



Développez une preuve de concept **Never Give Up**

Alan Blanchet
21 février 2024



Tuteur Neovision : Arthur DERATHE
Mentor OpenClassrooms : Chemsse EDDINE NABTI

Table des matières

1	Introduction	2
2	Contexte	2
2.1	Model-based	2
2.2	Model-free	2
2.3	On-policy	3
2.4	Off-policy	3
3	Algorithme	3
3.1	Reward	5
3.2	Episodic novelty module	5
3.3	Life-long novelty module	6
3.4	Augmented reward	7
4	Résultats	7
4.1	Environnements	7
4.2	Papier	7
4.3	Mon implémentation	8
4.4	Suite	9
5	Conclusion	11

1 Introduction

Le monde du Reinforcement Learning (RL) se rapproche de plus en plus de celui du Machine Learning (ML). De plus en plus d'approches utilisent des réseaux classiques tels que le ResNet[1] dans des algorithmes d'état de l'art comme IMPALA[2]. Réseaux de RL qui malgré son âge a pourtant toujours autant d'importance dans le domaine.

En RL, des recherches sont faites pour trouver des algorithmes qui permettent d'atteindre des objectifs de plus en plus complexes. Ces algorithmes se retrouvent toujours face à une problématique cruciale : 'Exploration vs Exploitation'.

Ce rapport a pour but de présenter l'algorithme 'Never Give Up' NGU[3] qui possède également des similarités avec le domain du ML. Il s'appuie sur une technique de récompense extrinsèque et intrinsèque pour maximiser l'exploration dans certains cas et la minimiser dans d'autres comme par exemple pour des observations redondantes ou des observations inutiles.

2 Contexte

Afin de comparer les résultats obtenus par NGU, des environnements standards sont utilisés comme ceux d'OpenAI Gym [4]. Ces environnements représentent des jeux vidéo utilisés pour tester les performances des algorithmes de RL. Ils sont simples à utiliser mais cependant, ces environnements ne sont pas adaptés à des problèmes réels comme par exemple pour la robotique car ils ne prennent pas en compte les différents facteurs des environnements.

NGU essaye de résoudre les problèmes d'exploration lié à une policy (politique en français) π . La policy est une fonction qui prend en entrer une observation de l'environnement s et retourne une action a à effectuer. L'objectif de NGU est de trouver cette policy π qui maximise la somme des récompenses futures tout en explorant un maximum son environnement. La récompense future est définie comme la somme des récompenses à partir de l'état s jusqu'à la fin de l'épisode. C'est la fonction dont l'intérêt est d'être optimisée.

Pour y parvenir, il existe plusieurs familles d'algorithmes qui utilisent des approches différentes.

2.1 Model-based

Les algorithmes de type *model-based* utilisent un modèle de l'environnement pour effectuer des *simulations*. Ceux-ci incluent des algorithmes puissants comme AlphaZero[5]. AlphaGo qui a réussi à battre un champion du monde de Go en 2016 est un exemple d'algorithmes de type *model-based*. Cependant, ces algorithmes sont très gourmands en ressources.

2.2 Model-free

Les algorithmes de type *model-free* n'utilisent pas de modèle de l'environnement. Dans ce type de configuration, l'agent apprend avec une méthode de *trial and error*. L'agent

n'est pas capable d'apprendre sur ses propres *simulations*. Par exemple, l'agent n'est pas capable de se dire "*Et si j'avais pris cette action ? J'aurais mieux fait ?*" et doit donc apprendre directement grâce à l'environnement en suivant une ligne directrice. Pour le voir différemment, dans un environnement avec 4 actions, le modèle *model-free* ne pourra pas concevoir l'issue d'effectuer chaque action pour une même observation. Il doit inévitablement choisir une action et peut uniquement s'appuyer sur l'observation récupérée de l'environnement pour s'améliorer. Plusieurs algorithmes populaires sont issus de ce groupe d'algorithmes tel que PPO[6] ou TD3[7]. C'est le cas de NGU.

2.3 On-policy

Les algorithmes de type *on-policy* peuvent être *model-free* ou *model-based*. Ceux-ci utilisent une seule policy π pour explorer l'environnement et y effectuer des actions. Les actions que l'agent réalise dans l'environnement sont basées sur la policy π_t qui est une fonction qui possède des poids définis par θ_t . La policy π_t est donc définie par $\pi_t(s, \theta_t)$, t représentant le *timestep* d'une expérience. L'agent va donc apprendre à optimiser les poids θ_t de la policy π_t en fonction des récompenses qu'il reçoit. La différence primordiale avec une méthode *off-policy* est que l'agent est limité à un apprentissage sur une *transition* qui est issue de $\pi_t(s, \theta_t)$ uniquement. Ainsi, en effectuant une mise à jour de notre policy, on ne peut plus utiliser la *transition* car nous sommes à l'état s_{t+1} et non plus à l'état s_t . Cette méthode est généralement plus simple à implémenter car elle ne nécessite pas de stocker les *transitions* dans une mémoire tampon. Cependant, elle est moins efficace car l'agent ne peut pas apprendre sur des *transitions* passées.

2.4 Off-policy

Les algorithmes de type *off-policy* peuvent être *model-free* ou *model-based* tout comme le type *on-policy*. La différence fondamentale est qu'on peut tout à fait réutiliser des expériences passées pour entraîner notre agent. Cela permet d'augmenter la vitesse d'apprentissage de l'agent car il peut apprendre sur des *transitions* qu'il a déjà effectuées. Ainsi, on dit qu'on gagne en *sample efficiency*. Cependant, cette méthode est plus complexe à implémenter car il faut stocker les *transitions* dans une mémoire tampon. De plus, il faut faire attention à ne pas utiliser des *transitions* trop anciennes car celles-ci ne sont peut-être plus représentatives de l'environnement actuel. Egalement, il faut faire attention à ne pas utiliser des *transitions* qui se chevauchent entre épisodes car cela pourrait biaiser l'apprentissage de l'agent. Ainsi, plein de concepts intéressants découlent de cette méthode comme les *replay buffers* ou du *importance sampling* [8].

3 Algorithme

L'algorithme NGU[3] est une combinaison et amélioration de plusieurs algorithmes existants. Tout d'abord, un algorithme *Q-learning* est utilisé pour choisir une action en fonction d'un état donné. Cet algorithme comporte une policy π dont la sortie permet de choisir une action. On entraîne donc cette policy π à travers un algorithme de *backpropagation*. Cet algorithme révèle des performances importantes pour l'époque, mais il souffre de plusieurs problèmes un entraînement instable par exemple. On utilise pour corriger ce

problème un deuxième réseau appelé *target network* qui possède exactement la même structure que notre policy. Le but de cette duplication est de stabiliser l'entraînement et de faire converger notre policy plus rapidement. Ainsi, on modifiera les poids de notre réseau *target* moins fréquemment que notre réseau utilisé pour récupérer une action à un instant t . Les poids du réseau principal seront utilisés pour mettre à jour le *target* suivant l'équation :

$$\theta' = \tau * \theta + (1 - \tau) * \theta'$$

τ étant un coefficient généralement entre 0.99 et 0.999 et θ' les poids du *target network*. En effet, si nous utilisions le même réseau pour choisir l'action et pour calculer la valeur de cette action, nous aurions un problème de *moving target* où on chercherait à effectuer une approximation d'un réseau qui est en constante évolution. Combiné avec des changements d'état parfois très fort, cela rendrait l'entraînement très instable.

Le réseau DQN[9] possède une architecture assez traditionnelle de *Convolutional Neural Networks* et utilise un module linéaire en dernière couche afin de proposer des actions en fonction d'une observation. Mais ce modèle ne permet pas de prendre en compte les actions passées. Par exemple dans le jeu de *pong*, le DQN ne peut pas connaître la direction de la balle. Pour cela il y a plusieurs solutions. Le DQN propose une stratégie dite de *stacking* qui consiste à passer aux premières layers des séquences d'observations. Une autre pratique est d'utiliser un réseau récurrent RNN[10] qui permet de prendre en compte les actions passées. On utilise plus couramment un LSTM[11]. Avec cette combinaison de réseaux ML classiques et LSTM on obtient réseau DRQN[12] (Deep Recurrent Q Network).

L'algorithme DRQN[12] permet donc de prêter attention à différentes séquences passées. Les auteurs du DRQN[12] précisent que leur algorithme n'améliore pas les performances du DQN[9]. Cependant, il est plus stable et permet de réduire le temps d'entraînement. C'est donc un bon compromis entre performance et temps d'entraînement. Cependant, il est possible d'aller plus loin en utilisant un réseau récurrent plus complexe. C'est ce que propose l'algorithme R2D2[13] en incluant plusieurs améliorations.

DQN[9] ajoute également une amélioration sur le sampling des données. Il propose d'utiliser une liste de priorités pour pondérer les différentes trajectoires de notre buffer. Les valeurs de pondération correspondent à l'erreur de prédiction du réseau et est modifiée pendant l'entraînement, soit *online*. Ainsi, les trajectoires les plus intéressantes seront plus souvent utilisées pour l'entraînement. Cela permet de réduire le temps d'entraînement et d'améliorer la convergence.

R2D2[13] ajoute la fonctionnalité de lancer plusieurs algorithmes DRQN[12] en parallèle dans des environnements qui leur sont propre. Ainsi R2D2[13] possède son propre réseau central DRQN[12] sur lequel l'entraînement sera effectué. Les réseaux à l'intérieur des processus en parallèle ont pour unique but de récupérer des données avec une policy fixe. Cette policy sera mise à jour selon une fréquence que l'on peut définir sous forme d'hyperparamètre. La mise à jour s'effectue de la policy de l'agent central vers la policy de l'agent dans le processus.

Il y a plus de détails sur le fonctionnement du DQN[9] et de ses dérivées dont j'aimerais bien parler, mais je vais me concentrer sur l'algorithme NGU[3] qui est le sujet de ce

rapport.

Finalement, le réseau NGU[3] vient se greffer à tous ces travaux.

NGU[3] cherche à guider l’agent vers des états qui pourraient l’intéresser en maximisant l’exploration et en attribuant un bonus d’exploration au calcul final de la récompense. Plus le réseau explore l’environnement, moins ce bonus est attribué à l’agent. Cette méthode s’appelle *curiosity-driven exploration* [14].

Le papier propose de séparer le reward en deux catégories. Le reward classique que l’agent obtient en effectuant des actions qui maximisent son reward de l’environnement et un reward d’exploration qui incite l’agent à effectuer des explorations. Soit le *intrinsic reward* et le *extrinsic reward* respectivement.

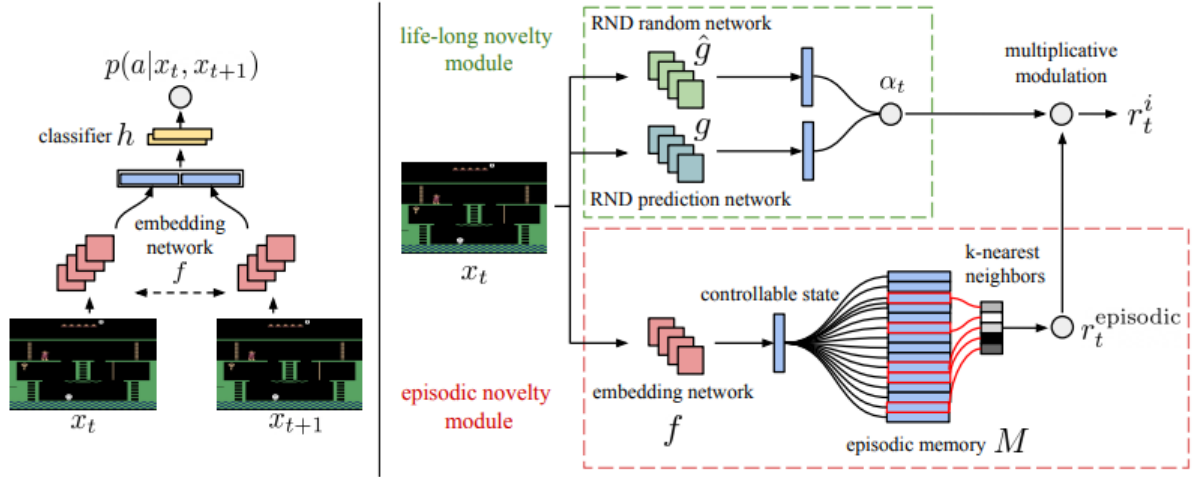


FIGURE 1 – Schéma de l’algorithme NGU[3]

3.1 Reward

Le papier utilise une technique de *reward shaping* pour attribuer un reward d’exploration à l’agent. Ce reward s’appelle le *intrinsic reward* noté r^i et est calculé à partir de deux sous-rewards. Un dans leur *episodic novelty module* noté r^{episodic} et un autre dans le *life-long novelty module* noté α . La formule est la suivante :

$$r_t^i = r_t^{\text{episodic}} \cdot \min\{\max\{\alpha_t, 1\}, L\}$$

Dans le papier, les auteurs ont choisit $L = 5$.

3.2 Episodic novelty module

L’*episodic novelty module* est un reward d’exploration qui est attribué à l’agent lorsqu’il visite un état qu’il n’a pas encore visité dans un épisode. Ce reward est calculé en fonction de la distance entre deux *controllable states* qui correspond à un embedding d’un état. Plus l’agent est loin d’un état qu’il a déjà visité dans un épisode, plus le reward d’exploration est élevé. C’est ce qui va permettre à l’agent de ne pas refaire les mêmes actions dans un épisode. L’embedding est ajouté au *replay buffer* pour ensuite plus facilement être comparé avec les embeddings des états futurs. Cette comparaison s’effectue

en calculant la distance grâce à la fonction K qui représente un algorithme traditionnel de K-Nearest Neighbors. La formule est la suivante :

$$r_t^{episodic} = \frac{\beta}{\sqrt{\sum_{x_i \in N_k} K(f(x_t), f(x_i)) + c}}$$

c est une constante permettant d'éviter une division par 0. K est la fonction de distance. x_i sont les embeddings observés dans un épisode.

Pour calculer ces embeddings, les auteurs utilisent un réseau de neurones f qui prend en entrée un état et qui renvoie un embedding. Pour ce faire, le réseau est entraîné sur un autre réseau d'*inverse dynamics* qui prend en entrée deux embeddings des états s_t et s_{t+1} , puis les concatène pour prédire l'action qui a été effectuée pour passer de s_t à s_{t+1} . La formule est la suivante :

$$p(a|x_t, x_{t+1}) = h(f(x_t), f(x_{t+1}))$$

h étant un réseau qui concatène ses deux entrées et qui renvoie une action. En entraînant ce réseau, on entraîne également le réseau f à trouver des embeddings qui permettent de prédire des actions.

Cette technique permet également de résoudre le problème du *noisy TV* où l'agent maximise son reward d'exploration intrinsèque en n'effectuant aucune action (*NOP*). L'agent obtient son reward uniquement en observant l'environnement changer. Ainsi avec cette méthode, si l'erreur de prédiction de l'action avec deux états s_t et s_{t+1} est élevée, alors l'agent sera encouragé à effectuer des actions autres que *NOP* pour réduire cette erreur.

3.3 Life-long novelty module

L'algorithme dispose également d'un *life-long novelty module* qui permet de calculer un reward d'exploration sur le long terme. Ce module permet de décourager l'agent à visiter des états qu'il a déjà visités dans d'autres épisodes. Ainsi, si dans un épisode on a un bonus $r_t^{episodic}$ élevé et identique pour des états identiques entre épisodes, le *life-long* reward quant à lui sera plus faible pour le deuxième épisode. C'est ce mécanisme qui guide l'agent vers des états qu'il n'a pas encore visité inter-épisodes.

Le life-long novelty module α est calculé de la manière suivante :

$$\alpha_t = 1 + \frac{err(x_t) - \mu_e}{\sigma_e}$$

μ_e et σ_e sont respectivement la moyenne mobile et l'écart-type mobile épisodiques.

$err(x)$ est calculé d'une façon spéciale. Les auteurs utilisent un réseau RND[15]. Ce réseau est composé de deux modules dont les poids sont initialisés aléatoirement. On va considérer un réseau comme étant notre target et l'autre notre réseau de prédiction. Ainsi, lorsqu'un état qui "sort du lot" sera découvert, on aura une erreur de prédiction élevée. C'est cette erreur qui sera utilisée pour calculer le reward d'exploration long-terme.

3.4 Augmented reward

La combinaison de ces rewards nous donne un *augmented reward*. Il se calcul de cette façon :

$$r_t = r_t^e + \beta r_t^i$$

On remarque que le reward d’exploration est multiplié par un facteur β qui permet de régler l’importance de l’exploration dans l’entraînement de l’agent. Plus β est élevé, plus l’agent sera encouragé à explorer son environnement. Les auteurs utilisent plusieurs valeurs de β avec l’algorithme R2D2[13]. Cela leur permet d’avoir plusieurs politiques dont certaines qui cherchent à explorer un maximum l’environnement et d’autres qui cherchent à maximiser le reward de l’environnement. Pour leurs expériences avec plusieurs β , il y a un $\beta = 0$, ce qui fait qu’il y a toujours une policy dans R2D2 qui n’utilise que le reward exploité de l’environnement. En tout, les auteurs utilisent 256 politiques différentes.

Plus l’agent explore l’environnement et devient familier avec celui-ci, plus le bonus d’exploration disparaît et l’apprentissage est uniquement guidé par les rewards extrinsèques.

4 Résultats

4.1 Environnements

Pour obtenir des résultats du réseau, les auteurs ont utilisé les jeux Atari grâce à la librairie OpenAI Gym [4]. Cette librairie expose des jeux vidéo classiques qui sont utilisés pour tester les performances des algorithmes de RL.

4.2 Papier

L’essence même du papier NGU[3] est de montrer qu’il peut mieux performer dans des environnements où l’exploration est primordiale. Ainsi, des tests ont été effectués sur des jeux où le score était auparavant très faible dû à un manque d’exploration de l’agent. Ces jeux comprennent *Montezuma’s Revenge*, *Pitfall!*, *Private Eye*, *Solaris* et *Venture*. Ils ont également comparé les résultats avec des jeux où l’exploration est beaucoup moins pertinente. C’est-à-dire dans des jeux avec des rewards *dense* (qui sont donnés fréquemment) comme *Beam Rider*, *Breakout*, *Enduro*, *Pong* et *Q*bert*.

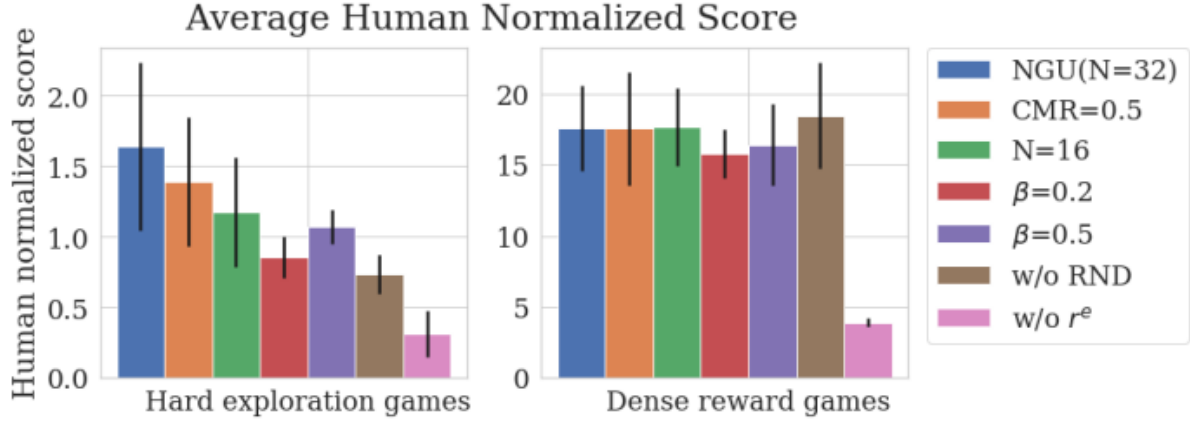


FIGURE 2 – Différentes configurations de NGU sur les jeux Atari avec un reward *dense* et *sparse*

Algorithm	Gravitar	MR	Pitfall!	PrivateEye	Solaris	Venture
Human	3.4k	4.8k	6.5k	69.6k	12.3k	1.2k
Best baseline	15.7k	11.6k	0.0	11k	5.5k	2.0k
RND	3.9k	10.1k	-3	8.7k	3.3k	1.9k
R2D2+RND	15.6k±0.6k	10.4k±1.2k	-0.5±0.3	19.5k±3.5k	4.3k±0.6k	2.7k±0.0k
R2D2(Retrace)	13.3k±0.6k	2.3k±0.4k	-3.5±1.2	32.5k±4.7k	6.0k±1.1k	2.0k±0.0k
NGU(N=1)-RND	12.4k±0.8k	3.0k±0.0k	15.2k±9.4k	40.6k±0.0k	5.7k±1.8k	46.4±37.9
NGU(N=1)	11.0k±0.7k	8.7k±1.2k	9.4k±2.2k	60.6k±16.3k	5.9k±1.6k	876.3±114.5
NGU(N=32)	14.1k±0.5k	10.4k±1.6k	8.4k±4.5k	100.0k±0.4k	4.9k±0.3k	1.7k±0.1k

FIGURE 3 – Résultats comparés aux algorithmes *baseline*, RND et R2D2. La baseline comprend les algorithmes DQN + PixelCNN [16], DQN + CTS et PPO + CoEx [17]

On remarque que NGU ne performe pas mieux partout. Il est plus performant sur Pitfall! et Private Eye et ce avec des paramètres différents. Il est également très proche des résultats à l'état de l'art sur MZ (Montezuma's Revenge).

4.3 Mon implémentation

Mon implémentation est présente dans un répertoire github¹. Elle comprend du code d'un autre répertoire github de 'Coac'² ainsi que mes modifications. Je dispose également d'un autre répertoire github où j'ai mené mes explorations sur les différents modèles et où j'essaie d'implémenter les algorithmes de RL tout en conservant un *clean code*.

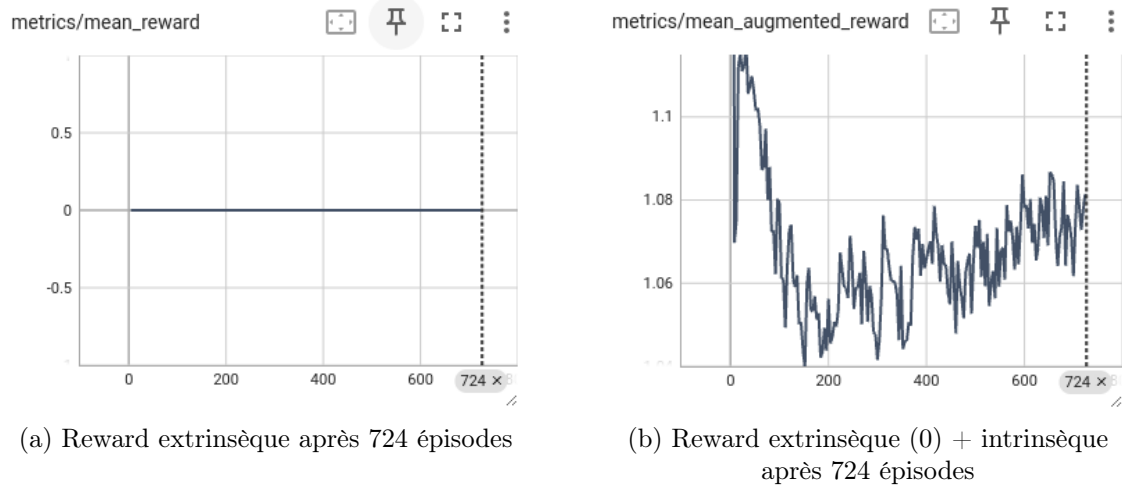
J'ai d'abord commencé par implémenter les versions DQN, DRQN puis R2D2. Mais mon code ne s'adaptait pas facilement à l'algorithme NGU car cela m'aurait fait casser toute ma structure du replay buffer et parallélisation. J'ai donc préféré récupérer l'implémentation de 'Coac' qui était déjà fonctionnelle mais ne possède quant à elle pas l'implémentation de R2D2.

1. <https://github.com/AlanBlanchet/DeveloppezUnePreuveDeConcept>

2. <https://github.com/Coac/>

J’ai apporté des modifications au code afin de pouvoir le tester sur le jeu choisi, *Montezuma’s Revenge*. J’ai également ajouté des logs pour pouvoir suivre l’évolution de l’entraînement à travers Tensorboard. L’implémentation de *Coac* ne prenait pas en compte l’image de l’environnement. J’ai donc dû modifier le code pour intégrer les observations 2D, les encoder en 1D dans l’objectif d’ensuite pouvoir construire l’embedding de NGU.

Ayant quand même eu des résultats sur *NGU*, je suis assez frustré de ne pas avoir eu le temps de proprement coder cet algorithme et de le tester. Je suis donc activement en train de restructurer le code pour pouvoir facilement et proprement lancer des tests poussés sur NGU. J’espère que ces résultats, mes recherches et expérimentations satisfairont mon jury malgré les limites de temps du projet.



Dans les résultats ci-dessus, on peut voir qu’il n’y a aucun reward extrinsèque *mean reward* tandis que le reward total (extrinsèque+intrinsèque) *mean augmented reward* est lui bien présent. Dans notre cas ci-dessus, on a :

$$mean_intrinsic_reward = mean_augmented_reward$$

car *Montezuma’s Revenge* ne nous donne pas de *extrinsic reward* avant beaucoup d’exploration.

L’algorithme a été entraîné pendant 8h et a parcouru un total de 72 500 frames sur un GPU RTX4070 Ti de nvidia. Dans le papier, plus de 35 milliards de frames ont été parcourues. Soit une équivalence de 440 ans d’entraînement pour mon cas.

4.4 Suite

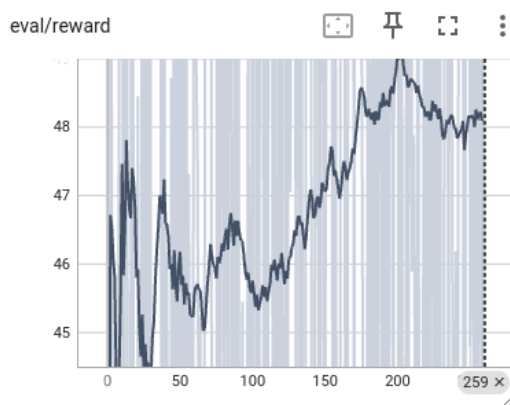
J’aimerais personnellement améliorer les résultats que j’ai pu obtenir malgré que cela soit suffisant pour ce projet encadré par *OpenClassrooms*. J’ai l’ambition de reconstruire un environnement de train/test plus proche de celui proposé dans le papier. Ainsi, je travaille actuellement sur la préparation du terrain dans le but de plus facilement entraîner un modèle RL type NGU (ou autre).

J’ai pour le moment reconstruit toute la partie DQN et ai souhaité optimisé la gestion des

données afin que les entraînements soient bien plus rapides. J’ai construit une architecture de Replay Buffer qui s’appuie sur des *Tensors* pré-alloué au démarrage de l’entraînement. Cela devrait me permettre de plus facilement récupérer les données de mon buffer qui jusqu’à maintenant utilisait une *Queue* de Python. Cette structure comprends les fonctionnalités suivantes :

1. Ne pas stocker les observation suivantes (*next_obs*). Les observations sont parfois en doubles entre deux tensors *obs* et *next_obs*. Cela permet d’optimiser la mémoire pour pouvoir stocker plus de *samples*.
2. Gestion de l’historique. On aimerait pouvoir récupérer une observation et ses *n* observations passées.
3. Circularité. Lorsque le buffer est plein, on écrase les données les plus anciennes. C’est une structure de *Buffer Circulaire*. On doit utilise donc un pointeur qui se déplace et écrase les anciennes données lorsqu’il atteint sa capacité maximale.
4. Sampling en tout genre. On doit pouvoir récupérer les données les plus récentes, de manière aléatoire, un épisode entier...
5. Masking : Les LSTM’s utilisent des masques pour ne pas montrer au réseau de neurones les données qu’il ne peut pas voir. Par exemple, pour la première *frame* du jeu, il n’y a pas d’historique. Le LSTM ne doit donc pas regarder les *n* observations précédentes.

J’ai déjà pu lancer un entraînement avec cette architecture et ai réussi à obtenir des résultats cohérents sur le jeu *Breakout*. J’ai testé cette architecture avec un DQN classique comprenant quelques-unes des fonctionnalités évoquées sur le DQN[9]. Voici un exemple de l’évolution du reward extrinsèque au cours de l’entraînement :



(c) Évolution du reward



(d) Vision de l’algorithme en mode training

J’ai laissé tourner l’algorithme pendant 1,2M de frames, soit 1 jour entier. Voici une vidéo³ de l’agent en pleine évaluation.

3. <https://youtube.com/shorts/d1X-cZxvf2w?feature=share>

5 Conclusion

Pour ce projet de Preuve de Concept, j'ai appris énormément de choses sur les approches de Reinforcement Learning de manière générale. Je n'avais pas encore de notions dans le domaine et était uniquement familier avec les approches type ML. J'ai donc découvert des bibliothèques, des sources d'informations, des papiers ou encore des articles de blog qui m'ont permis d'avancer fortement dans le domaine.

J'ai également pu m'améliorer en développement distribué sous Python, qui est moins simple que dans d'autres langages où le terme de *thread* y est plutôt standard.

Références

- [1] Kaiming HE et al. *Deep Residual Learning for Image Recognition*. <https://arxiv.org/pdf/1512.03385.pdf>. 2020.
- [2] Lasse ESPEHOLT et al. *IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures*. <https://arxiv.org/pdf/1802.01561.pdf>. 2018.
- [3] Adrià Puigdomènech BADIA et al. *NEVER GIVE UP: LEARNING DIRECTED EXPLORATION STRATEGIES*. <https://openreview.net/pdf?id=Sye57xStvB>. 2020.
- [4] Greg BROCKMAN et al. *OpenAI Gym*. <https://arxiv.org/pdf/1606.01540.pdf>. 2016.
- [5] David SILVER et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. <https://openreview.net/pdf?id=Sye57xStvB>. 2017.
- [6] John SCHULMAN et al. *Proximal Policy Optimization Algorithms*. <https://arxiv.org/pdf/1707.06347.pdf>. 2017.
- [7] Scott FUJIMOTO, Herke van HOOF et David MEGER. *Addressing Function Approximation Error in Actor-Critic Methods*. <https://arxiv.org/pdf/1802.09477.pdf>. 2018.
- [8] Tom SCHAUL et al. *Prioritized Experience Replay*. <https://arxiv.org/pdf/1511.05952.pdf>. 2016.
- [9] Volodymyr MNIH et al. *Playing Atari with Deep Reinforcement Learning*. <https://arxiv.org/pdf/1312.5602.pdf>. 2013.
- [10] David E. RUMELHART, Geoffrey E. HINTON et Ronald J. WILLIAMS. *Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network*. <https://apps.dtic.mil/dtic/tr/fulltext/u2/a164453.pdf>. 1985.
- [11] Sepp HOCHREITER et Jürgen SCHMIDHUBER. *LONG SHORT-TERM MEMORY*. <https://www.bioinf.jku.at/publications/older/2604.pdf>. 1997.
- [12] Matthew HAUSKNECHT et Peter STONE. *Deep Recurrent Q-Learning for Partially Observable MDPs*. <https://arxiv.org/pdf/1507.06527.pdf>. 2017.
- [13] Steven KAPTUREWSKI et al. *RECURRENT EXPERIENCE REPLAY IN DISTRIBUTED REINFORCEMENT LEARNING*. <https://openreview.net/pdf?id=r1lyTjAqYX>. 2019.
- [14] Deepak PATHAK et al. *Curiosity-driven Exploration by Self-supervised Prediction*. <https://arxiv.org/pdf/1705.05363.pdf>. 2017.
- [15] Yuri BURDA et al. *EXPLORATION BY RANDOM NETWORK DISTILLATION*. <https://arxiv.org/pdf/1810.12894.pdf>. 2018.
- [16] Georg OSTROVSKI et al. *Count-Based Exploration with Neural Density Models*. <https://arxiv.org/pdf/1703.01310.pdf>. 2017.
- [17] Jongwook CHOI et al. *CONTINGENCY-AWARE EXPLORATION IN REINFORCEMENT LEARNING*. <https://arxiv.org/pdf/1811.01483.pdf>. 2019.