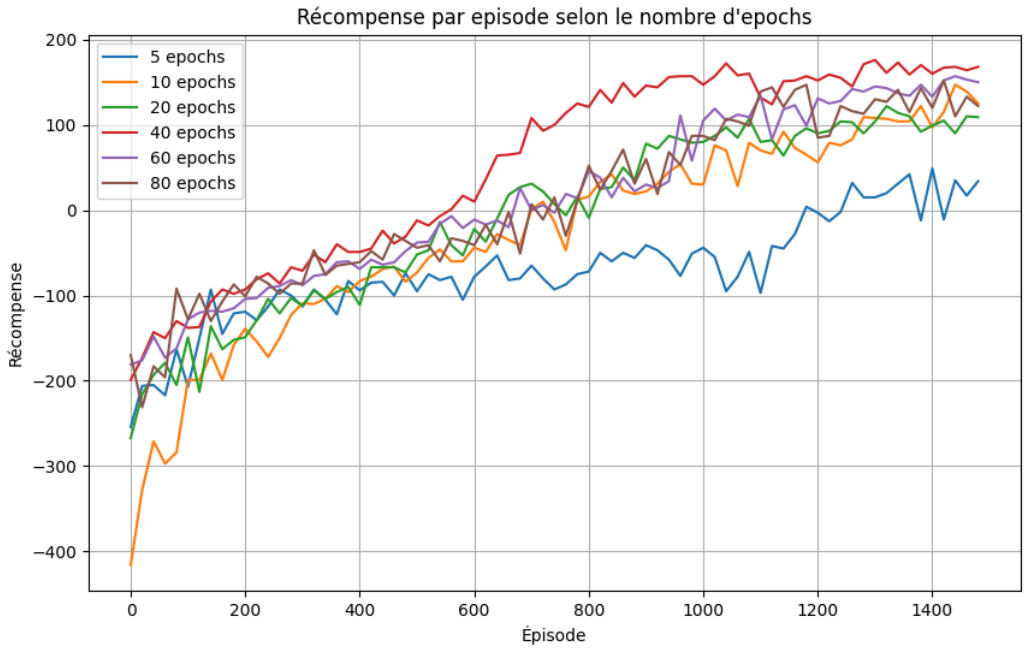
Les paramètres principaux du modèle et leurs valeurs initiales sont présentés dans le tableau suivant :

Paramètre	**Valeur initiale**	**Description**
solved_reward	200	Seuil moyen de récompense pour considérer l'environnement comme résolu.
max_episodes	2500	Nombre maximal d'épisodes avant d'interrompre l'entraînement.
max_timesteps	1000	Nombre maximal de pas par épisode.
update_timestep	4000	Pas accumulés avant de mettre à jour la politique.
action_std	0.5	Écart type initial de la distribution des actions.
k_epochs	40	Itérations d'optimisation par lot.
eps_clip	0.2	Intervalle de clipping pour limiter les changements de politique.
gamma	0.99	Facteur d'actualisation pour pondérer les récompenses futures.
lr	0.0003	Taux d'apprentissage.

III. Recherche et ajustement des paramètres

1. Nombre d'époques d'optimisation - *k_epochs*

Le nombre d'epochs est le paramètre qui m'a semblé le plus indépendant des autres et que j'ai décidé d'améliorer en premier. Je l'ai alors fait varier de 5 à 80 pour pour évaluer son impact sur les performances sur 1500 épisodes d'entraînement.



On voit alors que c'est avec 40 epochs que l'apprentissage se fait le plus rapidement. C'est cette valeur que nous garderons pour la suite du projet.

Par ailleurs, voyant que la récompense était proche de 200 (qui est le niveau pour lequel le problème est considéré comme résolu) au bout de 1500 épisodes, j'ai décidé de garder cela comme nombre de *max_episodes*. L'entrainement d'un agent prend alors environ 4 minutes, ce qui permet d'itérer.

2. Durée maximale d'un épisode - *max_timesteps*

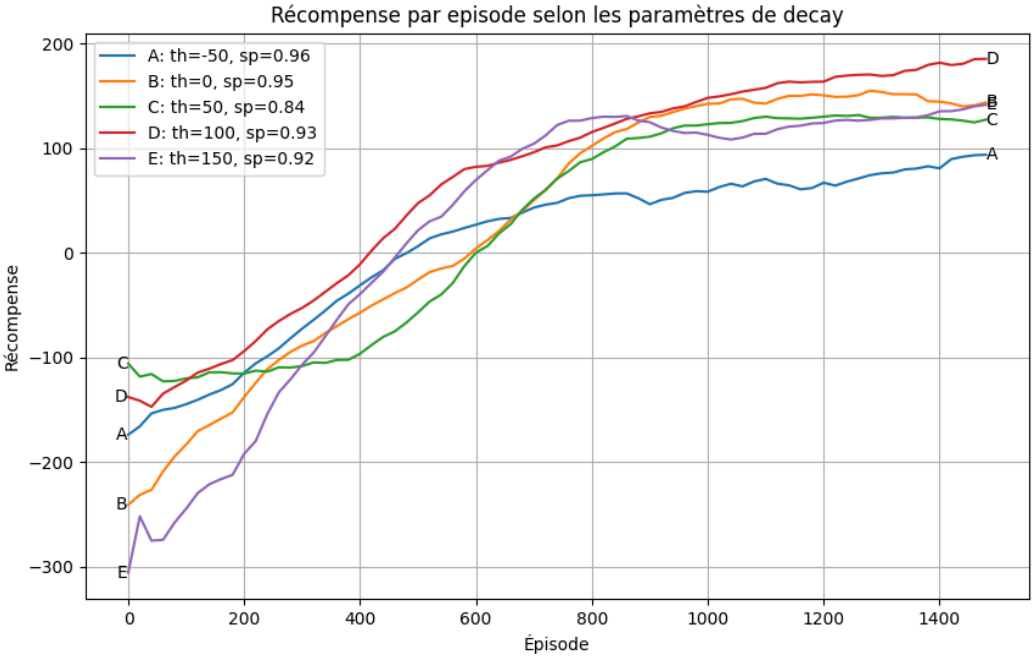
L'environnement Lunar Lander présente une particularité : lorsque le module atterrit mais continue de trembler sur la surface, l'épisode ne se termine pas automatiquement. Cela allonge inutilement les épisodes et complique l'apprentissage. En observant les comportements des agents, j'ai constaté que les atterrissages réussis se faisaient en moins de 300 pas. J'ai donc fixé la durée maximale à 350 timesteps pour laisser une marge raisonnable.

3. Écart type de la distribution de la politique - *action_std*.

Le fait d'avoir tracé lors de la recherche de *k_epochs* l'évolution des récompenses en fonction des épisodes permet de voir que l'agent commence avec une récompense d'environ -200, qu'il s'améliore, puis que son apprentissage ralenti à partir de 100 de récompense, et qu'il atteint un plateau à 150. La récompense considérée par défaut comme seuil de résolution du problème est de 200, avec 100 de récompense l'agent est donc déjà d'un certain niveau. Garder un paramètre *action_std* élevé une fois ce niveau atteint est donc contre-productif, il faut alors le faire réduire pour que l'agent fasse moins d'exploration et qu'il exploite plus ses connaissances déjà acquises.

J'ai introduit une heuristique adaptative qui réduit *action_std* lorsque la récompense moyenne dépasse certains seuils : -50, à 0, à 50, à 100 ou à 150, avec des réductions plus

lente pour les premières valeurs car les seuils sont atteints plus tôt.. La réduction se fait alors progressivement jusqu'à une valeur minimale de 0.1, pour maintenir un minimum d'exploration pour la suite de l'entraînement. Ces deux paramètres sont appelés *decay_threshold* et *decay_speed*, th et sp pour faire court dans la légende du graphique.



La courbe A correspond à faire réduire l'exploration à partir d'une récompense de -50, et cela conduit à atteindre un plafond de niveau car arrive un stade où l'agent n'arrive plus à s'améliorer.

La courbe E correspond au comportement opposé : attendre que l'agent atteigne une récompense de 150 pour faire moins d'exploration. Le problème de cette approche est que cette récompense est très longue à atteindre avec un écart type de 0.5, ou même pas atteignable avant la fin de l'entraînement. Cela a été le cas avec cet entraînement court.

La courbe D est la seule n'ayant pas atteint un plateau de récompense à la fin de l'entraînement. Nous allons alors partir sur un seuil de début de décroissance à une récompense de 100, et ajuster la vitesse de décroissance en fonction de la durée de l'entraînement.

IV. Entraînement et test de l'agent

Pour ce dernier entraînement, nous avons décidé d'augmenter le nombre d'épisodes à 2500, contre 1500 dans les précédentes expériences. En analysant le graphique précédent, nous constatons que l'agent atteint le seuil de récompense de 100 après environ 700 épisodes. Nous avons donc choisi de faire décroître progressivement l'écart type de 0.5 à 0.1 sur les 1000 épisodes suivants. Les 800 épisodes restants sont alors effectués avec un écart type fixe de 0.1.

L'ajustement de l'écart type est réalisé dans le code tous les 20 épisodes (choix arbitraire). La formule utilisée pour la décroissance est :

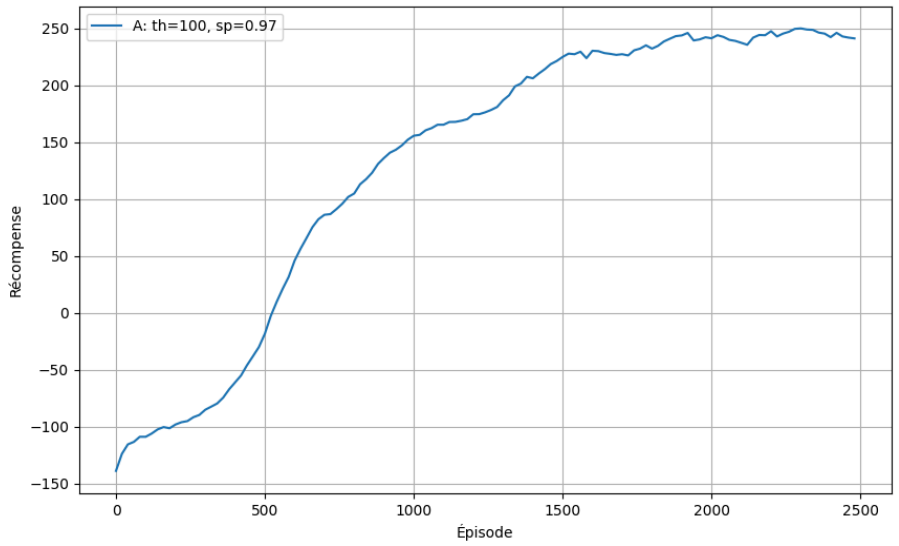
$$0.5 \times v^{\frac{1000}{20}} = 0.1$$

Ce calcul donne $v = 0.968$, que nous arrondissons à $v = 0.97$ pour simplifier.

L'entraînement à pris environ 8 minutes à être effectué.

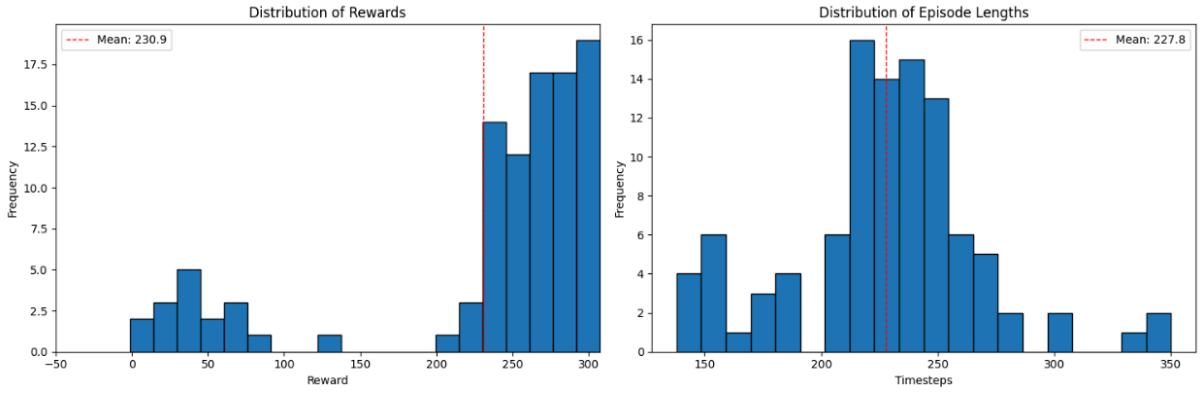
Une fois l'entraînement terminé, nous avons testé l'agent en utilisant le fichier dédié au test sur 100 épisodes. Les résultats obtenus montrent une récompense moyenne de **231** en **228 timesteps**. Ces chiffres indiquent un agent performant qui parvient à atterrir efficacement. Il y a cependant toujours certains atterrissages ratés, comme le montre l'histogramme des récompenses.

Ci dessous la courbe de l'évolution des récompenses lors de l'entraînement, au fil des épisodes :



Cette courbe confirme également que la stratégie de réduction progressive de l'exploration permet d'améliorer les performances finales en exploitant mieux les connaissances acquises. En effet, la vitesse d'apprentissage reste élevée entre 100 et 200, et continue même après.

Ci dessous les histogrammes de la distribution des récompenses des 100 épisodes de test et de leur longueur.

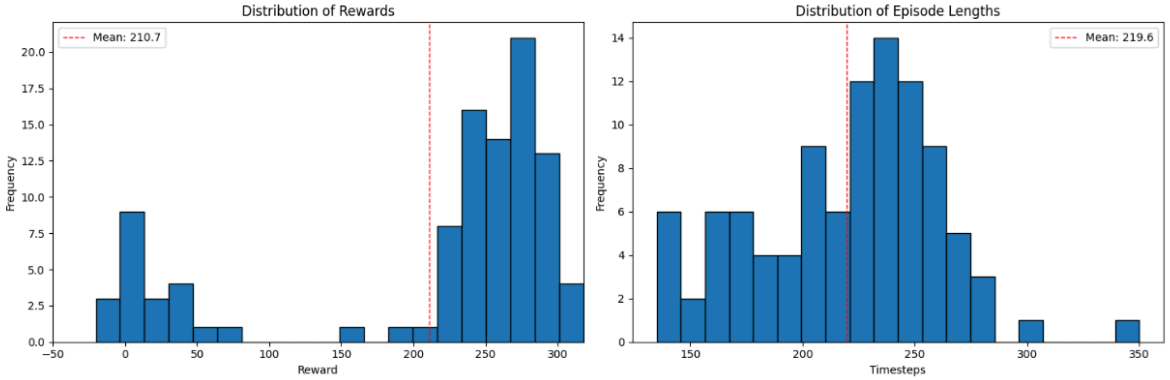


V. Test de l’agent dans un environnement complexe

Jusqu'à présent, nos expériences se sont déroulées dans un environnement où la seule force extérieure était la gravité. Cependant, le jeu Lunar Lander de Gym propose une variante incluant des conditions plus complexes, telles que le vent et les turbulences. Ces forces externes, dont on peut ajuster les intensités, introduisent une dynamique supplémentaire qui modifie les trajectoires et complique les manœuvres d'atterrissage.

Tester notre agent dans ce contexte est pertinent pour évaluer la robustesse de l'agent face à des variations environnementales, et surtout une complexification. Cela permet de mesurer sa capacité à généraliser son comportement à des conditions non vues pendant l'entraînement initial.

Les résultats obtenus sur 100 épisodes montrent une récompense moyenne de **210** en **220 timesteps**, ce qui est 10 de moins que ceux du test dans un environnement classique. Ces chiffres montrent que l’agent reste performant, résout le problème (seuil de 200 pour le considérer comme tel) et a donc été capable de s’adapter à un environnement d’une plus grande complexité, sans avoir été entraîné précisément pour.



Conclusion

Ce projet d’apprentissage par renforcement appliqué à Lunar Lander a permis de démontrer l’efficacité de l’algorithme Proximal Policy Optimization (PPO) dans un environnement continu. En partant d’implémentations open source, j’ai pu les améliorer et les optimiser afin de résoudre ce problème de manière plus efficace. L’intégration de techniques telles que la réduction progressive de l’exploration a joué un rôle clé dans l’amélioration des performances de l’agent.

Les résultats montrent qu’un agent entraîné selon cette approche est capable de résoudre le problème dans un environnement classique, mais aussi dans un environnement plus complexe avec vent et turbulences. Cela prouve non seulement ses capacités d’apprentissage, mais également une certaine robustesse et capacité de généralisation à des conditions non rencontrées durant l’entraînement. Ces performances, associées à des durées d’entraînement raisonnables, rendent cette méthode adaptée pour des tâches similaires nécessitant des agents performants.

Bibliographie

Nikhil, Barhate. “Minimal PyTorch Implementation of Proximal Policy Optimization.” *github.com/nikhilbarhate99*, GitHub, 2021, https://github.com/nikhilbarhate99/PPO-PyTorch.

YYCAAA. “V-MPO_Lunarlander.” *github.com/YYCAAA*, GitHub, 2020, https://github.com/YYCAAA/V-MPO_Lunarlander.