Bataille Navale

Règles de la bataille navale

La bataille navale est un duel entre deux joueurs, chacun des joueurs dispose d'une grille où il positionne ses bateaux (son camp) et une grille où il sur laquelle il note ses tirs servant ainsi de repère**. Ces grilles servent de carte maritime** afin de voir si les tirs ont touché ou non et ainsi mieux observer les précédents tirs et positions des bateaux ennemis. La grille peut être de taille variable allant d'une matrice 5x5 à une matrice de 15x15 par exemple.

Les **bateaux peuvent être placé de manière verticale ou horizontale** sur la grille. Ils ne peuvent pas être mis en diagonal ou être à l'extérieur de la grille. Le nombre et la taille des bateaux peuvent être de taille variable allant d'un simple torpilleur (2 :1 ou 1 :2 selon le positionnement horizontal ou vertical) au pote avion (5 :1 ou 1 :5 toujours selon le positionnement).

Le **but du joueur est de couler tous les navires du joueur adverse** en touchant ses bateaux. Il est nécessaire de toucher chaque partie du bateau afin de le couler (e.g : le torpilleur doit être touché sur toute sa longueur de 2 cases afin d'être coulé).

Certains tirs seront considérés comme :

- **Touché**, un bateau a été touché mais n'a pas été coulé
- **Touché coulé**, un bateau a été touché sur toutes ses parties et coule
- **Dans l'eau**, le tir a fini dans l'océan et n'a touché aucun bateau

Raisonnement & Initialisation

Le raisonnement nous poussant à réaliser ce projet a été le suivant :

Existe-il une stratégie optimale afin de toujours gagner à la bataille navale ?

Nous décidâmes de répondre à cette interrogation en entrainent un modèle sur le jeu de la bataille navale. Ainsi la bataille navale peut se résumer mathématiquement par des matrices de différentes tailles remplies exclusivement de 0. Dans ces matrices se trouvent un ou plusieurs bateaux. La position de ces bateaux sera symbolisée par un 1 sur notre matrice.

```
(array([[0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0]]),
 {'cruiser': [(0, 1), (1, 1), (2, 1)]})
```

Figure 1 carte maritime et position du bateau

Ci-contre un exemple sur une grille de 6x6 comportant un seul bateau de longueur 3.

Si un tir touche un 0 il sera dans l'eau donc mauvais, si un tir touche un 1 il sera considéré comme touché et si un tir touche un 1 après que les autres 1 soient touchés il sera considéré comme touché coulé car coule le bateau ennemi. Ainsi la partie sera remporté par le joueur.

Cependant il serait trop facile de montrer cette matrice à notre ordinateur. Présenter cette matrice de placement de bateau serait comme montrer la position des destroyers en tant de guerre à l'ennemi lui donnant ainsi l'information d'où tirer afin de toucher à coup sûr nos bateaux. La grille de 0 et 1 est donc la carte maritime et est **inconnu du joueur adverse**.

Le joueur adverse quant à lui dispose de sa grille repère où il positionne et répertorie tout ses tirs en connaissant la nature de chacun (touché, touché coulé ou dans l'eau). Le joueur commencera avec une grille vide à l'étape 0 où il devra découvrir où sont cachés les bateaux de son adversaire. Les tirs touchés seront symbolisés par un « X » sur sa grille et les tirs « à l'eau » seront symbolisés par un « O » dans la grille repère.

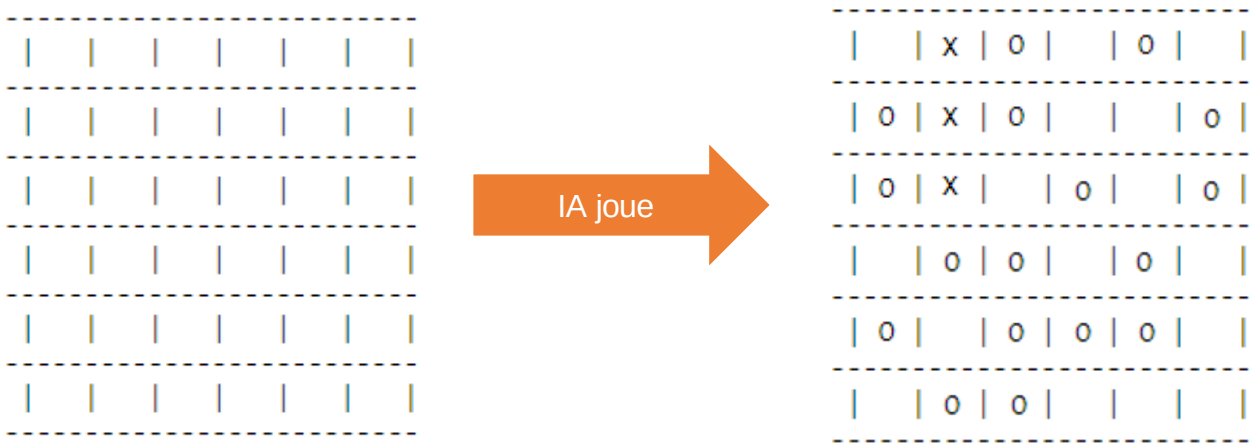


Figure 3 Phase initiale (step 0)

Figure 2 Phase finale (step 20)

Dans ce cas de figure l'IA a testé 20 possibilités pour détruire notre bateau. Nous remarquons qu'elle rata 17 fois (à l'eau) pour enfin trouver notre bateau et le couler en lui assénant 3 touchés qui le fera coulé. Cependant ce résultat n'est qu'un des nombreux cas de figure de la phase d'entrainement. Nous sommes toujours dans la phase d'initialisation et nous n'avons toujours pas convergé vers une solution optimale. **Pour l'instant notre algorithme joue presque par hasard**. A ce stade du jeu un humain serait plus optimal car, lorsqu'il touche un bateau (X) il sait que pour le couler il doit continuer à tirer autour de X afin d'endommager les autres parties du bateau et le couler, ce que notre algorithme ne sait pas (encore).

Un paramètre n'est toujours pas présent afin de faire apprendre à notre ordinateur. En effet en processus de décision markovien nous avons la trinité **Situation**, **Action** et surtout **Récompense**. La situation est notre grille repère actuelle, l'action est le tir de torpille cependant pour que notre algorithme apprenne il faut lui donner des récompenses en fonction de ses actions mais surtout aux résultats de ces dernières. Avec ces récompenses l'ordinateur optimisera ses actions afin de maximiser ses récompenses.

Rappelons que notre but est de **couler les bateaux ennemis avec un minimum de torpilles** donc de minimiser nos actions en maximisant nos récompenses. Nous pouvons représenter notre problème avec le terme suivant :

$$max_{a'} Q(s', a')$$

Nous utiliserons ici le Q-learning qui est un algorithme d'apprentissage basé sur la valeur. En effet l'ordinateur apprend la politique optimale à partir des antécédents de ses actions avec l'environnement. Pendant l'initialisation la fonction Q est donnée arbitrairement et au fil de l'entrainement l'ordinateur observe la récompense et le nouvel état. Nous notons s' et a' car nous apprenons de la meilleure action possible dans la situation s'. En d'autres termes on test et regarde quelles sont les meilleures actions dans s' afin de choisir le a' donnant le plus de récompense (maximisant Q).

Pour minimiser le nombre de torpilles tirées nous avons besoin d'émettre une pénalité. Ainsi **pour chaque action** nous réduisons la récompense de 1. L'ordinateur sera obligé de minimiser le nombre de torpille tirées pour maximiser ses récompenses. Récompense = R= -1

Si l'ordinateur effectue des actions interdites (tirer en dehors de la grille, tirer sur une case déjà touchée) il se verra attribuer une pénalité plus grande pour qu'il ne fasse jamais cette action. R= -12

Si l'ordinateur touche un bateau ennemi il obtiendra 6 points. Il est important de souligner qu'un « touché » et un « touché coulé » apporte les mêmes points. Dans certaines règles où plusieurs parties sont nécessaires afin de déterminer un gagnant le comptage de points par bateau coulé est obligatoire afin de comptabiliser les points et voir qui a coulé le plus de bateau. Cependant dans notre cas les adversaires ne s'affrontent seulement sur une partie où celui qui a coulé le bateau ennemi est gagnant. Notre stratégie donne donc des « touchés » et « touchés coulés » égaux en termes de points. R=6

Finalement **si le jeu est fini** et que tous les bateaux ont été coulé l'ordinateur sera grandement gracié. R=36

Il est important de remarquer que, via ce système de récompense, nous souhaitons terminer la partie avec un minimum de coup afin de maximiser nos récompenses. Plus la partie est longue, plus l'ordinateur rate de coup et moins grande sera sa récompense. (Voir le code Jupyter Notebook)

Condition	Effet sur la récompense R	Total de R
Pour chaque action (a+=1)	-1	-1
Action interdite	-12	-12 = -12
Bateau « touché » et/ou « coulé	+ 6	-1 + 6 = 5
Partie terminée	+36	36 + 5 = 41

Figure 4 Tableau récapitulatif des récompenses et pénalités

Nous avons utilisé l'algorithme **« Asynchronous Methods for Deep Reinforcement Learning (A3C)** » car très efficace pour l'entrainement des jeux. Différents tests ont été effectué sur des anciens jeux-vidéos Atari (entreprise de développement de jeux-vidéo) et l'algorithme a été particulièrement puissant et a convergé bien plus vite que les autres. L'algorithme utilise deux réseaux de neurones, appelés critique et acteur. Le critique estime les valeurs Q tandis que l'acteur met à jour la distribution de la politique dans le sens suggéré par le critique. Nous avons utilisé le modèle A2C qui est une légère modification où le critique et l'acteur travaillent simultanément au lieu de l'un après l'autre le rendant plus rapide (voir bibliographie). La partie Asynchrone a été retiré pour être maintenant Synchrone.

Notre algorithme étant fini il nous reste plus qu'à établir nos différents paramètres propres à la bataille navale.

Nous avons décidé de travailler avec une grille de 6x6 et un bateau de longueur 3. Cette petite grille est dû la puissance de calcul nécessaire au bon fonctionnement de l'algorithme. Nous travaillons exclusivement sur nos ordinateurs portables étant un frein à des algorithmes gourmands en puissance de calcul. Dans notre cas l'apprentissage de la bataille navale est de l'ordre du quadratique **O(x²)** lors d'augmentation de la taille de la grille. Plus la grille est grande et plus l'algorithme sera demandeur en puissance de calcul et mettra du temps à converger. Le choix arbitraire de 6x6 a été effectué car était le meilleur compromis entre ce que nous voulions faire et les capacités calculatoires de nos machines. Quant au bateau nous avons décidé d'en choisir qu'un seul car nous estimions que deux étaient trop.

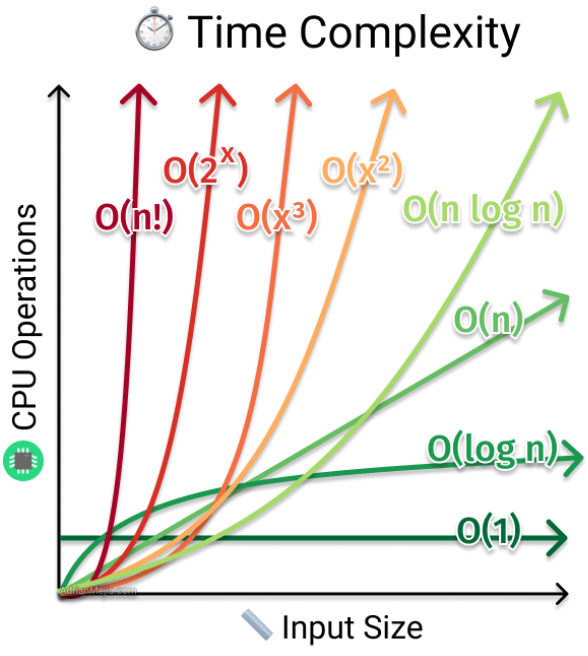


Figure 5 Différents types de time complexity

Convergence & Résultats

Nous avons entrainé notre modèle sur 15millions de parties, ce procédé a pris à l'ordinateur 10heures. Nous observons une augmentation continuelle des récompenses. Nous sommes passé de -80 points en moyenne à 15 points soit une augmentation de 95points entre le début et la fin. Cependant nous observons que l'ordinateur était toujours en train d'apprendre lorsque la dernière partie s'arrêta. Pour des raisons matérielles nous avons préféré nous arrêter à 10h d'entrainement (15m parties) empêchant ainsi d'avoir l'apparition d'une stratégie optimale et voir notre ordinateur convergé vers une politique optimale.

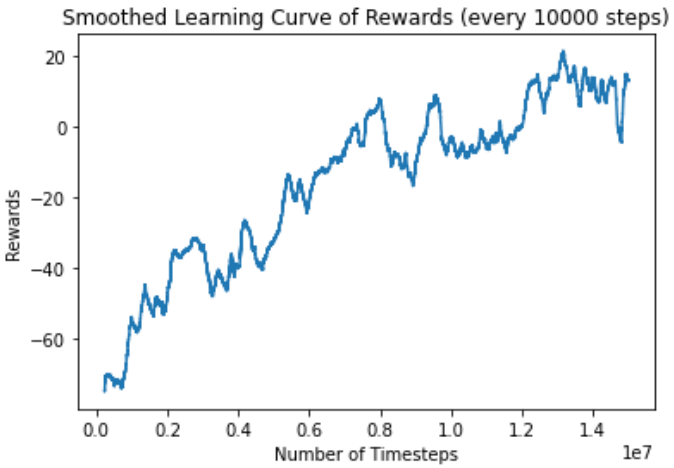
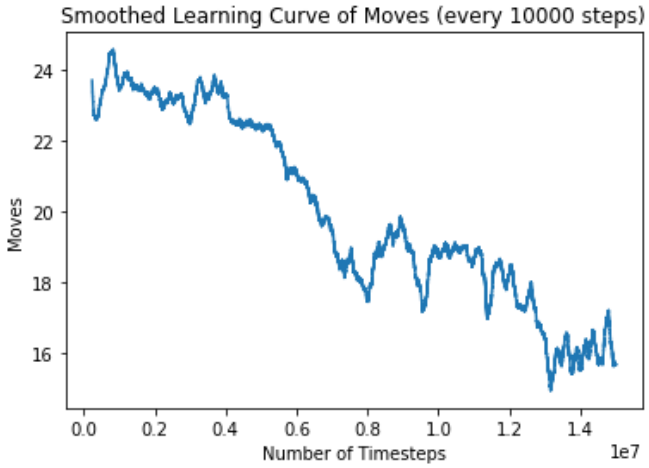


Figure 6 Récompenses obtenues au fil des parties jouées



Nous voyons une diminution du nombre de moyen de coup joué allant de 24 coups à 15 soit une diminution de 9 coups pour finir une partie.

Ainsi nous remarquons que notre ordinateur a appris et a augmenté ses récompenses et a eu besoin de moins de coups pour gagner.

Figure 7 Nombre de coups moyens pour gagner au fil des parties jouées

Malheureusement nous ne pouvons pas répondre à notre problématique de départ dû à nos limitations techniques. Il est probablement possible d'obtenir une convergence plus rapide en modifiant notre système de récompense. Un système de récompense plus indexé sur le nombre de torpille tirée évitera à notre acteur de trop se « balader » sur la grille le poussant ainsi à se concentrer davantage sur la découverte et destruction des bateaux. Une autre piste d'amélioration serait de changer d'algorithme et tester cet apprentissage avec d'autres algorithme qui pourrait être plus efficace.

Dans le cadre de ce projet nous avons cependant réussi à entrainer une IA à jouer à la bataille navale et s'améliorer à ce jeu en diminuant le nombre de coup nécessaire pour gagner et ainsi s'approcher de la stratégie optimale.

Bonus

Nous avons essayé un autre algorithme afin de, potentiellement observer un résultat différent mais surtout une convergence plus rapide et potentiellement voir la stratégie optimale. Ainsi nous avons troqué notre modèle A2C pour un modèle DQN. **Deap Q Network (DQN)** est un modèle de Q learning plus élaboré. Ce modèle utilise des réseaux de neurones afin de parvenir à ses fins. Étant donné que notre compréhension du modèle soit limitée nous préfèrerons nous concentrer sur les résultats de ce dernier que réellement connaitre ses mécanismes.

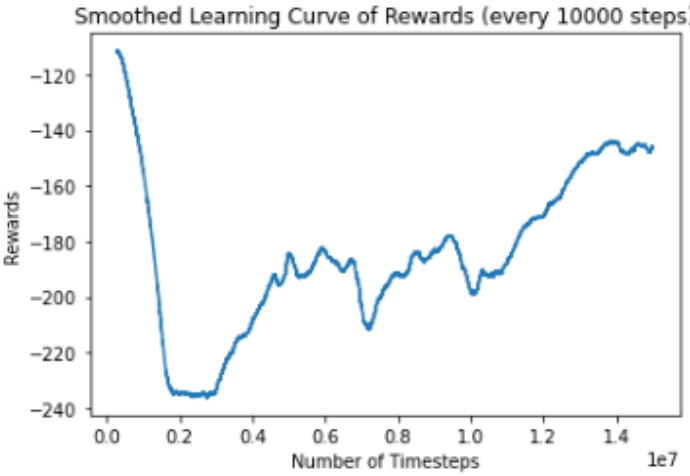


Figure 8 Récompenses obtenues au fil des parties jouées

Après 15 millions de parties (10h d'entrainement) nous observons le résultat de notre modèle. Nous remarquons immédiatement les résultats catastrophiques de ce dernier. En moyenne nous perdons 50points de récompense et le nombre de coup pour terminer une partie croit. En d'autres termes l'ordinateur devient plus mauvais.

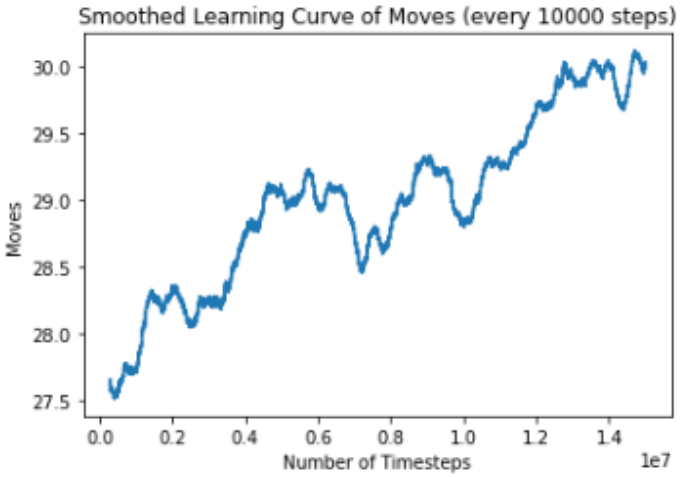


Figure 9 Nombre de coups moyens pour gagner au fil des parties jouées

Évidemment nous n'utiliserons pas ce modèle pour notre bataille navale à la vue de ses médiocres résultats. Le but de l'utilisation de ce modèle était d'essayer différents types de modèles afin de voir quel modèle donnait une convergence plus rapide.

Nous avons ensuite essayé un autre modèle, le **Trust Region Policy Optimization (TRPO)** afin de voir si notre ordinateur pouvait apprendre plus vite. Nous avions émis deux axes d'amélioration, le premier une amélioration de modèle, le deuxième étant une **modification des paramètres de récompenses**.

Nous avons ainsi fait tourner notre algorithme TRPO sur 2 combinaisons de paramètres. Suite à l'entrainement de A2C nous avions émis l'hypothèse qu'une récompense réduite de 1 à chaque coup était trop peu car était trop permissif et nous avions des situations où notre ordinateur se « baladait » sur la grille au lieu de se concentrer sur le bateau. Nous avons donc fait tourner une deuxième fois notre modèle sur une récompense réduite de 3 à chaque coup afin de presser d'avantage notre ordinateur.

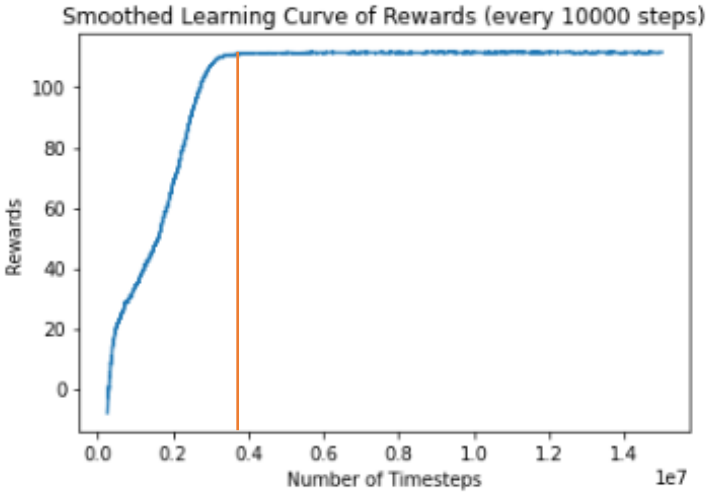


Figure 11 Récompense du TRPO avec -1 par coup

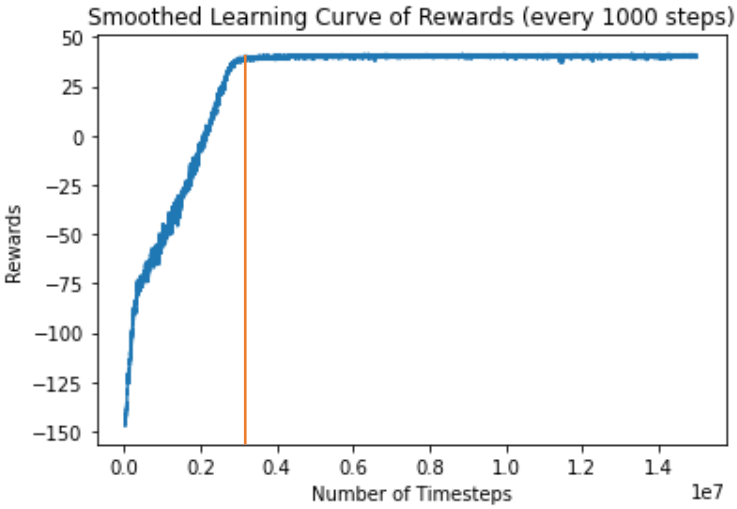


Figure 10 Récompense du TRPO avec -3 par coup

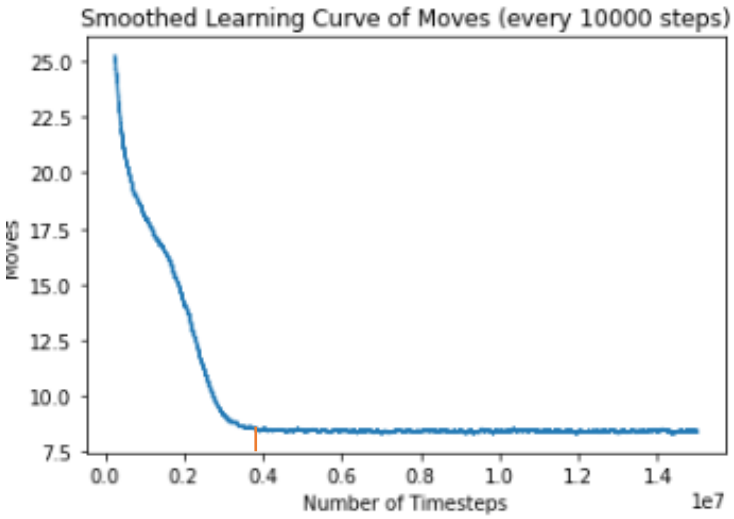


Figure 13 TRPO, nombre de coup pour terminer une partie avec -1

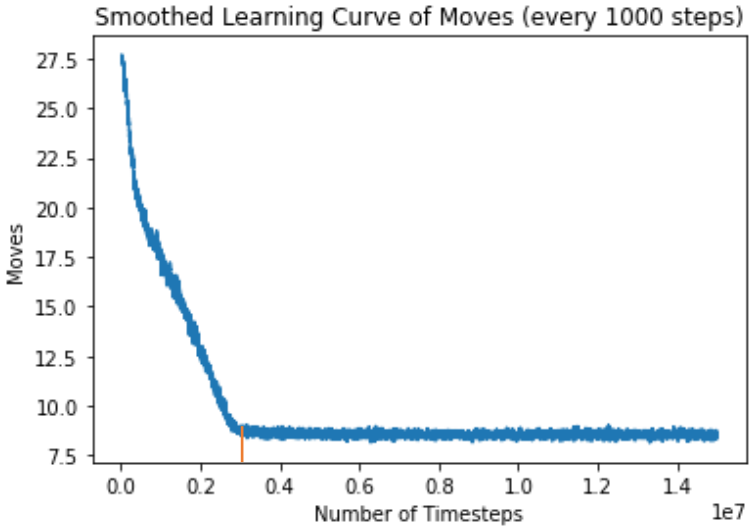


Figure 12 TRPO, nombre de coup pour terminer une partie avec -3

Avant toutes choses nous devons préciser que les modèles ont réaliser 15 millions de parties et ont convergé à environ 300.000 parties. La comparaison des récompenses obtenues est inutile, -3 par coup donnera un désavantage certain au deuxième algorithme. Nous voyons que le TRPO converge et est donc plus efficace que notre algorithme de base, le A2C pour entrainer un ordinateur sur la bataille navale.

	TRPO avec -1 par coup	**TRPO avec -3 par coup**
Nombre d'occurrences lors de la Convergence	390.000	300.000

Figure 14 Tableau récapitulatif de convergence des modèles

Le changement de politique de récompense a porté ses prix. La convergence se fait 90.000 occurrences (environ 40min avec 8GB de RAM) plus vite.

Notre modèle converge enfin nous donnant la possibilité de répondre à notre hypothèse de base qui était « Existe-il une stratégie optimale afin de toujours gagner à la bataille navale ». **Oui il existe une stratégie optimale** avec une grille de 6x6 et un bateau de 3x1 où en 8 coups en moyenne il est possible de terminer une partie.

Pour conclure cette partie « Bonus » qui était ni plus ni moins qu'une extension du projet où nous avons expérimentés différents types de paramètres nous dirons que notre combinaison de paramètres est encore à optimiser afin d'obtenir une convergence plus rapide. Il en est de même pour l'algorithme, bien que TRPO ai apporté les meilleurs résultats il est fort probable qu'un autre modèle performe mieux. Cependant nous avons trouvé la méthode d'apprentissage par renforcement très incertain à la vue du côté « tâtonnement » de ce dernier. **Il est impossible de calculer la combinaison de paramètres optimales ainsi que le meilleur modèle pour un jeu**, ici la bataille navale. Notre progression s'est faite par expérimentations (entrainement de différents algorithmes pendant 10h) ainsi que de rhétorique (nous trouvions que -1 par coup était trop clément avec le modèle). Cependant nous avons pu trouver une réponse à notre hypothèse et avons découvert comment fonctionne l'apprentissage ce qui nous réjouis.

Bibliographie

- **Projet dont nous nous sommes inspirés** : « https://towardsdatascience.com/an-artificial-intelligence-learns-to-play-battleship-ebd2cf9adb01 »

- "**Asynchronous Methods for Deep Reinforcement Learning**" Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, Koray Kavukcuoglu

- **Pourquoi A2C et non A3C** : stable baselines est un package développé par des chercheurs français et américains comportant certains algorithmes https://stablebaselines.readthedocs.io/en/master/modules/a2c.html

- **"Playing Atari with Deep Reinforcement Learning"** Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller

- **« Trust Region Policy Optimization » (TRPO)** John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, Pieter Abbeel