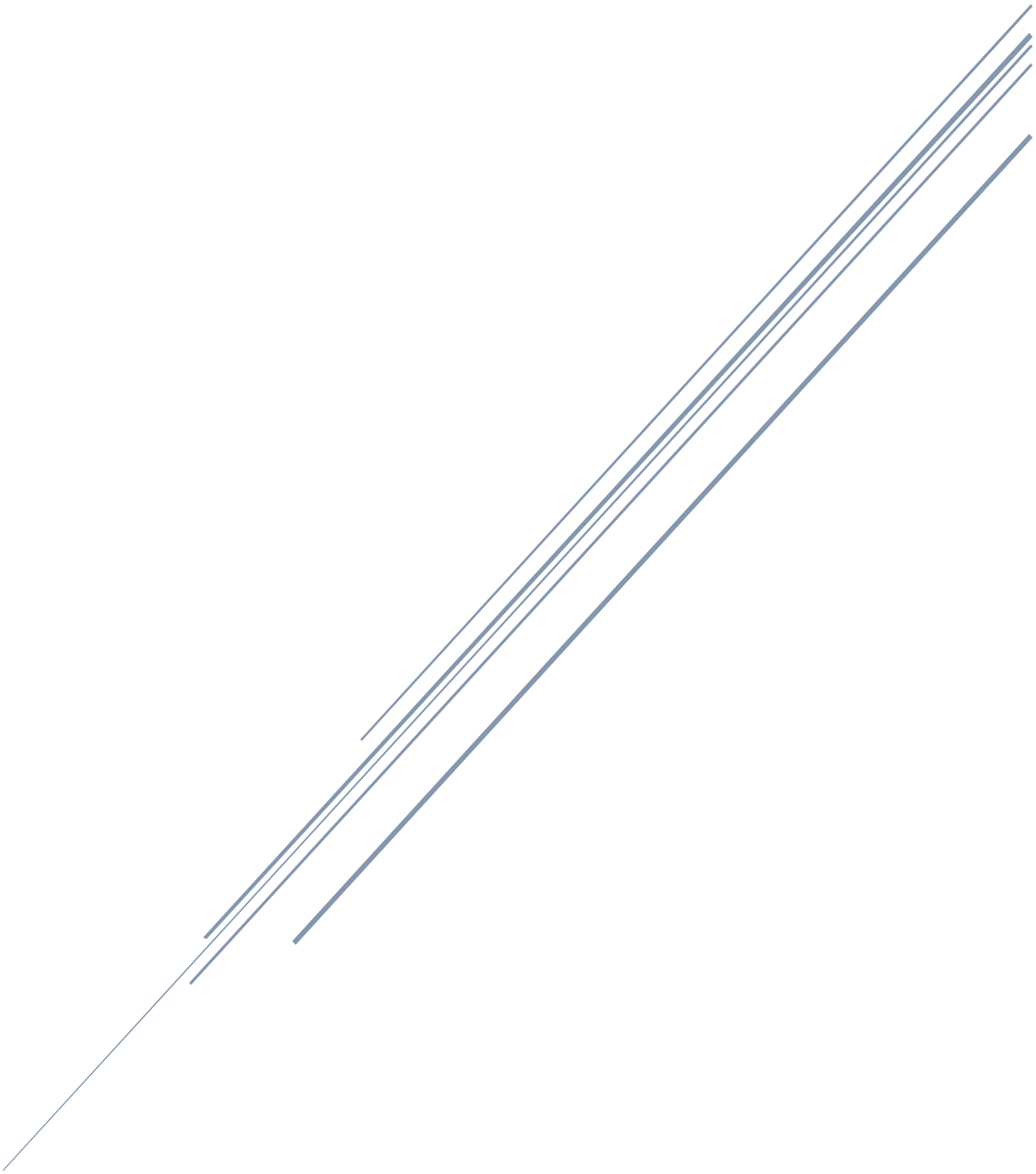
RAPPORT TP BIO-INSPIRÉ

Apprentissage profond par renforcement



Louis Guillotin & Benoît Alcaraz
Master 2 IA – 2020/2021

Table des matières

Introduction

L'objectif de ce TP est de comprendre et de maîtriser un algorithme d'apprentissage profond. Pour ce faire nous avons développé 2 réseaux de neurones différents : le premier étant un réseau de neurone entièrement connecté que nous avons développé lors d'un TP précédent et le seconde étant un réseau de neurone convolutionnel. Le réseau de neurone entièrement connecté permet à un agent d'apprendre à maîtriser l'environnement « CartPole-v1 », dans celui-ci, le but pour l'agent est de garder un bâton en position vertical le plus longtemps possible. Le réseau de neurone convolutionnel permet à un agent de maîtriser l'environnement « VizdoomBasic-v0 », dans celui-ci, le but pour l'agent est de gagner une partie de Doom en abattant un monstre.

Concernant la réplicabilité de notre projet, nous avons utilisé pipenv pour créer un environnement virtuel, toutes les dépendances sont dans le fichier « requirements.txt ». La procédure pour lancer notre projet se trouve dans le README.md du Github.

Partie 2 : DQN sur CartPole

L'expérience Replay

Comme inscrit dans le sujet, le DQN utilise l'expérience replay qui est une mémoire dans laquelle toutes les interactions sont stockées, les interactions étant de la forme (état, action, état_suivant, récompense, épisode_fini ?). Le DQN va se servir de cette mémoire, ou du moins, d'une partie de cette mémoire pour mettre à jour ses poids.

Pour implémenter cette expérience Replay, nous avons développé une classe nommée MemoryReplay. Dans cette classe, il y a 3 fonctions : un constructeur, une fonction pour ajouter une interaction à la mémoire et une fonction pour tirer aléatoirement un sous-ensemble de cette mémoire (servant à l'apprentissage). Le constructeur permet d'initialiser le buffer et une capacité passée en paramètre, le buffer ne pourra pas stocker un nombre d'interaction supérieur à cette capacité. L'ajout d'une interaction est géré dans la fonction PushMemory. Tout d'abord nous avons dût définir ce qu'était une interaction grâce à la fonction namedTuple de la librairie « collections ». Ensuite nous stockons ces interactions dans le buffer, si le buffer est plein, alors nous supprimons le premier élément de celui-ci (l'interaction est « oubliée » mais ancienne donc cet oubli aura probablement très peu d'effet si la taille du buffer est suffisamment grande), et nous rajoutons la dernière interaction faite. Cette fonction est appelée à la fin de chaque itération dans un épisode (après chaque action). La dernière fonction de cette classe permet de tirer un « sample » d'interaction dans le buffer. Elle prend en paramètre une taille de sample et utilise la fonction sample() de la librairie random qui fonctionne très bien pour faire ce qui est voulu.

Réseau de neurone

Le réseau de neurone que nous avons utilisé est le réseau de neurone que nous avions implémenté pour le TP avec Mathieu Lefort. Ce réseau est entièrement connecté. La topologie du réseau est définie dans la classe Network, néanmoins nous n'avons pas développé une topologie unique, en effet, le réseau est assez modulable, il est très facile de changer le nombre de couche ainsi que la taille des couches. Il y a également la possibilité de modifier la fonction d'activation (Sigmoid, TanH, ReLu, etc…). Nous expliquons dans la partie Choix des Hyperparamètres nos choix concernant la topologie du réseau ainsi que la fonction d'activation. La première couche du réseau est toujours de la taille d'un état et la dernière couche est de la taille de l'espace d'action (c'est-à-dire du nombre d'action différentes possibles). Ce réseau permet de retourner la Q-Valeur de chaque action (ce qui

explique la taille de la dernière couche) et permet à l'agent de choisir la prochaine action qu'il va effectuer.

Agent

L'agent est implémenté dans la classe AgentRandom et comprend 4 fonctions différentes. Tout d'abord, cette classe a un constructeur qui prend plusieurs paramètres : l'espace d'état, l'espace d'action, η le taux de d'apprentissage, un booléen pour savoir si l'agent est en mode test ou en mode apprentissage et une variable pour savoir dans quel environnement il se trouve (utile pour le mode test pour charger le bon réseau de neurone). C'est aussi dans le constructeur que l'on va instancier le réseau de neurone de base, le réseau de neurone « target » ainsi que l'optimizer (Adam, qui a semblé être meilleur que SGD durant nos expérimentations). Le réseau de neurone de base est celui sur lequel on va apprendre et évaluer les q-valeurs pour l'état s, et le réseau de neurone « target » est celui sur lequel on évaluer les valeurs pour l'état s+1, cela permet de stabiliser le comportement de l'agent. Nous avons trouvé judicieux de mettre les réseaux de neurone directement dans l'agent car c'est lui qui apprend et qui se sert d'eux. La seconde fonction est une fonction qui permet à l'agent de choisir l'action à effectuer en fonction d'un état passé en paramètre (qui correspond à l'état dans lequel il se trouve). Grâce au premier réseau de neurone, nous allons évaluer les q-valeurs des actions pour cet état (Q(s,a)) et ensuite nous allons suivre une stratégie Greedy, c'est-à dire que nous allons tirer un réel aléatoire entre 0 et 1 et nous allons le comparer à un epsilon passé en paramètre, s'il est supérieur alors on choisit l'action qui maximise la q-valeur, sinon on choisit une action aléatoire dans l'espace d'action. Nous avons choisi d'implémenter cette stratégie car elle est plutôt simple à mettre en place, de plus, nous l'avons déjà étudiée lors d'un TP de SMA avec Laeticia Matignon et cette stratégie s'avère être plutôt efficace. Pour améliorer cette stratégie, nous n'avons pas fixé la valeur de l'epsilon, en effet, sur plusieurs forums, il est écrit qu'il est intéressant de faire décroître la valeur de cette variable au fil des itérations dans le but d'intensifier de plus en plus et de diversifier de moins en moins. De cette manière, on permet à l'agent d'agir aléatoirement au début de l'expérience, ce qui lui permet d'explorer beaucoup d'état (et aussi de remplir le buffer), puis au fur et à mesure, on vient intensifier pour qu'il explore moins et qu'il maximise sa récompense. Cependant la valeur de l'epsilon ne devient jamais nulle, pour garder une partie d'exploration. La 3ème fonction est la fonction d'apprentissage. Cette fonction prend en paramètre un sample d'interaction (sample de la mémoire Replay), un paramètre γ qui correspond au facteur d'atténuation dans la formule de Bellman (utilisée pour la rétropropagation d'erreur) ainsi qu'un paramètre pour savoir à quelle fréquence nous mettons à jour le réseau « target ». A partir de ce sample, on va extraire un tenseur pour les états, un pour les actions, un pour les états suivants, un pour les récompenses et un pour les indicateurs de fin d'épisodes. Ensuite pour chaque état du sample, on va venir calculer sa q-valeur grâce au premier réseau de neurone puis on va calculer la valeur ciblée grâce au réseau de neurone « target » ($max_{a'}Q'(s', a')$). Ensuite on va appliquer l'équation de Bellman pour calculer la valeur attendue qui est égale à $r(s, a) + \gamma \max_{a'}Q'(s', a')$ si l'épisode n'est pas fini et qui est égale à r(s,a) si l'épisode est fini. La prochaine étape consiste à calculer l'erreur et de la rétro-propager dans le réseau principal. Pour cela nous avons utilisé la fonction MSELoss de Pytorch qui permet de calculer l'erreur de prédiction, puis on la rétro-propage. La dernière étape permet d'optimiser les paramètres du premier réseau de neurone grâce à l'optimizer. La 4ème fonction permet de mettre à jour le réseau « target » en recopiant entièrement le premier réseau de neurone dedans. Nous avons choisi cette méthode car elle facile à comprendre et qu'elle est plus simple à implémenter. Elle est appelée de façon régulière dans la fonction d'apprentissage, en fonction du paramètre de fréquence de mise à jour.

Boucle d'apprentissage

La boucle d'apprentissage se trouve dans le programme principale (main). Dans ce programme, il y a le paramétrage de tous les hyperparamètres ainsi qu'une boucle qui peut-être, soit une boucle d'apprentissage, soit une boucle de test des comportements appris (cf. README pour passer dans un mode ou dans l'autre). Concernant la boucle d'apprentissage, le déroulement est assez simple. Dans un premier temps, pour chaque épisode, on commence par réinitialiser l'environnement puis regarder l'état actuel. Puis, on rentre dans une boucle dans laquelle, à chaque itération, l'agent choisi l'action à réaliser en fonction de l'état, puis on récupère l'état prochain, la récompense et si l'état est fini ou pas. Une fois toutes ces informations récupérées, on ajoute l'interaction au buffer. Ensuite, si le buffer est assez grand (c'est-à-dire, d'une taille supérieure à la taille voulue pour le sample), on applique l'apprentissage, puis on diminue la valeur d'epsilon et on change la valeur de l'état actuel pour celle de l'état suivant. Si l'épisode est fini, on passe à l'épisode suivant.

Choix des Hyperparamètres

Pour toutes les expérimentations, les hyperparamètres par défaut (hormis le paramètre en cause) sont les suivants : η : 0.001, batch_size : 32, γ : 0.999, epsilon_start : 1, epsilon_min : 0.05, epsilon_decay : 0.998, update_freq : 500, réseau de neurone avec 2 couches cachées de taille 64 et fonction d'activation Sigmoid, capacité du buffer Replay : 100000

Pour choisir les hyperparamètres pour avoir le meilleur apprentissage possible, nous nous sommes appuyés sur nos expérimentations, nos intuitions (par rapport au formule) et sur certains tutoriels. Pour choisir η, on a suivi une intuition qui est la suivante : avec un η trop petit, l'apprentissage prend trop de temps et avec un η trop grand, l'apprentissage ne sera pas assez bon et les comportements appris seront trop instables. Il faut donc trouver le bon compromis. Cette intuition a pu se confirmer lors de nos expérimentations (cf. Figure 1).

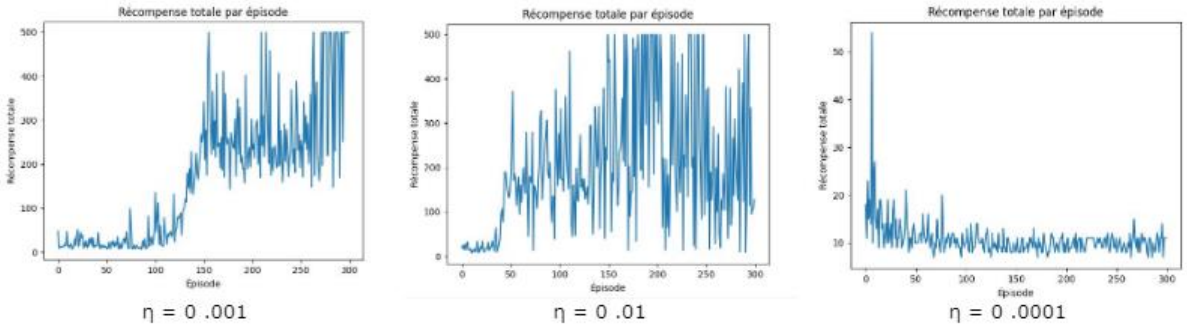


Figure 1 : Influence du taux d'apprentissage

Pour choisir la taille du sample (batch_size), nous avons expérimenté plusieurs tailles en ayant l'intuition qu'une taille trop petite ne permettra pas d'apprendre suffisamment car l'apprentissage se fera sur trop peu d'exemple et par conséquent beaucoup de cas de figure ne seront pas utilisé. En revanche, une taille trop élevée permettrait un bon apprentissage, mais avec une complexité

beaucoup trop importante (Cf. Figure 2). Une taille de 32 nous a parût être le bon compromis, au-delà, la complexité est trop importante et nos machines ne permettent pas un apprentissage rapide.

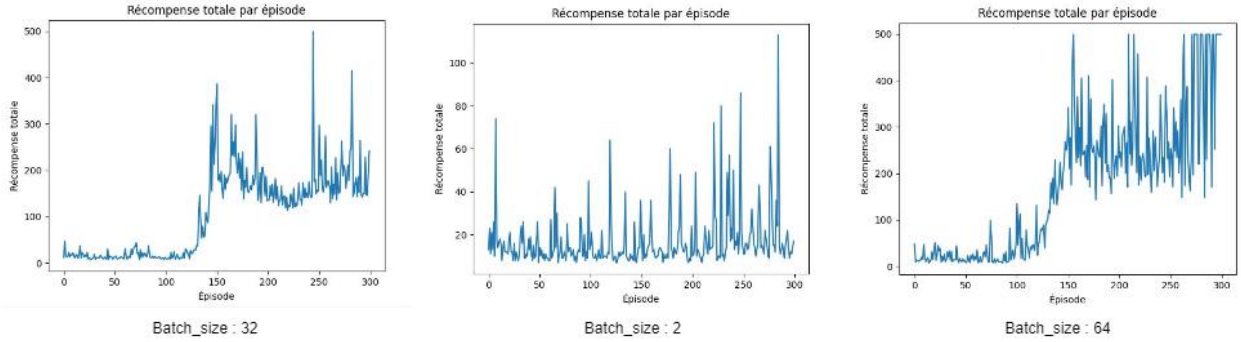


Figure 2 : Influence de la taille du sample

Pour choisir le facteur d'atténuation γ, nous nous sommes inspirés du cours de SMA et de nos expérimentations. En SMA, en général, ce facteur était compris entre 0.8 et 1.0. Dans ce TP, on a pu voir que ces valeurs fonctionnaient aussi. En effet, avec une valeur trop faible, les récompenses dans le futur lointain sont trop négligées et l'agent privilégie des actions qui apporte une bonne récompense dans l'immédiat mais qui n'assure pas toujours une bonne récompense dans l'avenir, ce qui réduit la qualité de l'apprentissage (cf. Figure 3).

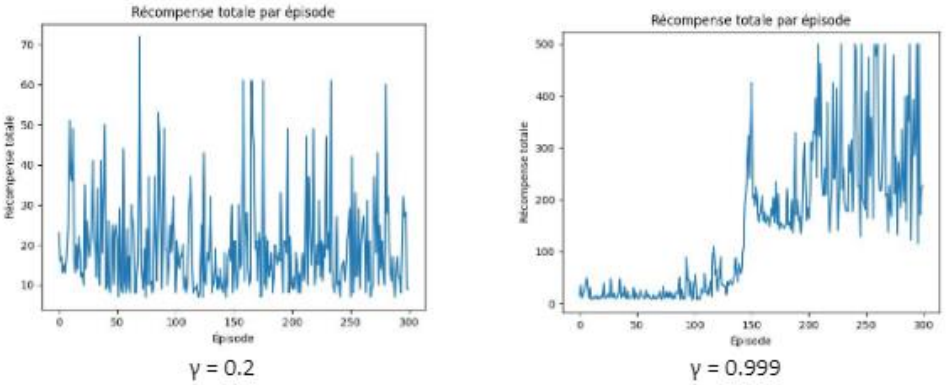


Figure 3 : Influence du facteur d'atténuation

Pour choisir nos hyperparamètres relatifs à epsilon nous savons qu'il est nécessaire de prendre un epsilon de départ (epsilon_start) assez élevé (entre 0.8 et 1.0) afin de beaucoup diversifier au début de l'expérience pour des raisons explicitées précédemment. Ensuite pour l'epsilon de fin (epsilon_end), il faut choisir une valeur pas trop élevée (entre 0.0 et 0.2) pour éviter de trop diversifier et pour intensifier dans la majorité des cas. Concernant le facteur de diminution de l'epsilon (epsilon_decay), il faut qu'il soit proche de 1, pour que l'epsilon ne diminue pas trop rapidement dans le but de garder une phase d'exploration assez longue au début.

Concernant la fréquence de mise à jour du réseau « target », elle ne doit pas être trop basse, car si elle l'est, ce réseau perdra de son utilité. Elle ne doit pas non plus être trop haute car lors de l'apprentissage, la différence entre le premier réseau et le second sera trop importante et l'erreur de prédiction sera « biaisée », par conséquent l'apprentissage ne sera pas bon (cf. Figure 4).

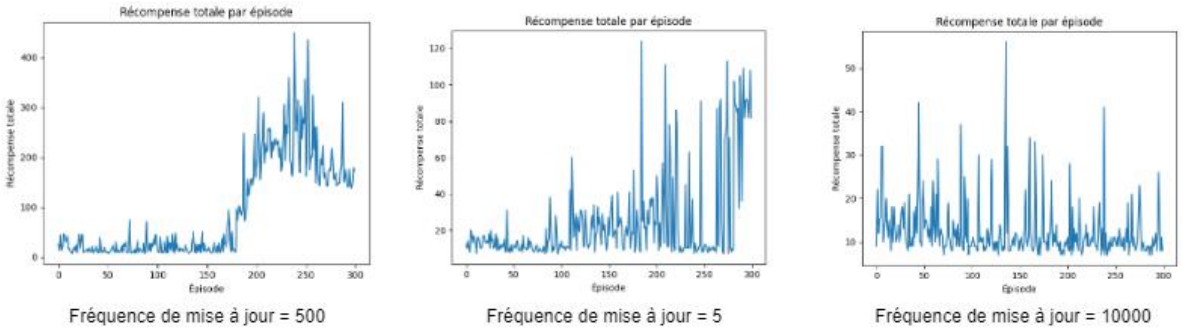


Figure 4 : Influence du facteur d'atténuation

Concernant la topologie du réseau de neurone, nous avons fait plusieurs expérimentations, pour choisir celle qui nous semblait la meilleure (Cf. Figure 5). Avec une topologie à 2 couches cachées, s'il n'y a pas assez de neurones par couche, l'apprentissage n'est pas efficace, en revanche s'il y a trop de neurone l'apprentissage est trop long. Avec des topologies à 3 couches cachées, on peut faire le même constat.

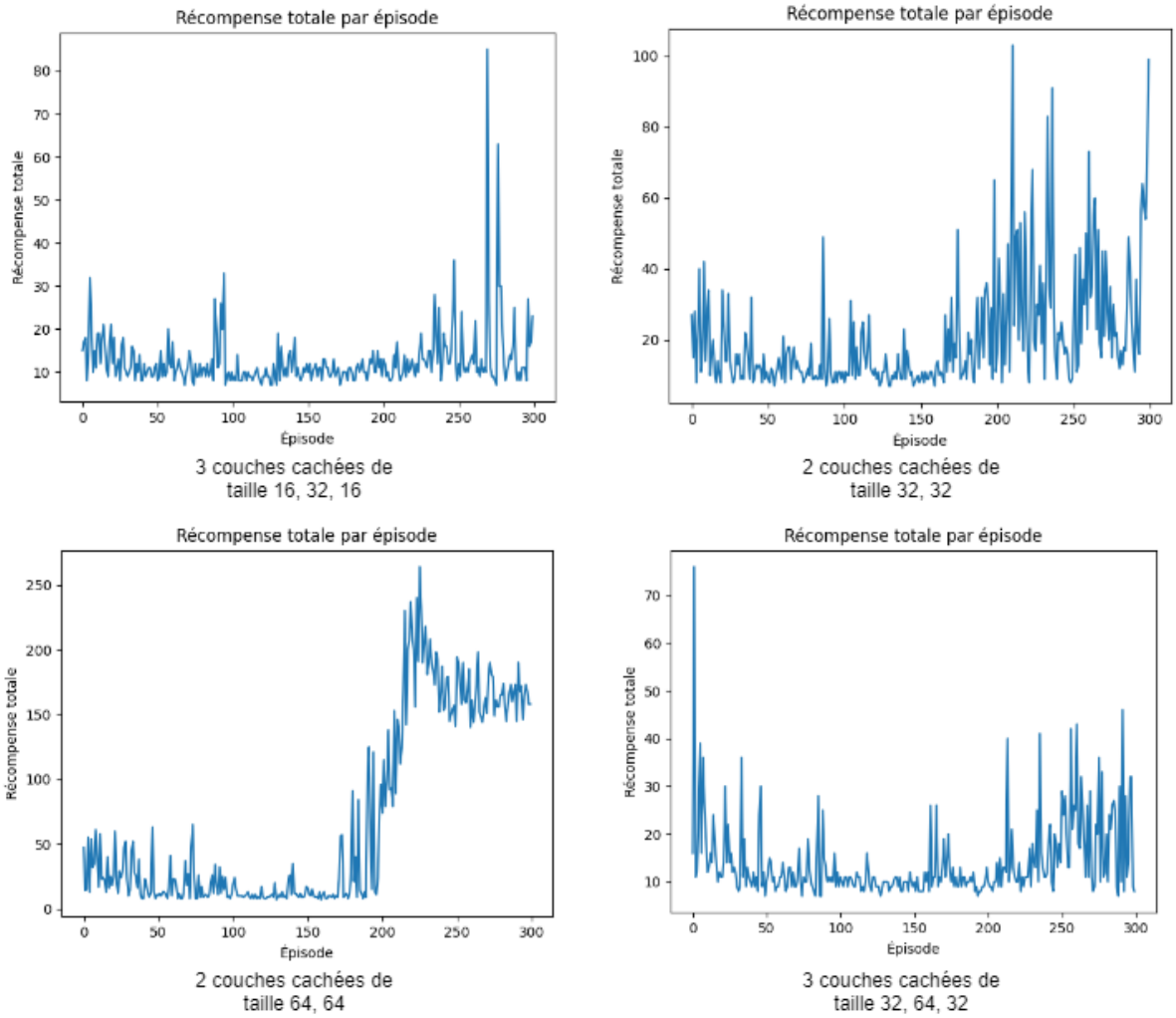


Figure 5 : Influence de la topologie

Pour finir, nous avons choisi la fonction d'activation grâce à nos expérimentations. Néanmoins aucune fonction ne semble réellement meilleure qu'une autre (cf. Figure 6).

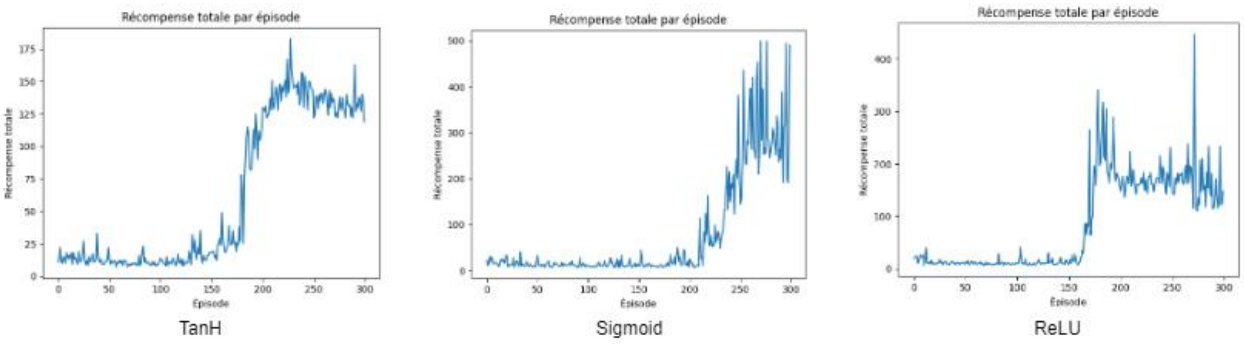


Figure 6 : Influence de la fonction d'activation

À la suite de toutes ces expérimentations, nous avons choisi d'utiliser les hyperparamètres suivants : η : 0.001, batch_size : 32 ou 64, γ : 0.999, epsilon_start : 1, epsilon_min : 0.05, epsilon_decay : 0.998, update_freq : 500, réseau de neurone avec 2 couches cachées de taille 64 et fonction d'activation Sigmoid, capacité du buffer Replay : 100000.

Tests & Résultats

Pour tester notre apprentissage, nous avons utilisé la fonction implémentée à la question 5 de la partie 3 qui permet de charger un réseau de neurone, par conséquent nous avons lancé une boucle d'apprentissage sur 1000 épisodes, puis sauvegardé le réseau de neurone et ensuite lancé une boucle de test de 100 épisodes en utilisant ce réseau de neurone. Lorsque nous sommes en boucle de test, nous affichons tous les épisodes.

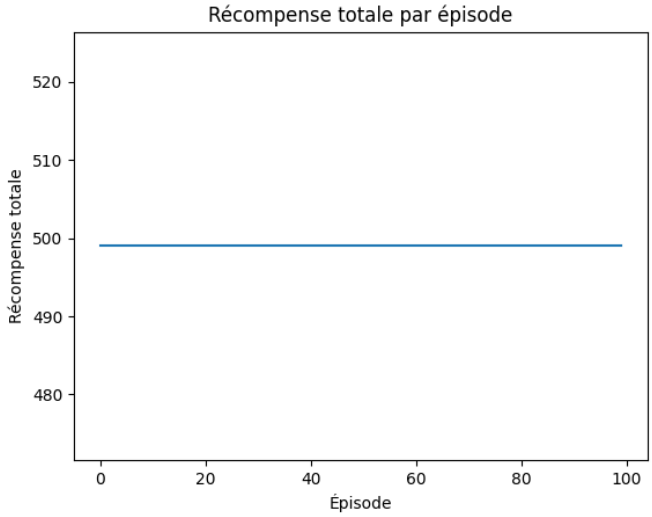


Figure 7 : Résultat de l'agent sur l'environnement CartPole-v1

Sur la figure précédente (cf. Figure 7), on peut voir le résultat de notre DQN sur l'environnement CartPole-v1. On peut remarquer que l'apprentissage a bien fonctionné car l'agent obtient la récompense maximale sur tous les épisodes.

Partie 3 : Environnement plus difficile : Vizdoom

Pour cette partie nous avons dût modifier le vizdoomenv.py de notre environnement virtuel en suivant la pull request proposée sur le GitHub de VizDoom.

Gestion des états

Dans l'environnement Vizdoom, un état correspond à une frame du jeu. De base, chaque frame a une résolution de 320x240 et est en RGB. Comme conseillé dans le sujet, il est intéressant de passer l'image en nuance de gris et de la redimensionner en image de taille 112x64. Ces transformations permettent de réduire le temps d'apprentissage. Nous avons donc utilisé le wrapper GrayScaleObservation pour passer en nuance de gris et le wrapper ResizeObservation pour redimensionner les frames. Comme écrit, dans le sujet, il peut aussi être intéressant de faire en sorte qu'un état soit un ensemble de frames consécutives pour améliorer les performances de l'agent. C'est pourquoi nous avons utilisé le wrapper FrameStack pour qu'un état soit une stack de 4 frames consécutives.

Modification du réseau de neurone

Pour implémenter cette partie 3, nous avons dût modifier notre réseau de neurone et le changer par un réseau de neurone convolutionnel. Pour ce faire, nous nous sommes inspirés d'un tutoriel PyTorch. Nous avons mis en place le réseau de neurone suivant : 3 couches conv2d de taille 4, 16, 64 à laquelle on applique respectivement un filtre de taille 7 (avec stride de 3), un filtre de taille 5 (avec stride de 2) et un filtre de taille 3 (avec un stride de taille 1) (le stride permet de savoir combien de pixel vont être résultant de la convolution) ; entre chaque couche conv2d il y a une couche de normalisation batchNorm2d, et la dernière couche est une couche linéaire prenant une entrée de la taille de l'output de la dernière couche de convolution (qui dépend de la taille de chaque filtre et de la valeur du stride). Cette taille est calculée grâce à la fonction calc_output_size. La fonction d'activation utilisée ici est la fonction relu.

Modification de l'agent

Nous avons créé une nouvelle classe VizdoomAgent basée sur l'ancienne. La seule différence ici se trouve dans les réseaux de neurones qui ne sont plus des réseaux entièrement connectés mais des réseaux convolutionnels. Il y a aussi quelques différences dans la gestion des états qu'il a fallu remanier convenablement pour qu'ils puissent être passés en entrée des réseaux.

Choix des hyperparamètres

Pour choisir nos hyperparamètres nous nous sommes basé sur les choix faits pour l'environnement « CartPole-v1 » et sur un papier de recherche de Hyunsoo Park et Kyng-Joong Kim (https://cilab.gist.ac.kr/wiki/lib/exe/fetch.php?media=public:paper:ieee_cig16_hyunsoo.pdf). Les hyperparamètres sont les suivants : η : 0.005, batch_size : 32 ou 64, γ : 0.999, epsilon_start : 1, epsilon_min : 0.05, epsilon_decay : 0.998, update_freq : 500, réseau de neurone décrit précédemment capacité du buffer Replay : 10000.

Tests & Résultats

Pour tester notre apprentissage, nous avons procédés comme pour le premier environnement, nous avons lancé une boucle d'apprentissage sur 1000 épisodes, puis sauvegardé le réseau de neurone et ensuite lancé une boucle de test de 100 épisodes en utilisant ce réseau de neurone. Dans un premier temps on peut voir que l'apprentissage a fonctionné (cf. Figure 8).

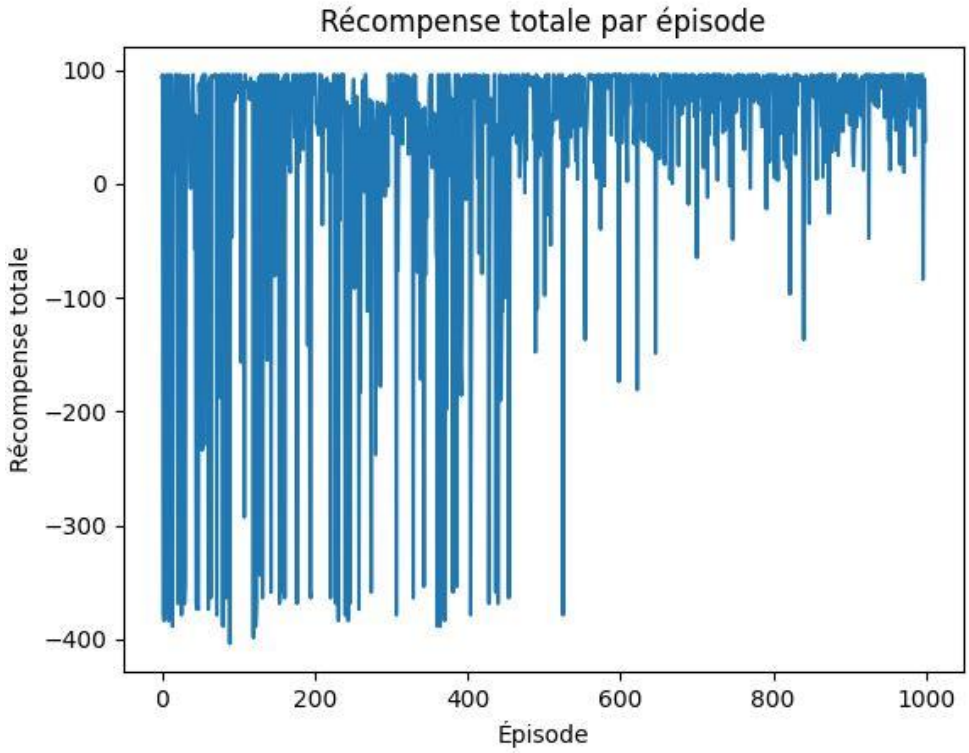


Figure 8 : Courbe d'apprentissage sur l'environnement VizdoomBasic-v0 (1000 épisodes)

En effet, sur cette figure, on peut voir qu'au fil des épisodes, l'agent perd de moins et moins et maximise de plus en plus sa récompense. Néanmoins, une boucle de 1000 épisodes prend beaucoup de temps à s'exécuter et ne permet pas encore d'apprendre des comportements parfaits (comme nous le montre la courbe des résultats sur la phase de test (cf. Figure 9). Nous n'avons pas eu le temps de tester des boucles avec plus d'épisode.

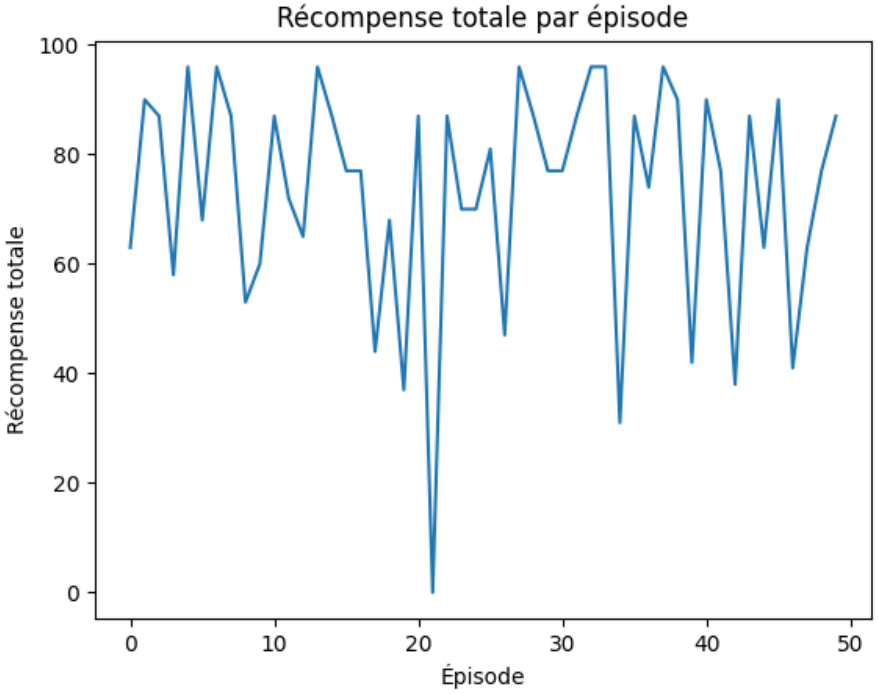


Figure 9 : Courbe des récompenses sur une phase de test sur l'environnement VizdoomBasic-v0

Sur cette figure on voit que l'apprentissage a plutôt bien fonctionné puisque l'agent obtient toujours une récompense positive. Néanmoins, il est légitime de se demander si l'apprentissage aurait pu être encore meilleur avec plus d'épisode ? Et est-ce que les récompenses ici sont maximisées ?

Nous avons ensuite essayé d'apprendre sur d'autre environnement tel que VizdoomCorridor-v0 et VizdoomTakeCover-v0. Par manque de temps nous avons dût réduire le nombre d'épisode à 200 pour l'apprentissage. Malheureusement avec 200 épisodes d'apprentissage, l'agent n'a pas le temps d'apprendre des comportements intéressants et par conséquent, les résultats ne sont pas à la hauteur (cf. Figure 10).

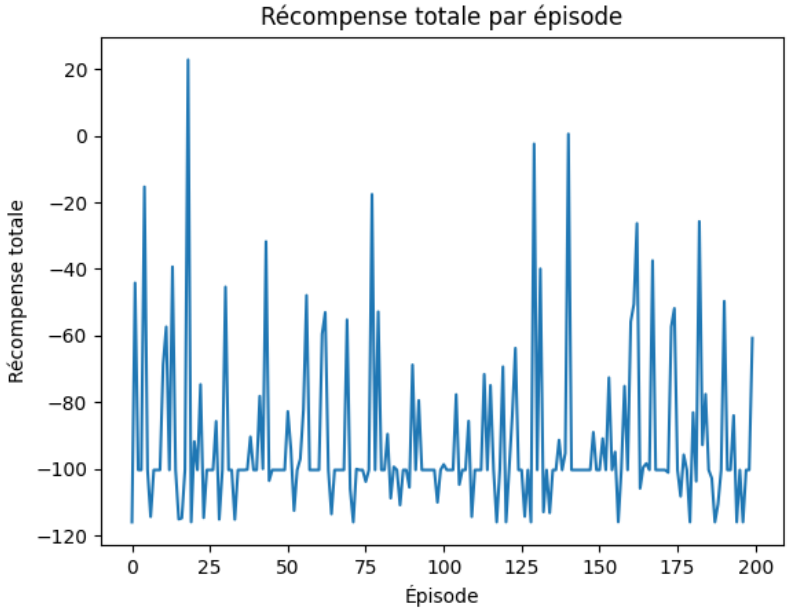


Figure 10 : Courbe d'apprentissage sur l'environnement VizdoomCorridor -v0

Bonus

Nous n'avons pas eu le temps d'implémenter les bonus, néanmoins nous avons pris le temps de regarder le principe ICM en regardant la page GitHub de Pathak. Ayant déjà lu un papier sur la motivation intrinsèque appliquée à la robotique développementale, nous avons trouvé cette approche intéressante même si nous n'avons pas eu le temps de l'implémenter.