# MSE301 - Rendu de Reinforcement Learning

Alexandre TURKI

Février 2025

# Table des matières

# 1 Résumé du papier

## 1.1 Présentation du papier

Dans ce rapport, nous allons travailler sur les **Proximal Policy Optimization Algorithms** de John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford et Oleg Klimov. Ce papier propose une nouvelle famille de méthodes de policy gradient method.
Cette méthode alterne entre échantillonnage de données par interaction avec l'environnement et l'optimisation d'une fonction "surrogate" à l'aide d'une méthode de descente de gradient stochastique.

## 1.2 Introduction

De nombreuses méthodes de Reinforcement Learning utilisant des approximations de fonctions de réseaux de fonctions ont été proposées ces dernières années. Le papier mentionne le **deep Q-learning**, les méthodes de **Policy Gradient vanille** et les **Trust Region Policy Optimization** (TRPO). Ces méthodes présentent cependant des problèmes de scalabilité, de robustesse ainsi que de data efficency.
L'objectif de ce papier est d'introduire la **Proximal Policy Optimization** (PPO) qui en utilisant uniquement de l'optimisation de premier ordre va résoudre les problèmes énoncés plus haut.

## 1.3 Contexte : Policy Optimization

Le papier introduit différents concepts nécessaires à sa compréhension :

**Méthode Policy-Gradient :**
Ce sont des méthodes qui calculent un estimateur du policy-gradient et à l'insérer dans un algorithme de montée de gradient stochastique. L'estimateur prend souvent la forme suivante :

$$\hat{g} = \hat{\mathbb{E}}_t[\nabla_\theta log\, \pi_\theta(a_t|s_t)\hat{A}_t]$$

avec $\pi_\theta$ policy stochastique et $\hat{A}_t$ un estimateur de la fonction avantage au pas t.

L'estimateur $\hat{g}$ est obtenu en différentiant l'objectif :

$$\mathbf{L}^{PG}(\theta) = \mathbb{E}_t[log\, \pi_\theta(a_t|s_t)\hat{A}_t]$$

**Trust Region Methods** :
Les TRPO cherchent à maximiser des fonctions objectives ("surrogate" objective) soumises à des contraintes sur la taille de la mise à jour de policy.
Il est suggéré d'utiliser une pénalité plutôt qu'une contrainte et d'ainsi résoudre l'équation suivante :

$$\underset{\theta}{maximize}\, \hat{\mathbb{E}}_t\, [\frac{\pi_\theta\,(a_t|s_t)}{\pi_{\theta_{old}}\,(a_t|s_t)}\hat{A}_t - \beta KL[\pi_{\theta_{old}}(.|s_t), \pi_\theta(.|s_t)]]$$

Cependant, dans le cas de la TRPO il est difficile de choisir un $\beta$ performant. Ainsi, pour un algorithme qui veut résoudre au premier ordre, il n'est pas suffisant de choisir un $\beta$ fixe.

**Clipped Surrogate Objective** :
On définit $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ le ratio de probabilité. La TRPO maximise un objectif surrogate :

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[r_t(\theta)\hat{A}_t]$$

Cependant, sans contrainte, cet objectif crée une mise à jour de policy beaucoup trop large, c'est pourquoi on introduit l'objectif suivant :

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

On ajoute ainsi une pénalité pour restreindre la mise à jour de policy. L'objectif est le ratio de probabilité reste entre $1-\epsilon$ et $1+\epsilon$.

**Coefficient de pénalité adaptatif de KL**
Il est possible d'utiliser une divergence de KL en y ajoutant une pénalité comme expliqué plus haut. Ainsi, à chaque mise à jour de policy il est possible de réaliser l'algorithme suivant :

- $L^{KLPEN}(\theta) = \hat{\mathbb{E}}_t\,[\frac{\pi_\theta\,(a_t|s_t)}{\pi_{\theta_{old}}\,(a_t|s_t)}\hat{A}_t - \beta KL[\pi_{\theta_{old}}(.|s_t), \pi_\theta(.|s_t)]]$
- calculer $d = \hat{\mathbb{E}}_t[\beta KL[\pi_{\theta_{old}}(.|s_t), \pi_\theta(.|s_t)]]$
- si $d < d_{targ}/1.5$, $\beta \leftarrow \beta/2$
- si $d < d_{targ} * 1.5$, $\beta \leftarrow \beta * 2$

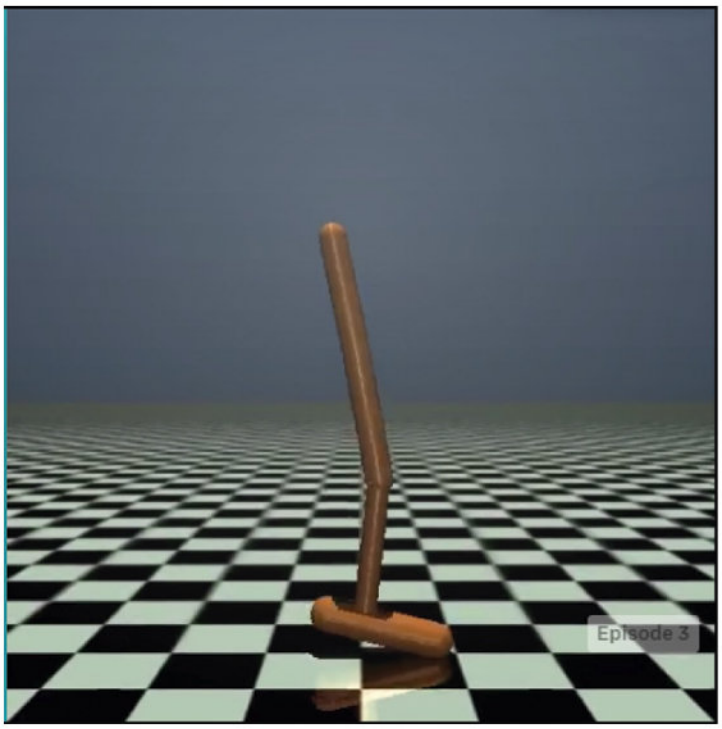
On utilise alors $\beta$ pour la mise à jour de policy suivante. Dans notre cas, nous allons nous limiter au clipped surrogate objective.

# 2 Implémentation de l'algorithme sur Mujoco

## 2.1 Présentation du problème Hooper v2

Nous allons implémenter l'algorithme décrit dans ce papier de OPENAI. Comme dans une partie du papier, nous allons l'utiliser sur un environnement Mujoco qui est inclus dans le package *gym* de OPENIA.

L'environnement choisi sera le Hopper v2. L'objectif est de faire sauter la jambe illustrée ci-dessous.




L'environnement dispose pour cela de 11 variables d'état qui décrivent la position et la vitesse de différents éléments de l'objet comme :

- La coordonnées z du haut de la jambe.
- L'angle du haut de la jambe.
- La vitesse de la coordonnée x du haut de la jambe.
- La vitesse de la coordonnée z du haut de la jambe.
- L'angle de la cheville de la jambe
- etc,...

L'environnement dispose de 3 variables d'actions qui permettent d'interagir avec l'environnement :

— Le couple appliqué au joint de la cuisse.

— Le couple appliqué au joint du pied.

— Le couple appliqué au joint de la jambe.

## 2.2 Présentation de l'algorithme du papier :

Le papier présente l'algorithme suivant :

**Algorithm 1** PPO, Actor-Critic Style

**for** iteration= $1, 2, \ldots$ **do**
  **for** actor= $1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{old}}$ in environment for $T$ timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  **end for**
  Optimize surrogate $L$ wrt. $\theta$, with $K$ epochs and minibatch size $M \leq NT$
  $\theta_{old} \leftarrow \theta$
**end for**

Pour implémenter cet algorithme, nous allons utiliser le package pytorch pour définir les différentes policy et optimiser la fonction loss surrogate. En particulier, nous utiliserons la version $L^{CLIP}$. L'optimisation se fera par l'algo d'optimisation de Adam de pytorch.

On définit alors un **Réseau de Politique (Policy Network)** :

Son objectif est de déterminer la meilleure action à suivre à un état donné du système.
Il prend alors en entrée les 11 dimensions de l'état du système et en ressort une distribution de probabilité sur l'ensemble des actions possibles.
Les poids associés à chaque prise de décision seront alors optimisés par PPO.

On définit aussi un **Réseau de Valeur (Value Network)** :

Il estime la valeur d'un état donné ou bien la récompense associée à la politique actuelle du système.
Il prend alors l'état actuel du système tout comme le réseau de politique et produit ensuite une estimation de la valeur de cet état.
Il permet donc de calculer les avantages lors de la PPO et de choisir si cette action est moins bonne ou pire que la moyenne à un état donné.

Ces deux réseaux seront alors entraînés par la PPO pour obtenir une bonne politique face au problème du Hooper.

Il est aussi important de définir la **PPO loss function**. L'objectif est de calculer à chaque itération la Policy Loss et la Value Loss de la politique actuelle.

— La **Politique Loss** (perte de politique) permet de minimiser la perte de la politique en utilisant la plus petites perte des deux politiques.

— La **Value Loss** (perte de valeur)permet de minimiser l'erreur entre les valeurs estimées par le réseau et les récompense réelles. Dans notre cas, nous utiliserons l'erreur quadratique.

C'est une étape particulièrement importante dans l'algorithme car c'est à partir de ces valeurs que l'algorithme va être optimisé.

Une fois les **Réseaux** et la **PPO Loss Funtion** définis. Il faut créer une fonction de **collecte de trajectoires** qui sera utilisée à chaque epoch dans la PPO. C'est une étape importante car les trajectoires permettent de fournir des données expérimentales de la politique actuelle pour par la suite améliorer cette politique.

Lors de cette étape, les avantages sont calculés. Ils mesurent la qualité des actions prises à chaque trajectoire par rapport à la politique actuelle utilisée.
Ces données seront alors utilisées par les réseaux et la Policy Loss Function pour améliorer la politique actuelle.

L'algorithme de la PPO va calculer les **avantages** mesurant si une action est meilleure ou moins bonne que la moyenne. Les avantages sont calculés comme la différence entre les récompenses réelles et les valeurs estimées dans le Value Network.

Les avantages vont permettre de calculer la PPO Loss Function définie plus haut et les pertes de la politique associée. Ces pertes permettront de calculer les gradients de perte par rapport aux paramètres des Value Network et Policy Network et de les mettre à jour.

Les **Value Network** et **Policy Network** sont mis à jour de cette manière car ils garantissent :

— **Stabilité** via le **Clipped Surrogate Objective**. Assurant que les mises à jour de la PPO n'aient pas de fluctuations trop importantes.

— **Efficacité** grâce à la mise à jour simultanée des deux Networks.

— **Réduction de Variance** avec l'aide des avantages calculés à partir des récompenses réelles et celles estimées par le Value Network.

# 3 Expérimentations et Résultats

Nous avons expliqué comment utiliser l'algorithme de la PPO. Nous allons pour ce projet reproduire une partie de l'algorithme. Dans notre cas, nous allons utiliser un Avantage très simple mais avec un Clipped Surrogate Objective. Nous allons aussi utiliser une Mean Square Error plutôt que la KL loss dans le code réalisé. L'algorithme d'optimisation qui sera utilisé à chaque Epoch sera un Algorithme d'Adam (Adaptative Moment Estimation) qui est développé dans le package pytorch.

## 3.1 Essai de l'algorithme

Nous allons commencer par utiliser l'algorithme à l'aide des hyperparamètres décrits dans le papier sur la PPO c'est-à-dire :

- **Adam stepsize** : 3 × 10e-4
- **Num. epochs** : 10
- **Minibatch size** : 64
- **Clipped Epsilon** : 0.1
- **TimeSteps** : 1000000

Nous faisons alors tourner le code pour ces hyperparamètres :

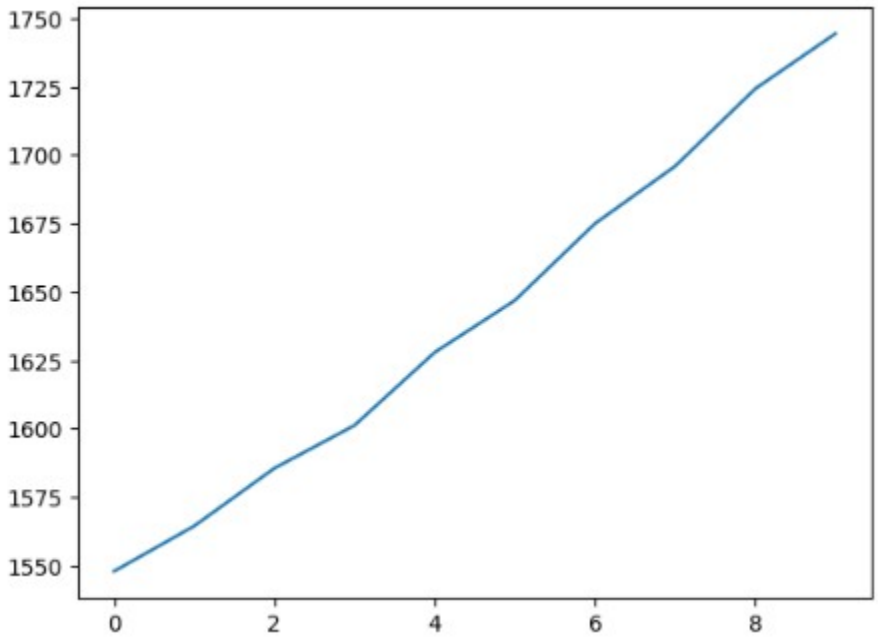



FIGURE 1 – Evolution de la Récompense pour un premier set d'hyperparamètres.

Nous remarquons que la reward augmente bien, cependant, elle n'augmente que très légèrement. Cela est probablement dû à un nombre d'épisodes très faible et une learning rate potentiellement

inadéquate qui empêche l'algorithme d'Adam d'avancer correctement.
Il est aussi important de regarder les pertes de politiques et les pertes de valeur :

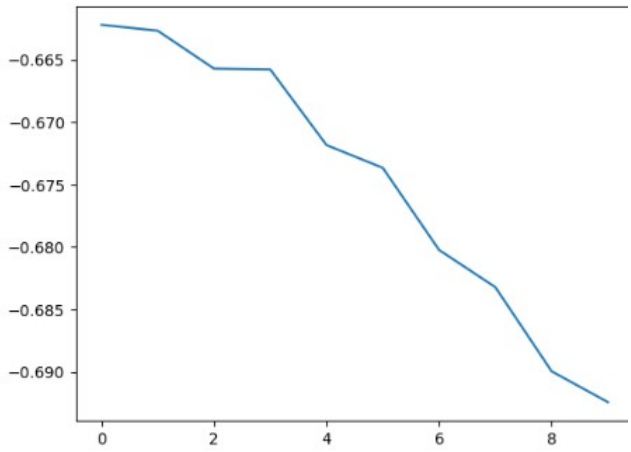



FIGURE 2 – Evolution de la Policy Loss pour 10 Epoch et 1,000,000 Time steps.

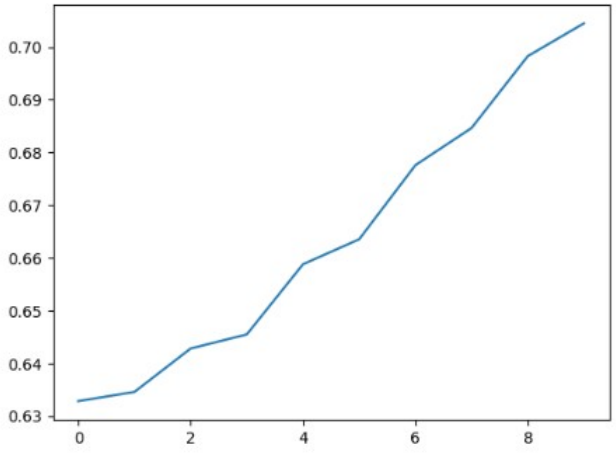



FIGURE 3 – Evolution de la Value Loss pour 10 Epoch et 1,000,000 Time steps.

Nous remarquons que les pertes augmentent malgré que la récompense de l'algorithme augmente. Ce n'est pas bon signe pour les pertes d'augmenter dans un tel algorithme. Cependant, je n'ai pas trouvé d'explication autre que le nombre d'épisodes est insuffisant pour obtenir des résultats pertinents malgré le grand nombre de trajectoires par épisodes. Il est possible que la façon de calculer l'avantage dans la Loss Function ne soit pas assez élaborée ou bien que la Mean Square Root Loss ne convienne pas au problème. Nous allons maintenant étudier l'algorithme avec d'autres paramètres pour en tirer de meilleures conclusions potentiellement.

### 3.2 Impact du nombre d'Epoch

Nous avons vu que les Value Loss et Policy Loss restent assez loin de zéro. Il est donc naturel de vouloir augmenter le nombre d'épisodes de l'apprentissage. Nous avons ensuite regardé l'impact du nombre d'épisodes sur l'apprentissage. Les hyperparamètres utilisés sont les mêmes que plus haut à l'exception que nous choisirons un **timesteps** de 2048 et un nombre d'épisodes (epoch) de 1000. L'évolution de la récompense, de la Value Loss ainsi que de la Policy Loss est étudiée au cours de cette expérimentation. Nous commencerons par l'évolution de la récompense :

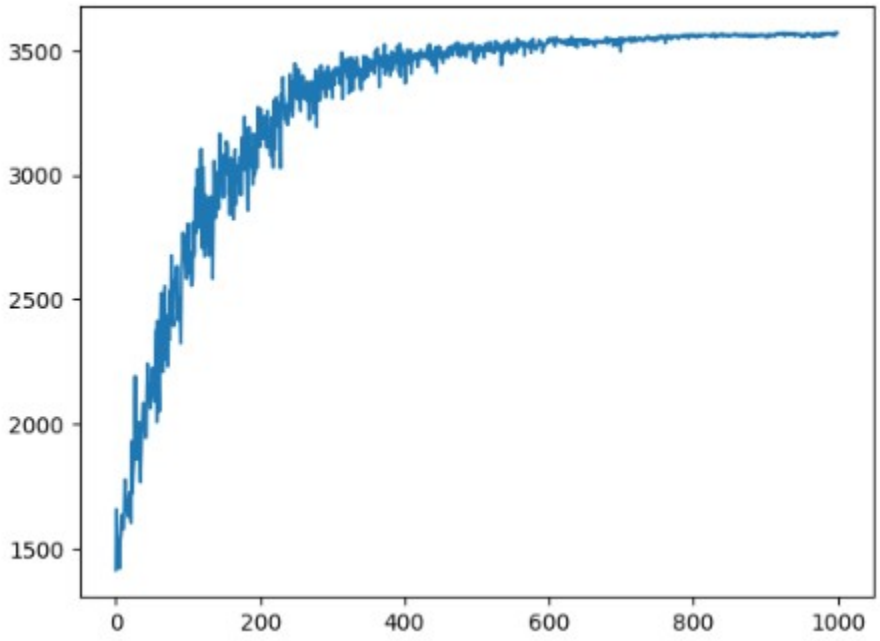



FIGURE 4 – Evolution de la Récompense pour 1000 Epoch.

Comme attendu, nous remarquons bien que le nombre d'épisodes a un gros impact sur la qualité de l'entraînement. Il semble donc préférable de choisir de plus petits TimeSteps avec un plus grand nombre d'Epoch au premier abord. Il est tout aussi important d'observer l'évolution de la Policy Loss et de la Value Loss qui sont des paramètres tout particulièrement importants pour la partie optimisation de l'algorithme.

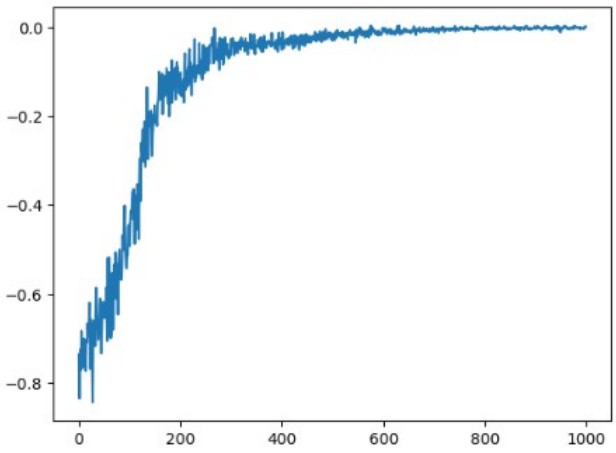



FIGURE 5 – Evolution de la Policy Loss pour 1000 Epoch.

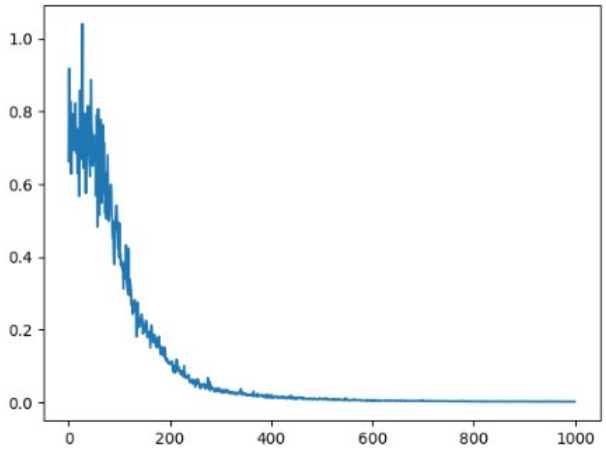



FIGURE 6 – Evolution de la Value Loss pour 1000 Epoch

On a bien pour les hyperparamètres choisis un algorithme fonctionnel. En effet, les deux Loss semblent tendre très clairement vers 0. On remarque que la convergence vers ces valeurs est parti-

culièrement stable. Cela peut se traduire par l'existence du Clipped Surrogate Objective qui permet de limiter les variations des Loss pour une optimisation plus stable.

### 3.3 Evolution de la Récompense pour 1000 Epoch sans Clipped Surrogate Objective

Il peut être intéressant d'observer l'évolution de cet Algorithme sans le Clipped Surrogate Pour voir si celui-ci a plus tendance à diverger. En utilisant les mêmes paramètres que précédemment, on obtient les résultats suivants :

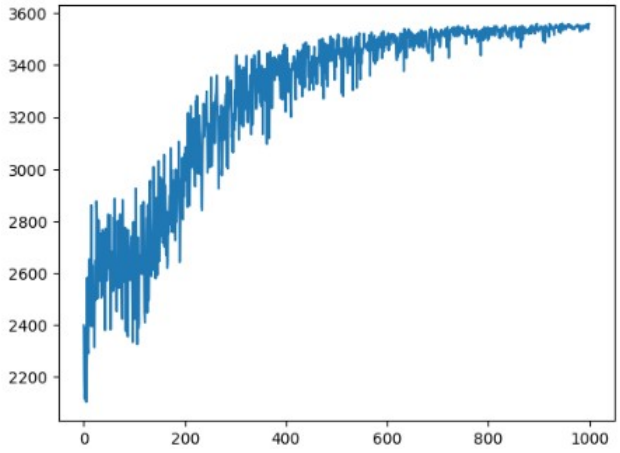



FIGURE 7 – Evolution de la Récompense pour 1000 Epoch.

De plus, pour les Policy et Value Loss, on obtient les résultats suivants :

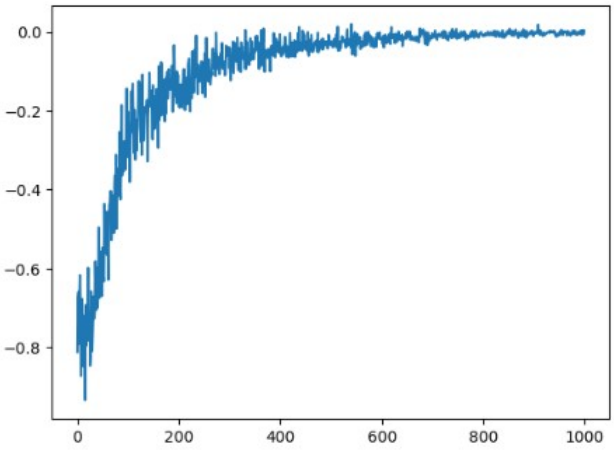



FIGURE 8 – Evolution de la Policy Loss pour 1000 Epoch.

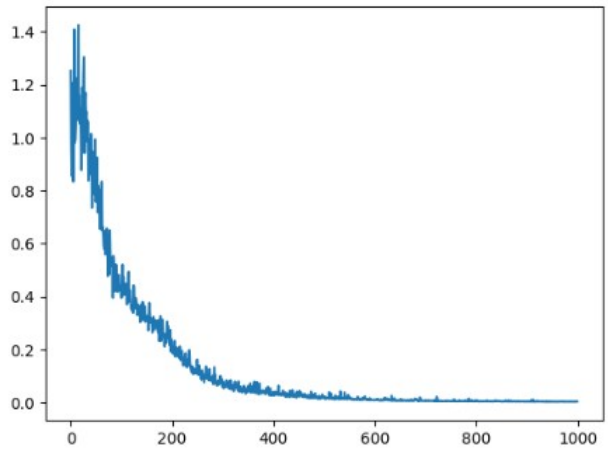



FIGURE 9 – Evolution de la Value Loss pour 1000 Epoch

Nous remarquons tout naturellement que la variance des résultats est plus grande.D'où l'importance d'avoir utilisé un clipped Surrogate Objective dans cet algorithme. De plus, l'algorithme

semble converger plus rapidement en étant clipped tout en diminuant la variance. C'est donc un paramètre important à prendre en compte dans l'algorithme PPO.

### 3.4 Evolution de la récompense pour différents Clipping

Nous pouvons aller encore plus loin avec cette expérience et observer les résultats de l'algorithme en fonction du niveau de clipping réalisé. Nous allons en particulier nous intéresser à la récompense moyenne pour les 10 derniers épisodes sur 200 épisodes, 2000 timesteps pour un clipping allant de 0 à 1 par pas de 0.1.
On obtient le résultat suivant :

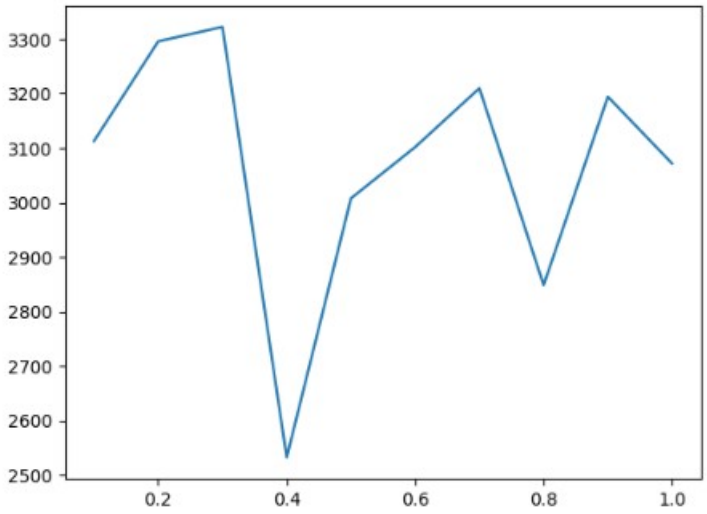



FIGURE 10 – Evolution de la Récompense pour différents clipping.

Nous pouvons voir que les résultats ne sont pas mauvais pour 200 épisodes. Cependant, on remarque que pour $\epsilon = 0.4$, le résultat est particulièrement mauvais. On peut supposer que les Loss ont commencé à diverger sur cet essai. D'où un résultat plus médiocre.

### 3.5 Impact du nombre de timesteps à Epoch fixe

Nous allons maintenant regarder l'impact du nombre de TimeStep à Epoch Fixe. Pour cette observation, nous allons diviser la récompense par 2000 puis la multiplier par le nombre de timestep. La récompense réelle que nous utilisons étant la somme de toutes les rewards sur tous les timesteps, il est important de la normaliser pour une observation cohérente puisqu'elle est croissante du nombre de TimeStep ici.

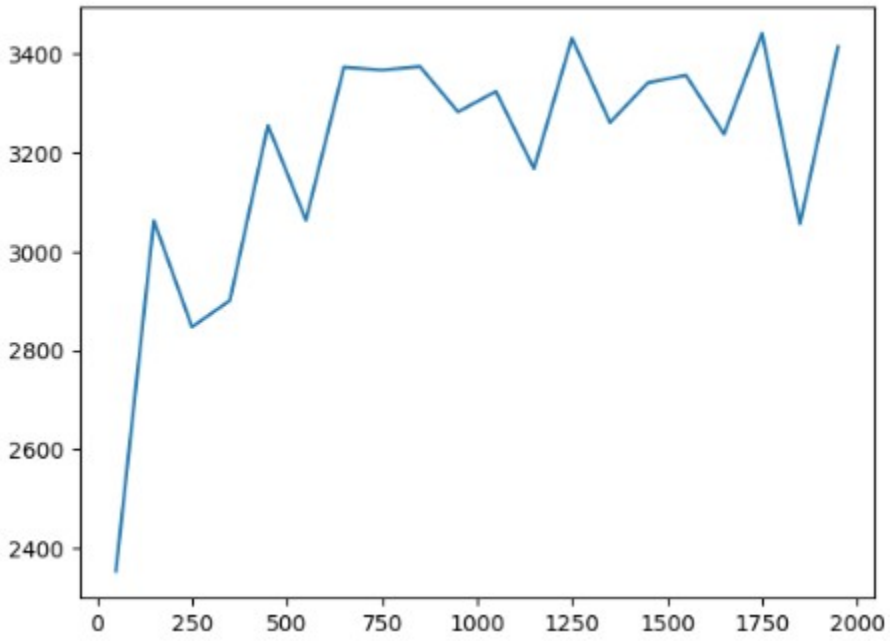



FIGURE 11 – Évolution de la récompense pour 200 Epoch et Timesteps de 50 à 2050.

Nous remarquons que plus le nombre de TimeStep est grand, plus le résultat obtenu est bon. Ce qui est logique puisque cela permet à l'algorithme d'optimisation d'avoir accès à plus de données lors de son utilisation.
On remarque aussi que pour le nombre d'épisodes choisis, on arrive assez rapidement à la Reward maximale obtenue pour ces paramètres aux alentours de 3500. Ainsi, on se rend bien compte que pour cet exercice, un grand nombre d'épisodes n'est pas forcément nécessaire.

### 3.6 Impact de la Learning rate

Lors de toutes les expériences, nous avons utilisé la même learning rate sur l'algorithme d'optimisation d'Adam (Adaptative Moment Estimation). C'est un hyperparamètre particulièrement important puisqu'il contrôle la vitesse de convergence de l'algorithme. Ainsi, une Learning Rate trop élevée peut faire diverger notre PPO. D'où l'importance d'une learning rate assez élevée pour permettre une convergence plus rapide mais assez faible pour permettre de la stabilité.

Pour illustrer ces propos, on peut faire tourner notre PPO pour différentes learning rates :

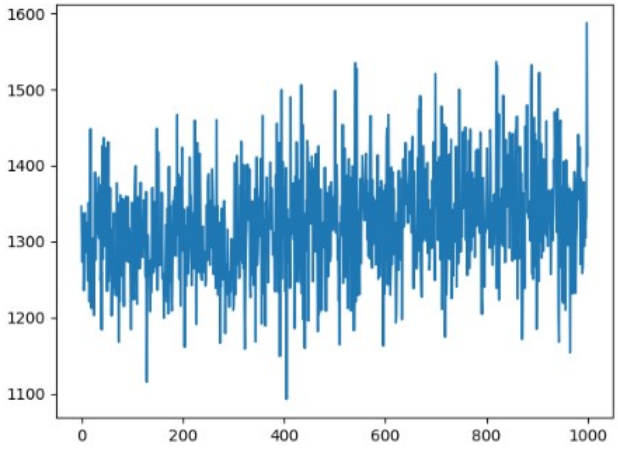



FIGURE 12 – Learning rate : 3e-6, Epoch : 1000, TimeSteps : 2000

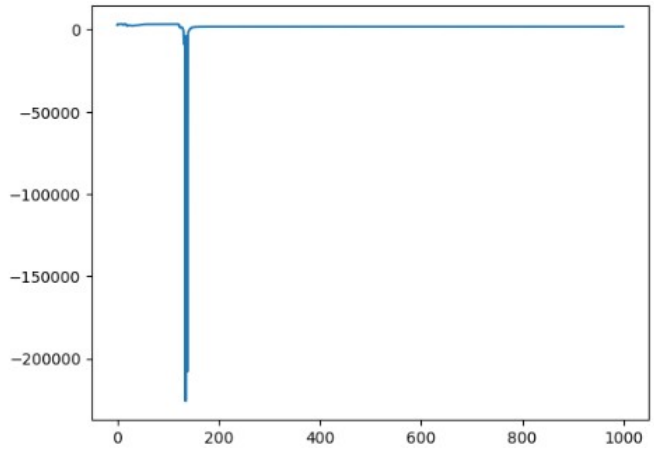
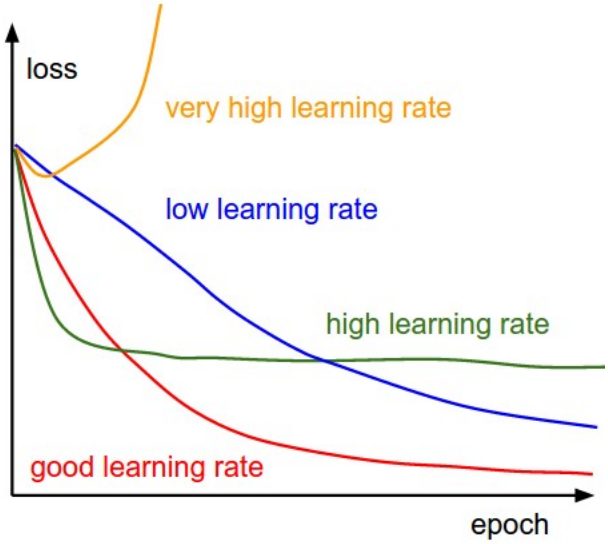



FIGURE 13 – Learning rate : 3e-2, Epoch : 1000, TimeSteps : 2000

Nous remarquons bien que pour une learning rate trop élevée, l'algorithme risque de diverger. La stabilité est une caractéristique très importante dans un algorithme de Reinforcement Learning et de Machine Learning en général.
De plus, une learning rate trop faible empêche l'algorithme d'avancer dans son optimisation.
Nous pouvons alors identifier les problèmes classiques liés à une learning rate trop grande ou trop faible :




Une learning rate trop faible implique une convergence très lente, une learning rate élevée peut faire diverger l'algorithme. On remarque que la learning rate proposée dans le papier était un bon compromis.

# 4 Conclusion

Nous avons dans le cadre de ce projet appliqué l'algorithme de la PPO à un exemple très simple. Ce qui a permis d'obtenir des résultats très satisfaisants malgré qu'une bonne partie des recommandations de l'algorithme n'ait pas été prise en compte.
Nous avons vu que c'est un algorithme avec énormément de potentiel, d'autant plus en ajoutant une KL divergence avec un coefficient adaptatif et un bonus d'entropie par exemple.

Ce Projet a été très intéressant pour moi. C'était mon premier algorithme de machine learning et il a été très challengeant pour moi.
Je pense que j'aurais pu bien mieux optimiser mon algorithme et obtenir des résultats bien plus satisfaisants, mais je reste très heureux que mon algorithme ait fonctionné.
Ce projet m'a permis de mieux comprendre une partie des concepts liés au Reinforcement Learning et fut très intéressant à mes yeux.

Un Jupyter Notebook est disponible dans le fichier github. Il permet de jouer avec les hyperparamètres à sa manière comme je l'ai fait tout au long du rapport.