

Exploration de la redistribution de récompenses en apprentissage par renforcement

Francois-Alexandre Tremblay, Luc Coupal et William-Ricardo Bonilla-Villatoro
Université Laval

{francois-alexandre.tremblay.1, luc.coupal.1, william-ricardo.bonilla-villatoro.1}@ulaval.ca

Abstract

Ces travaux s'intéressent à l'usage des réseaux de neurones à connexions récurrentes dans la redistribution de récompenses avec retard en apprentissage par renforcement, tel que proposé par Arjona-Medina et al. [1]. Des fonctions de redistribution sont apprises préalablement à partir de jeux de données de séquences pour ensuite s'intégrer aux structures des agents apprenants et ainsi accélérer leur apprentissage. Nous réalisons des expériences originales dans des environnements où les récompenses sont retournées à retardement. Nos travaux montrent le potentiel de cette technique dans certains contextes pour accélérer la convergence des algorithmes d'apprentissage par renforcement. Le code source est disponible sur notre GitHub 

1 Introduction

L'**apprentissage par renforcement** (*RL*), un sous-domaine de l'intelligence artificielle, est un problème dans lequel un agent apprend des séquences de décisions optimales par interaction avec son environnement à travers un processus « d'essais-erreurs ». Dans cette configuration, l'agent reçoit des récompenses de son environnement en fonction de ses décisions qui l'incitent à développer une politique de sélection d'actions (i.e. comportement) optimales [2]. Néanmoins, dans plusieurs problèmes réels [3], les récompenses sont souvent épisodiques ou éloignées dans le temps. La réception de récompenses à retardement peut à ce moment ralentir exponentiellement la convergence des algorithmes de *RL*, un problème particulièrement récurrent dans ce sous-domaine [1]. En effet, les algorithmes basés sur les approches Temporal Difference et Monte Carlo requièrent nettement plus de mises à jour pour réduire respectivement le biais et la variance de l'espérance des récompenses estimées futures. Cette quantité, qui porte aussi le nom de *Q-value*, constitue une composante centrale de plusieurs algorithmes de *RL*. Elle est aussi fréquemment estimée à partir de réseaux de neurones.

Dans ce rapport, nous explorons le potentiel de la méthode **RUDDER** (*Return Decomposition for Delayed Rewards*), proposée par Arjona-Medina et al. [1], qui s'intéresse à approximer la *Q-value*, à partir d'un réseau LSTM, dans des environnements où les récompenses sont obtenues à retardement. Le réseau est entraîné préalablement sur des séquences de décisions pour ensuite être combiné aux réseaux de neurones pleinement connectés de l'agent. Durant l'entraînement, la politique de sélection d'actions bénéficie ainsi des récompenses redistribuées du LSTM pour se mettre à jour et accélérer sa convergence vers des prises de décisions optimales.

Notre rapport se divise en quatre sections. La première porte sur la définition de deux concepts clés : le paradigme du *RL* et la redistribution de récompenses. La seconde section s'intéresse à la méthodologie et plus particulièrement aux environnements, aux architectures de réseaux, à l'entraînement du LSTM et aux données utilisées. Dans la troisième section, nous réalisons des expériences empiriques sur des environnements différents de Arjona-Medina et al. [1] afin de valider notre implémentations des algorithmes. Finalement, nous discutons des résultats.

2 Définition

Cette section porte sur les concepts d'apprentissage par renforcement et de redistribution de récompenses.

2.1 Apprentissage par renforcement

Le problème de *RL* est formulé sous la forme d'un processus de décisions séquentielles de Markov noté $(\mathcal{S}, \mathcal{A}, P, r, s_0)$ avec l'espace d'état \mathcal{S} , l'espace des actions \mathcal{A} , les probabilités de transition $P(s_{t+1}|s_t, a_t)$ *, la fonction de récompenses $r(s_t, a_t)$ et l'état initial s_1 [2]. La fonction de récompenses exprime la rétroaction retournée par l'environnement après avoir exécuté l'action a_t dans l'état s_t . Selon cette configuration, l'agent cherche à maximiser l'espérance des récompenses cumulatives en priorisant les actions qui font augmenter ses gains à long terme. Il tente ainsi d'apprendre la politique de sélection d'actions optimales π sur des séquences ou trajectoires de paires états-actions $\tau \triangleq (s_1, a_1, \dots, s_T, a_T)$ [2]. Formellement, l'objectif standard en *RL* est :

$$\max_{\pi} \mathbb{E}_{\tau \sim \pi_{\theta}(\tau)} \left[\sum_{t=1}^T r(s_t, a_t) \right]$$

Intuitivement, l'agent, qui se retrouve dans un certain état, prend des décisions à chaque pas de temps causant un changement d'état de l'environnement (Figure 1). Celui-ci reçoit des récompenses de l'environnement qui l'encouragent à prendre ou à ne pas prendre certaines décisions.

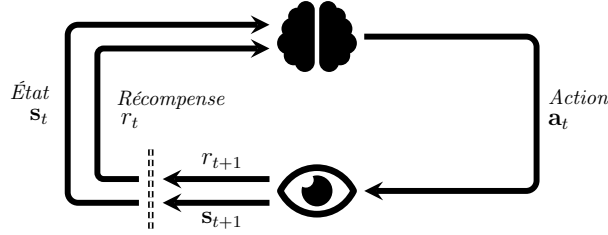


FIGURE 1 Boucle de rétroaction

Tel que mentionné précédemment, il est fréquent en *RL* que l'efficacité d'un algorithme à optimiser la politique de l'agent repose en tout ou en partie sur sa capacité à estimer la *Q-value* [2]†. Cette valeur, généralement estimée, indique à l'agent à quel point il est bénéfique de choisir une action dans certains états de l'environnement. Durant l'apprentissage, l'agent doit ainsi découvrir les actions, selon les états, qui sont plus profitables à long terme en opposition à celles où la récompense instantanée est importante. Malheureusement, les *Q-values* sont plus difficiles à déterminer puisqu'elles sont estimées et ré-estimées durant la durée de vie de l'agent [2] contrairement aux récompenses reçues instantanément par l'environnement. La tâche devient par ailleurs exponentiellement plus ardue lorsque les récompenses sont reçues à retardement [1]. Le lien entre les paires d'états-actions et la récompense devient davantage indirect.

2.2 Redistribution de récompenses

La redistribution de récompenses est une procédure qui consiste à redistribuer, à travers chaque séquence $(s_1, a_1, \dots, s_T, a_T)$, les récompenses de l'environnement. Autrement dit, dans les problèmes où la récompense est envoyée seulement à la fin de la trajectoire (ex. gagner une partie d'échec), cette méthode vise à gratifier l'agent, après chacune de ses actions posée dans un état, par une estimation des récompenses. La relation entre les paires états-actions et les récompenses devient à ce moment plus directe. Dans leurs travaux, Arjona-Medina et al. [1] suggèrent d'utiliser un LSTM pour réaliser la tâche de redistribution. Leur approche porte le nom de RUDDER.

*. Il s'agit de la probabilité conditionnelle, aussi appelé la dynamique du système, de transiter d'un état $s_t \in \mathcal{S}$ vers un état $s_{t+1} \in \mathcal{S}$ suivant l'exécution de l'action $a_t \in \mathcal{A}$ au temps t .

†. Formellement, cette valeur s'exprime par $Q(s_t, a_t) = \mathbb{E}_{\pi} \left[\sum_{k=0}^{T-t} r(s_{t+k}, a_{t+k}) | s_t = s, a_t = a \right]$

3 Méthodologie

Cette section porte sur l'information essentielle à la description de la méthodologie suivie. Compte tenu de la grande quantité de variables et d'informations sur les environnements et les hyperparamètres, nous référons le lecteur à l'annexe pour des précisions additionnelles, notamment à des fins de reproductibilité.

3.1 Environnement d'apprentissage

OpenAI Gym [4], une librairie développée en langage Python, permet de simuler des environnements et leur dynamique afin de comparer les performances des algorithmes de *RL*. Elle est fréquemment utilisée dans le domaine comme base de référence. Dans le cadre de nos travaux, nous nous servons de deux environnements : CartPole et Mountain Car. Le Tableau 1 en annexe présente les actions, les états et les récompenses de nos environnements expérimentaux.

CartPole (Figure 2) est un environnement dans lequel un pôle instable est attaché en un point sur un chariot. L'agent doit apprendre à garder le pôle en équilibre le plus longtemps possible en décidant, à chaque pas de temps, si le chariot doit se déplacer à gauche ou à droite. La trajectoire se termine lorsque le pôle a un angle de moins de 15 degrés par rapport à l'axe vertical, si la position du chariot sort du cadre de référence ou si le nombre de pas de temps atteint la valeur de 500. Dans l'environnement standard de CartPole, l'agent reçoit une récompense de +1 à chaque pas de temps où l'état du système n'est pas terminal. Dans le cadre de notre rapport, nous modifions cet environnement afin que l'agent reçoive ses récompenses, à retardement, à la toute fin de la trajectoire.

Mountain Car (Figure 3) est un environnement qui a pour objectif d'amener une voiture, positionnée initialement au creux d'une vallée, au dessus d'une colline. À chaque pas de temps, l'agent peut décider de faire avancer ou reculer la voiture ou de rester au neutre. Chaque décision implique de recevoir une récompense de -1 jusqu'à atteindre 200 pas de temps ou lorsque le véhicule dépasse le drapeau sur la colline de droite. En fait, la difficulté du problème réside dans l'incapacité du moteur de la voiture à générer assez de vitesse pour monter d'un seul coup la colline. La seule façon de résoudre l'environnement est ainsi de reculer et avancer pour accumuler du momentum, et ce, le plus rapidement possible pour éviter les récompenses négatives après chaque action. Bien que l'environnement soit simple, la tâche n'est pas moins complexe compte tenu que la rétroaction reçue à chaque pas de temps t est la même du début jusqu'à la fin de la trajectoire. L'agent doit ainsi explorer l'environnement pour estimer les gains à long terme associés aux états et aux actions. Tout comme pour CartPole, nous retournerons les récompenses seulement à l'atteinte de l'état terminal.

De plus, les séquences de CartPole et Mountain Car sont nécessairement de différentes tailles. Dans le cadre de notre implémentation, nous utilisons les fonctions `packed_padded_sequence` et `pad_packed_sequence` de PyTorch [5] pour récupérer adéquatement leur information et réduire le temps de calcul.

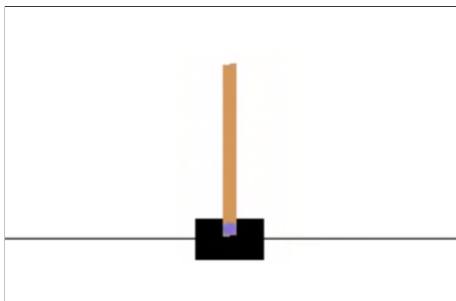


FIGURE 2 Environnement CartPole

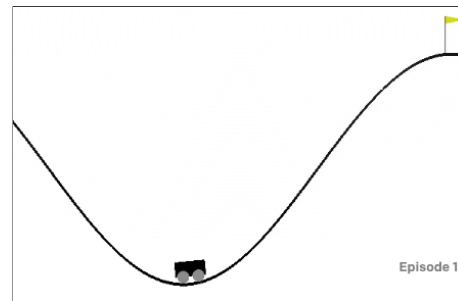


FIGURE 3 Environnement Mountain Car

3.2 Architecture des réseaux de neurones

Le **LSTM** prend en entrée des séquences «packed» de paires d'états-actions ; c'est-à-dire des séquences où chaque état et action sont concaténés. Les actions, contrairement aux états, sont encodées sous la

forme de *one-hot vectors* afin d’éviter une fausse relation d’ordre. En sortie, le LSTM retourne la valeur estimée pour ensuite la passer à une couche linéaire qui renvoie la récompense redistribuée.

Puisqu’il s’agit d’un contexte de régression, le réseau repose sur la métrique de perte quadratique comme technique d’optimisation. Le LSTM optimise en fait une tâche principale et une tâche auxiliaire de manière simultanée. La première tâche consiste à prédire la valeur d’action Q à la fin de chaque séquence alors que la seconde vise à prédire cette valeur pour chaque pas de temps de la trajectoire. Cela revient à rétro-propager la récompense finale à travers le temps. Cette tâche est donc beaucoup plus difficile à apprendre pour le réseau récurrent puisqu’il doit estimer un futur inconnu. Selon les auteurs Arjona-Medina et al. [1], il n’est toutefois pas nécessaire que le LSTM retourne une prédiction parfaite pour bien décomposer la récompense. La prédiction finale est davantage importante. Conséquemment, nous avons pondéré la tâche auxiliaire de 0.5 afin de considérer cet aspect.

Pour réaliser l’apprentissage de l’agent, nous avons développé notre propre version de **PPO** (*Proximal Policy Optimization*), un algorithme d’apprentissage stochastique utilisé par les auteurs de RUDDER. Notre implémentation, qui repose sur les travaux de Schulman et al. [6] et de Brockman et al. [4], comporte deux réseaux pleinement connectés. Le premier consiste à estimer la valeur d’état $V(s_t)$ [‡] alors que le deuxième permet de dériver la politique π à partir de cette valeur. L’entraînement de l’agent est réalisé de manière successive, à l’aide de la prédiction du LSTM pré-entraîné, jusqu’à convergence ; c’est-à-dire, lorsque l’agent ne semble plus apprendre.

3.3 Données d’entraînement

Le réseau LSTM requiert des séquences d’entraînement pour optimiser ses paramètres par la descente du gradient stochastique. Dans le cadre de notre rapport, nous générons nos propres séquences via des politiques entraînées à partir de l’algorithme PPO sur les environnements CartPole et MountainCar. Nous commençons par sélectionner les politiques sous-optimales, qui occasionnent moins de récompenses, et les politiques optimales, qui en occasionnent davantage. Le Tableau 2 montre les politiques optimales et sous-optimales sélectionnées pour chaque environnement en référence au Figure 4 et Figure 5. Par la suite, les politiques choisies servent à produire un ensemble de 2875 séquences sous-optimales et un autre, de même taille, de séquences optimales. En fait, on peut voir ces données comme des trajectoires clés permettant d’intégrer de la connaissance a priori du problème afin d’accélérer l’apprentissage de l’agent. Dans des problèmes réels, ces trajectoires pourraient provenir de systèmes-experts ou de démonstrations humaines [7]. Enfin, les récompenses des séquences générées sont cumulées et sauvegardées pour pouvoir les utiliser dans notre problème de délai.

3.4 Entraînement du LSTM

La méthode RUDDER est particulièrement difficile à entraîner compte tenu de la grande quantité d’hyperparamètres. Un choix particulièrement rigoureux de ceux-ci est nécessaire à l’obtention de résultats valides. Pour répondre à cette limite, nous avons élaboré un algorithme de recherche aléatoire d’hyperparamètres [8] [§] permettant de tester plusieurs réseaux LSTM afin d’isoler celui ayant la meilleure performance. Nous avons d’abord raffermi le champ de recherche des hyperparamètres par des explorations manuelles pour ensuite choisir de mettre à l’épreuve les hyperparamètres suivants : taille de la batch, nombre de cellules cachées, pourcentage de trajectoires optimales, taux d’apprentissage et *weight decay*. Nous utilisons également un générateur de nombres aléatoires pour sélectionner le plus possible les mêmes données d’entraînement et ainsi faciliter la comparaison des réseaux générés.

Lors de l’entraînement du LSTM, les séquences sont échantillonnées aléatoirement dans l’ensemble de données. Le taille de l’ensemble et le pourcentage de trajectoires optimales sont des hyperparamètres modifiables selon l’expérience. Sur le total des séquences obtenues, nous en conservons 80% pour la phase d’entraînement et 20% pour l’évaluation du modèle.

[‡]. Cette valeur représente à quel point il est bénéfique pour l’agent de se retrouver dans un certain état sur le long terme. Formellement, elle se définit comme suit : $V(s_t) = \mathbb{E}_\pi \left[\sum_{k=0}^{T-t} r(s_{t+k}, \mathbf{a}) | s_t = s \right]$ [2]

[§]. Bergstra and Bengio [8] démontrent qu’une recherche aléatoire d’hyperparamètre est plus efficace qu’une recherche en grille et au moins aussi efficace pour trouver de bons hyperparamètres.

À la fin de l'entraînement, les paramètres du LSTM sont transférés à la classe `LSTMCell` de Pytorch [5]. Ce détail d'implémentation est important compte tenu que la classe `LSTM` a le désavantage de retourner seulement l'état caché final contrairement à la classe `LSTMCell` qui permet de renvoyer les états cachés intermédiaires d'une séquence. En plus de bénéficier de l'optimisation du LSTM en entraînement en évitant les structures itératives du langage Python, l'agent peut accéder à la récompense estimée Q après chaque décision.

4 Résultats empiriques

Cette section présente les résultats empiriques de nos expériences sur les environnements `CartPole` et `Mountain Car`.

4.1 Génération de séquences

À titre de preuve de fonctionnement de notre algorithme d'apprentissage et dans le but de générer nos séquences, nous avons déployé *PPO* sur les environnements standards de OpenAI Gym. Les figures Figure 4 et Figure 5 montrent les récompenses moyennes obtenues durant l'apprentissage. Il est à noter que de fortes fluctuations sont monnaie courante en *RL* durant l'apprentissage [2] d'autant plus que la politique obtenue par *PPO* est stochastique. Tel que mentionné dans la sous-section sur les environnements, nous remarquons que `Mountain Car` est plus difficile à résoudre que l'environnement `CartPole`. La Tableau 2 en annexe est fourni à des fins de reproductibilité.

4.2 Entraînement de RUDDER

Nous entraînons ensuite le LSTM à partir des séquences générées à l'étape précédente. La Figure 6 et la Figure 7 montrent les résultats de la perte empirique en phase d'entraînement et de test selon les hyperparamètres en annexe (Tableau 3). Le réseau converge vers une solution dans les deux environnements. Il est aussi à noter qu'aucune transformation n'a été appliquée sur les récompenses cumulatives afin de conserver l'ordre de grandeur et, conséquemment, éviter des valeurs de sortie du réseau LSTM trop grandes ou trop petites pour la structure de l'agent.

4.3 PPO avec RUDDER

Les figures ci-dessous (Figure 8 et Figure 9) présentent les courbes d'apprentissage de l'agent avec et sans la méthode *RUDDER* dans des configurations où les récompenses sont obtenues à retardement. À noter que dans la première configuration, l'agent reçoit ses récompenses instantanément du LSTM plutôt qu'avec du retard.

Avec la redistribution de récompenses (courbe bleue), l'agent converge vers une politique optimale dans l'environnement `CartPole` seulement. Dans l'environnement `Mountain Car`, il réussit à atteindre de justesse le sommet de la colline en moins de 200 pas de temps dans certaines situations. Sans la contribution du LSTM (courbe orange), on remarque que la tâche est nettement plus difficile à apprendre. Cela suggère que la rétro-propagation des récompenses cumulatives à travers le temps est plus ardue pour l'algorithme *PPO*.

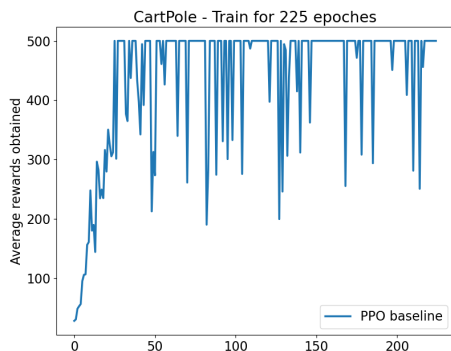


FIGURE 4 `CartPole` - Politiques générées

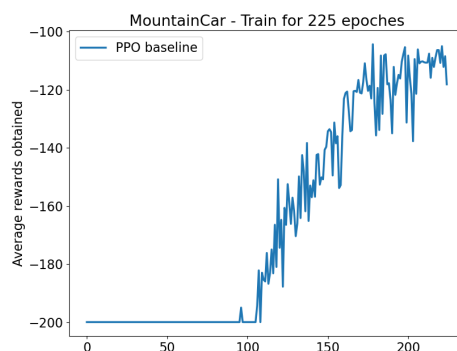


FIGURE 5 `Mountain Car` - Politiques générées

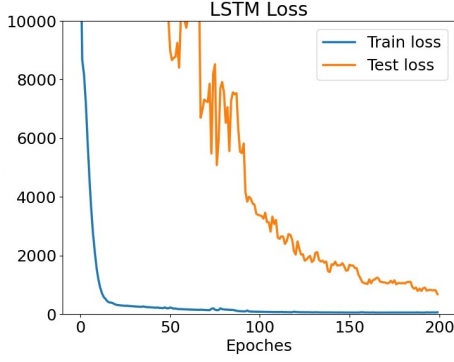


FIGURE 6 CartPole - Perte empirique

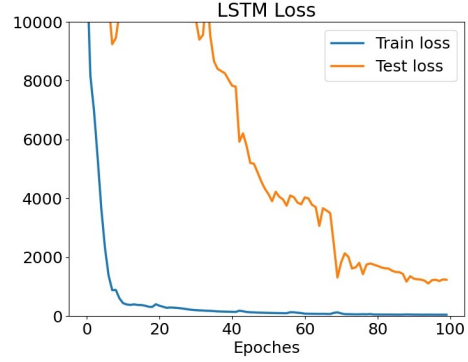


FIGURE 7 Mountain Car - Perte empirique

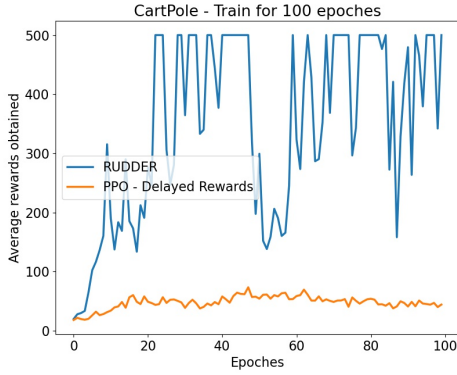


FIGURE 8 CartPole - PPO et RUDDER

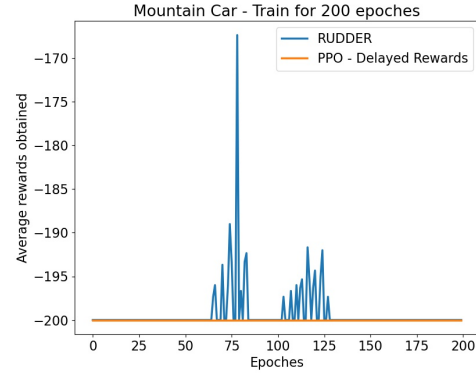


FIGURE 9 Mountain Car - PPO et RUDDER

5 Discussion

Nos expériences mettent en lumière le potentiel de la redistribution de récompenses dans l'environnement de CartPole. De façon intéressante, la courbe de récompenses moyennes issues de la Figure 4 ressemble à celle de la Figure 8. Ce résultat suggère que la méthode RUDDER pourrait peut-être aider dans les problèmes où la fonction de récompenses est difficile à définir et advenant la disponibilité de données séquentielles. Ce type de problème est récurrent en *RL* et constitue un champ de recherche en soi [9].

Malgré des recherches manuelles et aléatoires d'hyperparamètres, nous n'avons pas réussi à trouver de configurations optimales pour l'environnement Mountain Car. Toutefois, nous pensons que cela est notamment attribuable à la particularité de ce dernier. Intuitivement, si une force est appliquée en sens inverse au momentum de la voiture à la suite d'une action de l'agent, la vitesse diminue très fortement. La récompense espérée, qui est envoyée par le LSTM, doit ainsi être particulièrement précise pour éviter d'induire l'agent en erreur et d'occasionner un freinage catastrophique. Le LSTM entraîné n'atteindrait donc pas la précision nécessaire dans le cadre de cet environnement. Une deuxième explication peut également reposer sur le choix d'hyperparamètres spécifiques. Dans tous les cas, ces justifications s'ajoutent aux limites identifiées par Arjona-Medina et al. [1].

Par ailleurs, les mêmes auteurs donnent peu d'informations sur les données utilisées dans le cadre de l'entraînement du LSTM. Par exemple, est-ce que le réseau est entraîné à partir de séquences optimales, sous-optimales ou une mixture des deux ? Quelle est la proportion de trajectoires optimales ? Combien de séquences au total font partie de l'ensemble d'entraînement ? Selon nos observations, il est nécessaire de détenir des observations optimales et sous-optimales. Cela permet d'explorer à la fois les espaces d'états-actions payants et non payants pour mieux orienter l'agent. Toutefois, un plus grand nombre de séquences optimales semblent donner de meilleurs résultats.

6 Conclusion

Notre rapport met en lumière le potentiel de l'approche RUDDER dans les problèmes de décisions séquentielles où les récompenses sont obtenues à retardement. À partir de connaissances a priori sur la tâche à accomplir, cette méthode permet d'injecter l'information nécessaire à l'apprentissage de politiques de *RL*. Nos résultats montrent que RUDDER peut parfois s'avérer utile lorsque l'agent obtient ses récompenses avec du retard. Cela est attrayant pour plusieurs problèmes où la fonction de récompenses est difficile à définir.

Malgré son potentiel, la redistribution de récompenses est particulièrement difficile à mettre en oeuvre et à entraîner. Déjà que l'entraînement distinct de RUDDER et PPO est ardu, leur amalgame ne fait qu'augmenter le niveau de difficulté d'apprentissage. Le choix des hyperparamètres et la connaissance de l'environnement sont des aspects cruciaux assurant la convergence des algorithmes.

Dans un prochain rapport, il serait intéressant d'étudier des environnements différents où l'espace des actions est continu par exemple. Des expériences sur d'autres classes d'environnement permettraient également d'investiguer plus en profondeur les choix d'hyperparamètres. Finalement, il serait intéressant de trouver des hyperparamètres encore plus raffinés à la résolution de l'environnement Mountain Car.

ANNEXE

| Espaces | CartPole | Mountain Car |
|----------------------|---|---|
| États | <ul style="list-style-type: none"> • Position du chariot • Vitesse du chariot • Angle du pôle • Vitesse angulaire du pôle | <ul style="list-style-type: none"> • Position sur l'axe horizontale • Vitesse de la voiture |
| Actions | <ul style="list-style-type: none"> • Gauche • Droite | <ul style="list-style-type: none"> • Avancer • Reculer • Ne rien faire |
| Récompenses cumulées | • +1 à chaque pas de temps à l'état terminal | • -1 à chaque pas de temps à l'état terminal |

TABLE 1 Environnements

| Hyperparamètres | CartPole | Mountain Car |
|---|-----------|--------------|
| Technique d'optimisation | Adam | Adam |
| Nb d'époques | 225 | 225 |
| Taux d'apprentissage | 0.01 | 0.02 |
| Dimension entrée | 4 | 2 |
| Nb couches cachées | 1 | 1 |
| Nb neurones couches cachées | 18 | 18 |
| Dimension de sortie | 1 | 1 |
| Nb total de pas de temps par époque | 1000 | 600 |
| Cible Kullback-Leibler | 0.015 | 0.015 |
| Itérations par époque | 80 | 80 |
| Gamma | 0.99 | 0.99 |
| Lambda | 0.97 | 0.97 |
| Politiques optimales sélectionnées | 110 à 179 | 190 à 225 |
| Politiques sous-optimales sélectionnées | 1 à 45 | 151 à 205 |

TABLE 2 Hyperparamètres de la génération de séquences (PPO)

| Environnement | CartPole | Mountain Car |
|--------------------------------|-----------------|---------------------|
| Technique d'optimisation | Adam | Adam |
| Nb d'époques | 250 | 100 |
| Taille de la batch | 8 | 8 |
| Taux d'apprentissage | 0.001 | 0.01 |
| Dimension entrée | 6 | 4 |
| Dimension états cachés | 35 | 30 |
| Dimension sortie | 1 | 1 |
| Weigth decay | 0.01 | 0.01 |
| Nb trajectoires sous-optimales | 500 | 280 |
| Nb trajectoires optimales | 2000 | 2520 |
| Nb de trajectoires totales | 2500 | 2800 |

TABLE 3 Entraînement LSTM

| Environnement | CartPole | Mountain Car |
|-------------------------------------|-----------------|---------------------|
| Technique d'optimisation | Adam | Adam |
| Nb d'époques | 100 | 200 |
| Taux d'apprentissage | 0.01 | 0.02 |
| Dimension entrée | 4 | 2 |
| Nb couches cachées | 1 | 1 |
| Nb neurones couches cachées | 18 | 18 |
| Dimension de sortie | 1 | 1 |
| Nb total de pas de temps par époque | 1000 | 600 |
| Cible Kullback-Leibler | 0.015 | 0.015 |
| Itérations par époque | 80 | 80 |
| Gamma | 0.99 | 0.99 |
| Lambda | 0.97 | 0.97 |

TABLE 4 Entraînement PPO et RUDDER

BIBLIOGRAPHIE

- [1] Jose A. Arjona-Medina, Michael Gillhofer, Michael Widrich, Thomas Unterthiner, Johannes Brandstetter, and Sepp Hochreiter. Rudder : Return decomposition for delayed rewards, 2019.
- [2] R.S. Sutton and A.G. Barto. Reinforcement Learning : An Introduction. IEEE Transactions on Neural Networks, 1998.
- [3] J. Luoma, S. Ruutu, A.W. King, and H. Tikkanen. Time delays, competitive interdependence, and firm performance. Strategic Management Journal, 38(3) :506 – 25, 2017/03/.
- [4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.
- [5] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.
- [6] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.
- [7] Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. Offline reinforcement learning : Tutorial, review, and perspectives on open problems, 2020.
- [8] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. Journal of Machine Learning Research, 13(10) :281–305, 2012.
- [9] Yujing Hu, Weixun Wang, Hangtian Jia, Yixiang Wang, Yingfeng Chen, Jianye Hao, Feng Wu, and Changjie Fan. Learning to utilize shaping rewards : A new approach of reward shaping, 2020.