Apprentissage par renforcement profond

Réalisé par :

- **Doriane AJMI**
- **Mohamed El Mehdi MAKHLOUF**

Encadré par :

- **Mathieu LEFORT**
- **Devillers ALEXANDRE**

Introduction

Ce rapport détaille la mise en œuvre et l'étude des techniques d'apprentissage profond par renforcement. Nous cherchons à développer et trouver la meilleure politique d'action pour un agent interagissant dans un certain environnement. Dans ce projet, la théorie du Reinforcement Learning est combinée avec celle du Deep Learning pour résoudre le jeu Minecraft.

Les algorithmes seront implémentés en Python à l'aide de la librairie Pytorch. L'environnement sera modélisé à l'aide de la librairie Gym.

Ce TP explore dans un premier temps l'apprentissage par renforcement dans l'environnement **Cartpole-v1** puis dans un environnement plus complexe : MineRL : **Mineline-v0**.

Code et implémentation

Lien vers le dépôt Forge Lyon 1

Le dépôt contient un fichier requirements.txt pour faciliter l'installation et la préparation d'un environnement virtuel python avant de tester le code.

Les fichiers Cartpole.py et MineRL.py exécutent l'environnement associé. Ces deux fichiers commencent par des paramètres contrôlant les phases d'apprentissage et de test du réseau.

Par défaut, ces fichiers sont réglés pour exécuter immédiatement la meilleure politique calculée par les auteurs.

Le premier fichier Cartpole.py contient des parties de codes commentés qui montrent les étapes intermédiaires du TP ainsi une trace des premières expérimentations réalisées afin de se familiariser avec la librairie Gym.

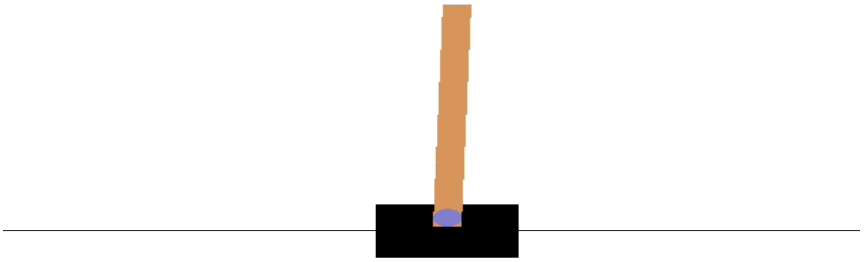
Gym

Gym propose une interface unifiée pour les interactions agent-environnement avec une indépendance dans l'implémentation entre agent et environnement. Cela permet ainsi d'utiliser de façon quasi transparente un agent dans n'importe quel environnement.

CartPole

Dans l'environnement Cartpole, l'agent à deux possibilités d'action : il peut déplacer le chariot (bloc noir) sur la droite ou sur la gauche. Son objectif est de maintenir le bâton (bloc marron) droit. L'agent doit donc apprendre et trouver un équilibre.

Le système de récompense par défaut accorde un point lorsque le bâton est en équilibre et 0 point lorsque celui-ci franchit un seuil d'inclinaison. Sur cette base, nous avons implémenté un algorithme de DQN qui tente de maximiser les récompenses reçues par l'agent.



Premier agent

Le début du fichier CartPole.py contient le code de notre premier agent qui choisit aléatoirement une action parmi les deux pour chaque interaction.

Afin d'évaluer le comportement de notre agent et visualiser son évolution, nous avons décidé d'implémenter la fonction **plot_evolution**, qui permet de tracer un graphe contenant la somme des récompenses pour chaque épisode. Nous utilisons également la sortie du terminal pour avoir des logs de ses valeurs de récompenses et le nombre d'interactions effectué par chaque épisode et également des statistiques globales sur tous les épisodes.

Comme demandé dans le sujet, nous avons mis en place la possibilité de filmer les performances de nos agents afin de pouvoir les évaluer qualitativement sans avoir besoin d'exécuter du code.

Deep Q-Network

Expérience replay

Vous trouverez l'implémentation de notre buffer circulaire dans la définition de la classe **ReplayBuffer** dans les deux fichiers CartePole-v1.py et MineRl.py.

Comme demandé, cette classe permet de stocker des expériences d'interactions de l'agent avec son environnement, mais aussi de tirer aléatoirement un ensemble de ses interactions.

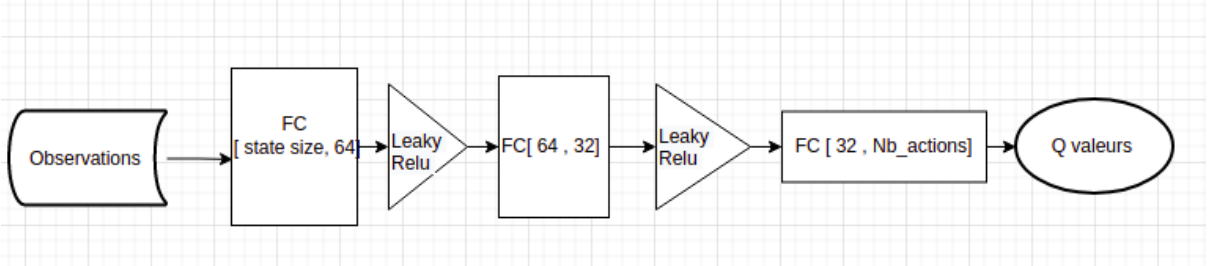
Cette classe contient une mémoire basée sur l'implémentation des deque de python, ce qui permet d'écraser les anciennes expériences si la mémoire est pleine. Cette dernière mémoire stocke les expériences sous formes de tuples nommés namedtuple.

La fonction de classe sample permet de tirer un nombre précis d'expériences de la mémoire sous forme de tenseur Pytorch (il est apparu que la conversion en tenseur lors de la phase d'apprentissage ralentissait très fortement le temps de calcul par rapport à un stockage direct au format tenseur.

Premier réseau de neurones

Le choix de l'action de l'agent utilise une approche Q-learning : un réseau de neurones profond qui prend en entrée l'état courant de l'environnement et donne en sortie un score associé à chaque action.

Pour le problème du Cart Pole, la structure du réseau de neurones choisie est très simple. Pour l'instanciation de ce réseau de neurones profonds il faut juste lui passer la taille de l'état ainsi que le nombre d'actions qu'il peut prendre pour finalement obtenir une architecture qui ressemble à cela :



Architecture du réseau de neurone profond dense utilisé pour la deuxième partie

Exploration

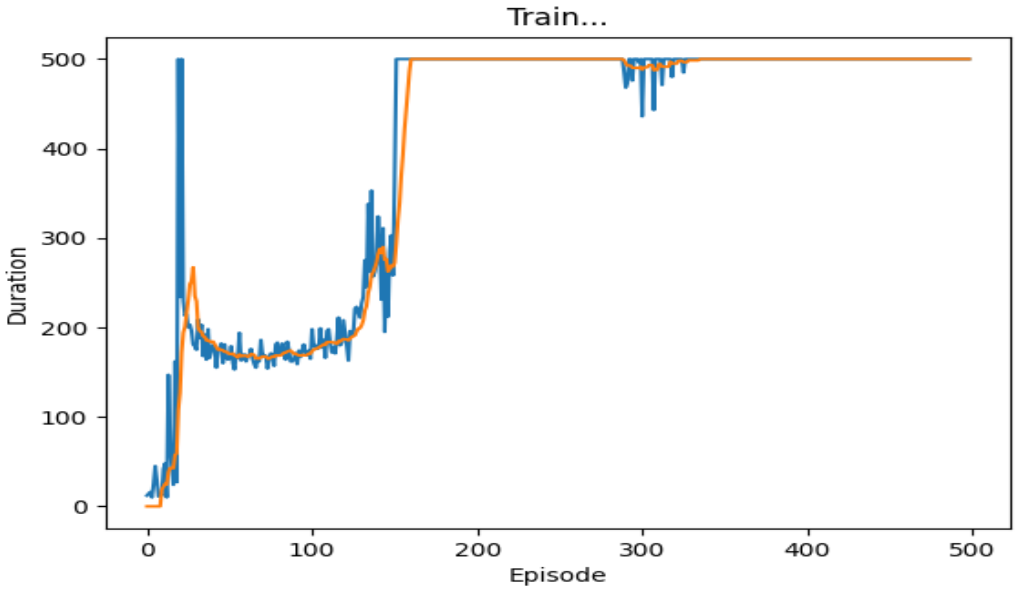
Pendant la phase d'apprentissage, la méthode choose_action() de la classe DQNAgent applique une politique ε-greedy : l'action choisie est celle recommandée par le réseau avec une probabilité $1 - \varepsilon$ ou une action aléatoire avec une probabilité.

Apprentissage

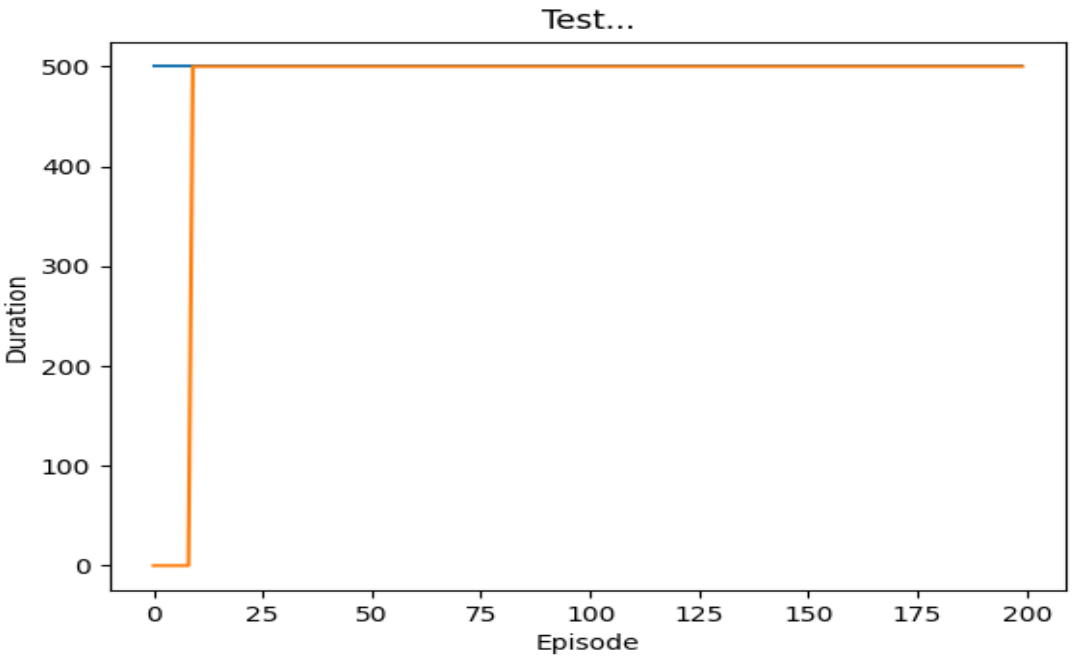
Dans cette partie nous avons suivi la formule donnée dans le sujet et qui correspond finalement à l'algorithme Double Q learning comme décrit dans Deep Reinforcement Learning with Double Q-learning, (Hasselt et al., 2015).
Le code se trouve dans la fonction learn.
Ainsi nous avons mis en place la possibilité de copier les paramètres du policy network dans le target network avec une fréquence à la place d'utiliser la formule donnée pour mettre à jour les paramètres du target network.

Résultats

Les essais sur Cartpole-v1 montrent le bon fonctionnement de l'algorithme. Les deux graphiques ci-dessous détaille notamment l'évolution de la récompense sur 500 épisodes d'entraînement avec 500 actions par épisodes au maximum et sur 200 épisodes de test :



Évolution des récompenses pendant le train de l'agent



Évolution des récompenses durant la phase de test de l'agent

Nous pouvons remarquer sur le graphe que l'agent arrive à atteindre l'objectif déclaré sur le sujet (500).
Nous pouvons également observer sur la figure qui montre l'évolution des récompenses de l'agent durant la phase de test qu'il obtient 500 comme récompense pour chaque épisode.
Nous pouvons remarquer également sa courbe d'apprentissage qui montre clairement l'évolution de son comportement au cours du temps.
Les valeurs des hyperparamètres sont précisées dans l'annexe A et aussi dans le fichier **CartPole-Hyper_Params.txt** sur le dépôt git.
À noter que l'entraînement se fait sur des épisodes avec un nombre d'actions infini si l'agent ne fait jamais tomber le bâton (en réalité l'environnement s'arrête automatiquement après 500 actions).
En effet, stopper l'entraînement lorsque le bâton tombe ne permet pas à l'agent de mémoriser suffisamment de cas négatifs pour apprendre correctement ce qui explique la taille du batch choisi parmi les paramètres (batch size= 500).

Les poids du réseau de neurones ont été enregistrés. Le code Cartpole-v1.py est réglé de sorte à charger ses poids et à lancer un test immédiatement sans entraînement. Pour vérifier les performances de notre implémentation, il suffit donc d'exécuter le fichier Cartpole.py.

Vous trouverez également une vidéo de démonstration des performances de notre agent sur l'environnement Carte Pol v1 sur la forge. **demo-CartPole.mp4**

MineRL

Présentation :

La bibliothèque MineRL permet de faire le lien entre une instance du jeu et le code Python en ajoutant directement au registre de Gym des environnements Minecraft.

Implémentations :

Pré-traitement des observations :

Pour ce faire, nous avons décidé d'utiliser des wrapper customisé qui nous ont permis de:

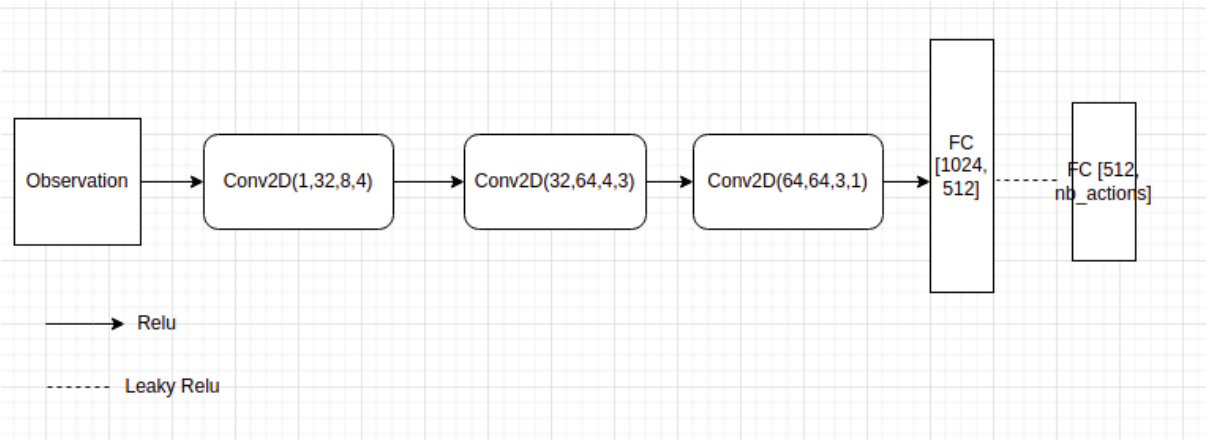
- Change la résolution de nos observations pour avoir des observations de taille 84*84.
- Transformer nos observations en image noir et blanc.
- Normaliser les valeurs de nos observations pour faciliter l'apprentissage.

Pré-traitement des actions :

Sachant que les actions dans l'environnement MineRL se présentent comme un dictionnaire, nous avons décidé d'utiliser deux dictionnaires qui vont nous permettre de les encoder et où chaque action aura un identifiant. Ce qui facilite la mémorisation de ces actions dans le buffer pour choisir des actions grâce aux Q valeurs du réseau de neurone convolutif que l'on va présenter dans la suite.

Réseau de neurones

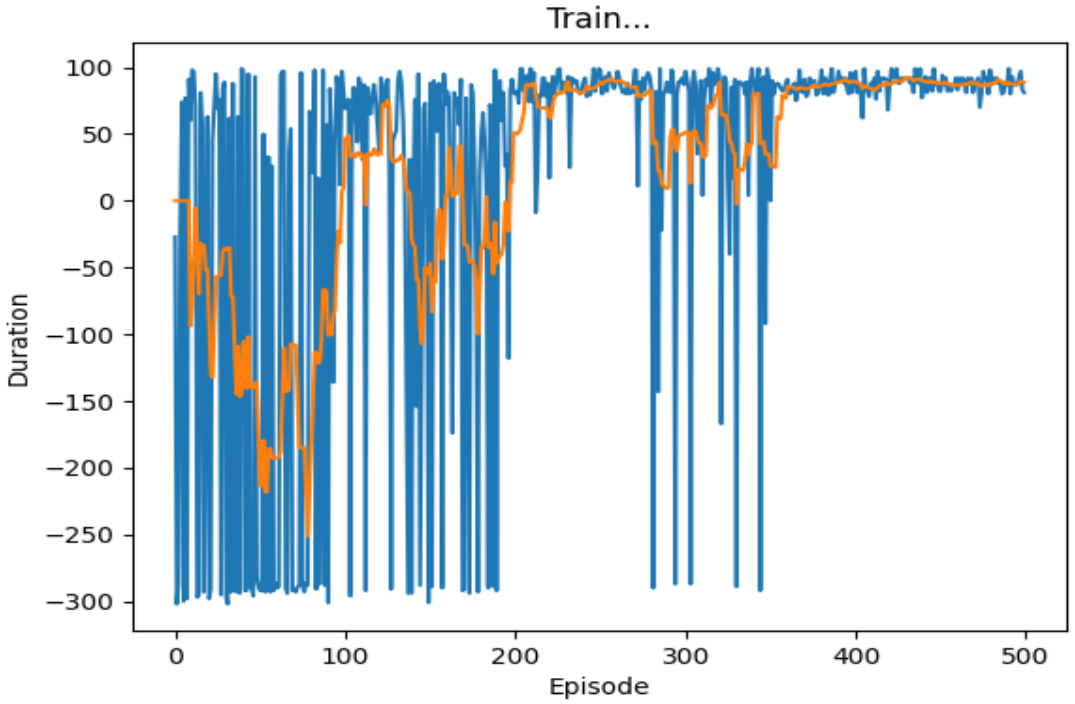
L'utilisation d'images comme observation suggère très fortement l'utilisation de couche convolutive dans la structure du réseau de neurones. En s'inspirant de réseau profond suggéré par l'article de Mnih & al. nous proposons la structure suivante :



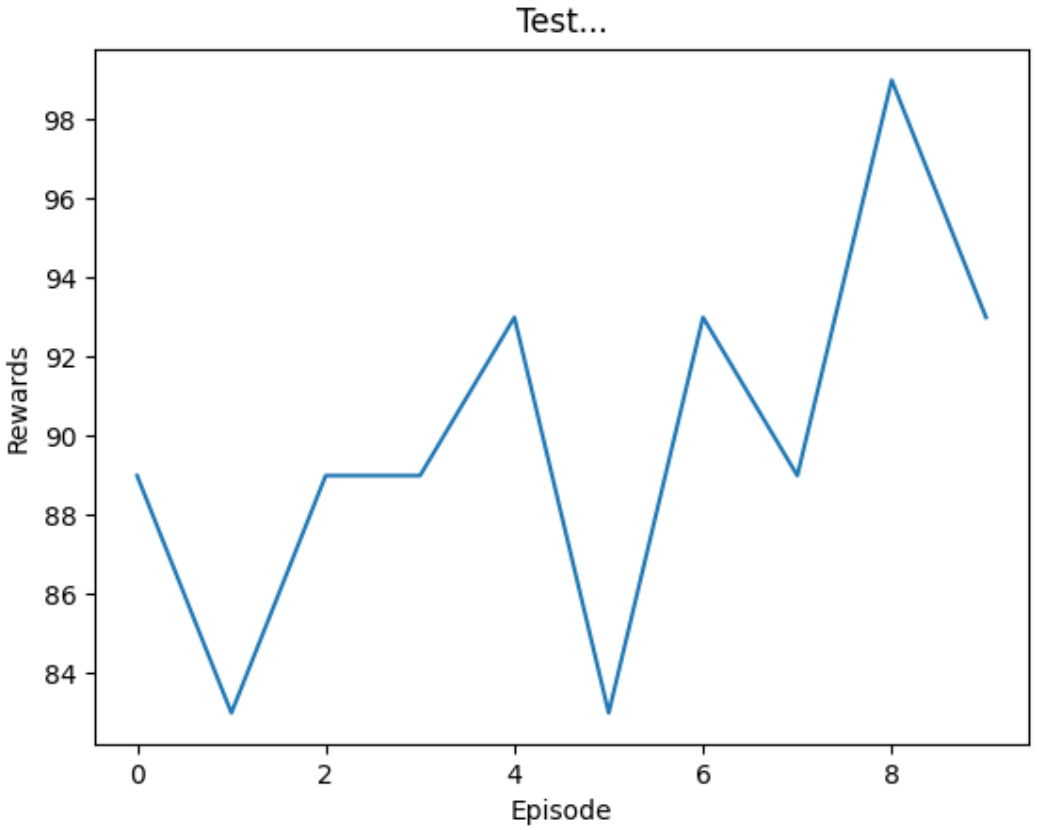
Architecture du réseau de neurone convolutif utilisé pour la deuxième partie

Résultats

Les essais sur **Mineline-v0** montrent le bon fonctionnement de l'algorithme. Les deux graphiques ci-dessous détaillent notamment l'évolution de la récompense sur 500 épisodes d'entraînement avec 300 actions par épisodes au maximum et sur 10 épisodes de test :



Évolution des récompenses pendant la phase d'entraînement de l'agent



Évolution des récompenses pendant la phase de test MineRl

Nous remarquons que sur le graphique d'entraînement, notre agent a pu apprendre sur les 500 épisodes et que sur les 150 derniers épisodes il a un comportement stable avec une moyenne de récompense égale à 90 environ.

Cette dernière remarque peut être confirmée également sur le graphique de test, ou nous avons pu constater que l'agent a eu une moyenne de récompense sur les 10 épisodes de test égale à **90** avec un écart type de **4.5**.

Donc nous pouvons constater que notre agent a pu apprendre grâce à ces réseaux de neurones basés sur des couches convolutives de la même manière que le premier agent l'as fait avec des couches denses seulement.

Comme pour la première expérience, vous trouverez les paramètres du réseau de neurone utilisé ainsi que les hyper paramètres utilisés pour l'entraînement sur la forge.

Nous avons gardé les mêmes hypers paramètres utilisés pour le premier agent sur l'environnement Cart Pole v1 et ils ont donné de bons résultats.

Vous trouverez aussi une vidéo démonstrative du comportement de l'agent sur un seul épisode.

Frame-stacking

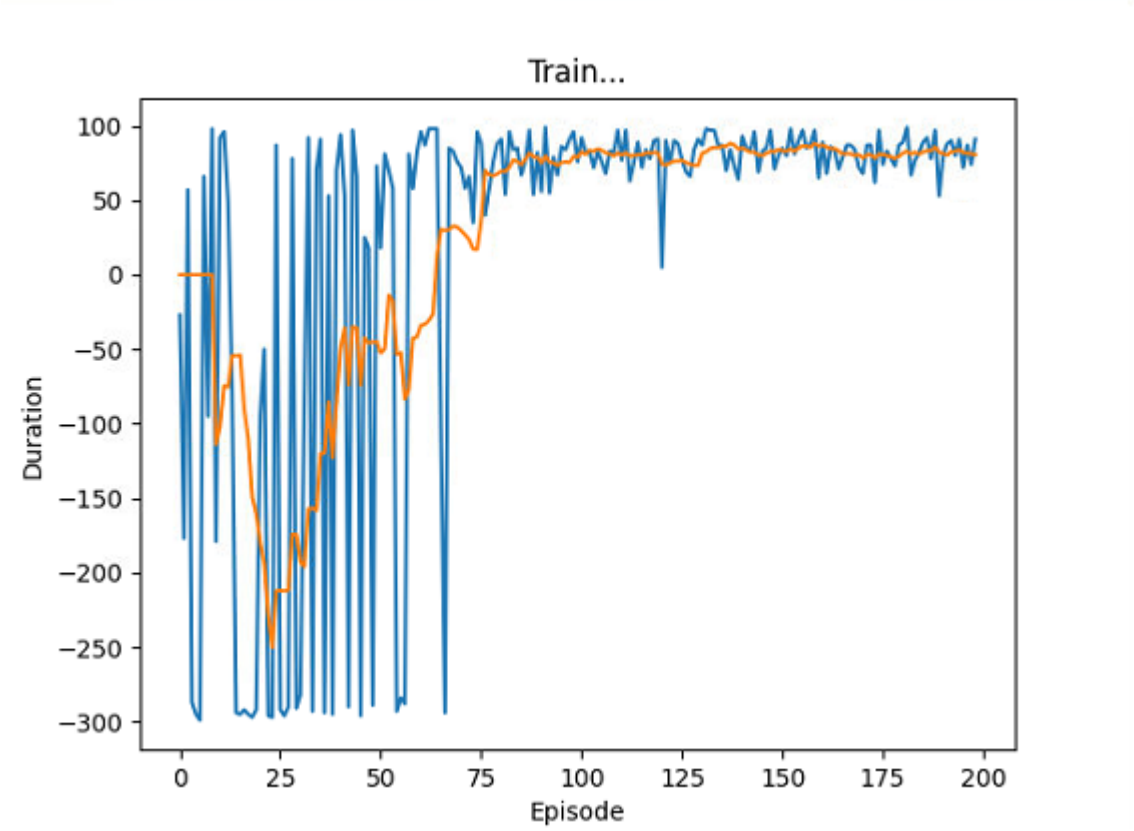
Pour mettre en place le concept du frame stacking nous avons décidé d'utiliser l'implémentation proposée par gym.

Par conséquent, nous avons modifié la première couche de notre réseau de neurone convolutif pour lui permettre de prendre des entrées avec 4 chanel, ce qui correspond au 4 frames que le wrapper Frame Stacking permet de concaténer.

Après cela, nous avons relancé l'entraînement pour voir la différence et ce que le frame stacking peut apporter au comportement de notre agent.

Résultats avec frame stacking

Voici les résultats de l'entrainement de l'agent apprenant sur l'environnement MineRl avec 200 épisodes d'entrainement avec un nombre maximum d'actions par épisodes égale à 300 actions et après on l'a testé sur 10 épisodes pour comparer avec l'agent précédent apprenant sans la technique de stacking.

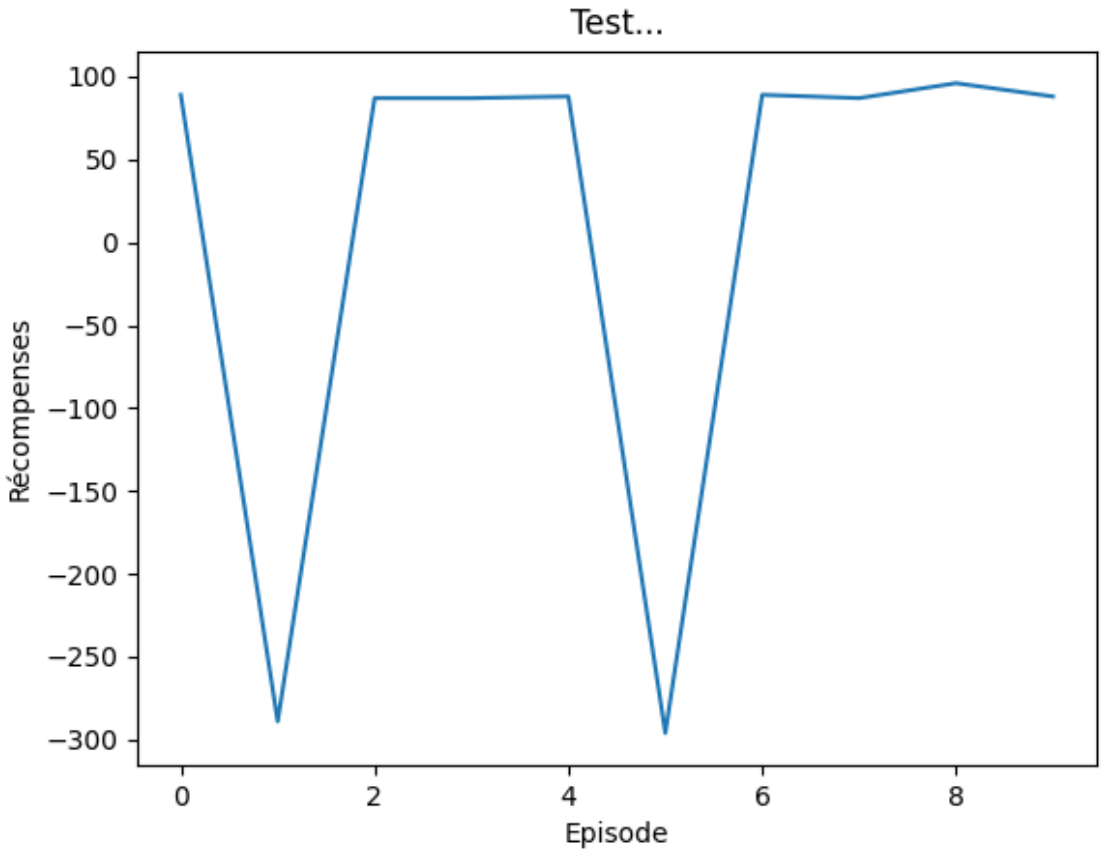


Évolution des récompenses reçue par l'agent durant la phase d'entrainement

Sur la phase d'entrainement on remarque que l'agent avec le frame stacking arrive a atteindre les même perfermonces que l'agent sans frame stacking avec une nombre

d'épisodes d'entrainement inférieur à celui du premier agent, ce qui peut nous donner une idée sur l'affluence de la technique du frame stacking sur l'apprentissage de l'agent.

Observons son comportement sur une phase de test.



Évolution des récompenses au cours des 10 épisodes de test de l'agent apprenant avec la technique de frame stacking

On remarque sur la courbe d'évolution des récompenses au cours des épisodes que :

- L'agent arrive à atteindre plus de 90 de récompenses sur la plupart épisodes.
- Sur d'autres épisodes, il n'arrive pas à avoir de bonnes récompenses.(épisodes 1 et 5 par exemple)
- Cela fait baisser la moyenne de ces récompenses sur les 10 épisodes à 12.6 avec un écart type très élevé 152.(à cause des deux épisodes où il n'a pas eu de bonnes performances)

On peut donc dire que la technique du frame stacking a bien permis à notre agent d'améliorer ses récompenses (il atteint des niveaux de récompenses légèrement plus élevées que l'agent sans frame stacking).

On peut expliquer également les deux épisodes ou il fait de mauvaises performances par le nombre d'épisodes d'apprentissage qui était inférieur à la moitié du premier agent et peut être aussi par les hyper paramètres choisis pour l'entrainement.

Pour conclure cette partie, on peut accepter que notre méthode de comparaison entre les deux agents n'est pas fiable à 100% vu qu'on n'utilise pas les mêmes hyper paramètres d'entrainement pour les deux agents, mais ça nous a permis quand même de démontré l'utilité de la technique du frame stacking, sachant que cette dernière méthode peut avoir plus d'influence sur d'autre environnement ou d'autre problème ou le frame stacking aura plus d'effet.

Méthode pour discrétiser les mouvements de caméra.

Une méthode possible pour discrétiser les mouvements de la caméra serait de définir des actions avec des codes associés.
CEla peut être par exemple 4 actions avec l'encodage et la sémantique suivante :

Action 0	Bouger la caméra de 15 degrés vers la droite
Action 1	Bouger la caméra de 15 degrés vers le haut
Action 2	Bouger la caméra de 15 degrés vers le gauche
Action 3	Bouger la caméra de 15 degrés vers le bas

Ainsi avec un tel encodage, on pourra imaginer 4 neurones de plus sur la dernière couche de sortie qui vont être utilisés pour contrôler la caméra. Donc logiquement il faudra aussi les stocker dans le buffer des expériences pour pouvoir faire un apprentissage sur ces actions lié au mouvement de la caméra et permettre à l'agent d'apprendre comment bouger sa caméra.

Test de l'implémentation sur les autres environnements

Par manque de temps, et la nature du travail demandé qui est chronophage (la phase d'entraînement et choix des hyper paramètres) nous n'avons pas pu tester notre implémentation sur d'autres environnements.

Mais techniquement, l'implémentation est censée marcher et va permettre à l'agent d'apprendre des comportements qui vont lui permettre d'augmenter des récompenses sur différents environnements.

Il est certain que les performances des agents entraînés vont changer entre chaque environnement selon la complexité de l'environnement. Cela reste donc intéressant et peut être même obligatoire de coupler des techniques avec l'algorithme de DQN pour avoir des meilleures performances comme l'utilisation d'un LSTM ou d'un réseau de neurones récurrents.

Conclusion

Finalement, l'approche Deep Q-Learning Network permet effectivement à un agent d'apprendre une politique maximisant l'espérance de récompense. Cependant, cela demande un temps d'entraînement considérable lorsque l'environnement est complexe.

Annexes

Paramètres Cartpole

Buffer size	Batch size	LR	ε	ε min	γ	target update freq	ε decay
20000	512	1e-05	0.99	0.025	0.99	20	0.999

Paramètres MineRl

Buffer size	Batch size	LR	ε	ε min	γ	target update freq	ε decay
20000	512	1e-05	0.99	0.025	0.99	20	0.999

Paramètres MineRl-Frame Stacking

Buffer size	Batch size	LR	ε	ε min	γ	target update freq	ε decay
20000	250	1e-04	0.99	0.025	0.99	40	0.9999

Environnement	Épisodes entraînement	Max actions par épisodes
CartPole	500	-
Mine Rl	500	300
Mine Rl FS	200	300