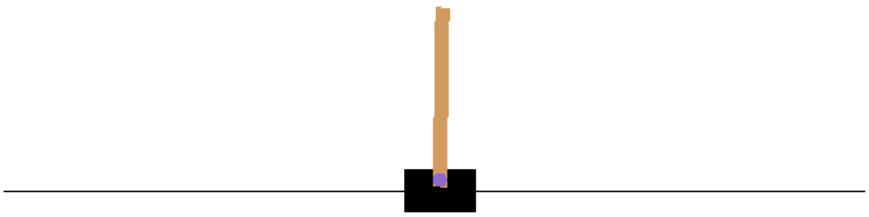
Rapport de projet

-

Apprentissage par renforcement profond

Partie 2 - Bio-inspiré Machine Learning



Khaled ABDRABO p1713323
Jean BRIGNONE p1709655

2022 / 2023
M2 IA - UCBL1

Environnement CartPole

L'environnement CartPole se compose d'un chariot qui se déplace le long d'une piste sans frottement, un poteau posé dessus. Le système est contrôlé en appliquant une force de +1 ou -1 au chariot. Le pendule démarre debout, et le but est de l'empêcher de tomber. L'espace d'état est représenté par quatre valeurs : la position du chariot, la vitesse du chariot, l'angle du poteau et la vitesse de la pointe du poteau. L'espace d'action se compose de deux actions : se déplacer vers la gauche ou vers la droite. Une récompense de +1 est fournie pour chaque pas de temps pendant lequel le poteau reste debout. L'épisode se termine lorsque le pôle est à plus de 15 degrés de la verticale ou que le chariot se déplace à plus de 2,4 unités du centre.

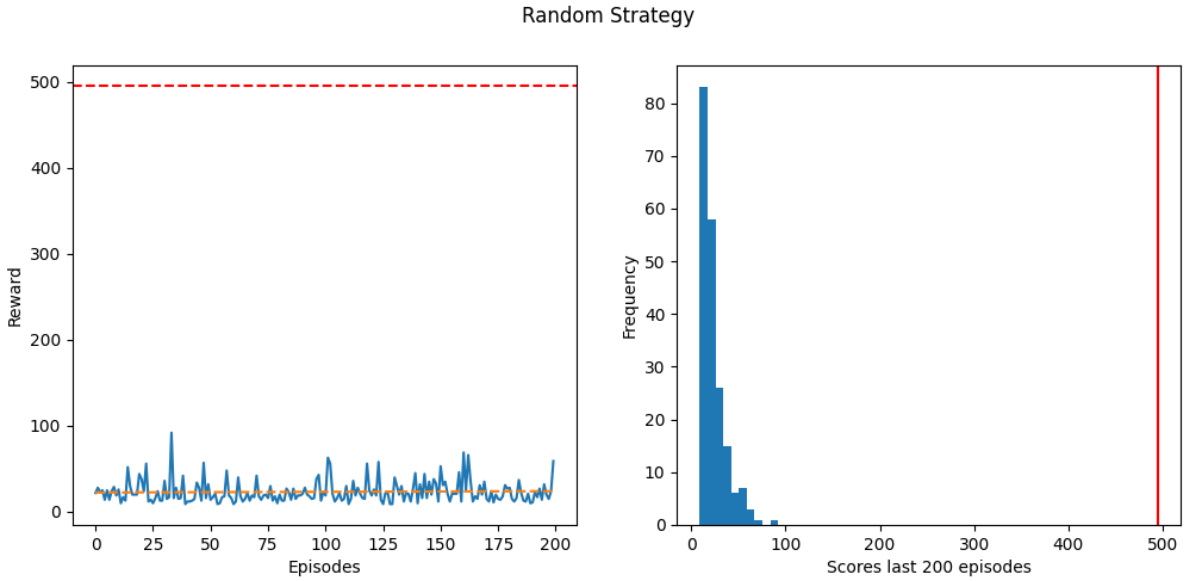
1. Agent CartPole minimal

Fichier : `random_agent.py`

Enregistrements chaque 25 épisodes : `episodes/random_agent/`

Avant de mettre en œuvre toute approche d'apprentissage profond, nous avons écrit un simple script où l'action est échantillonnée de manière aléatoire à partir de l'espace d'action. La mise en œuvre de cet agent nous a servi de base pour d'autres stratégies et a facilité la compréhension du fonctionnement des environnements Gym d'OpenAI.

Résultats obtenus



Les graphes ci-dessus représentent la stratégie aléatoire. Le premier graphe (celui de gauche) trace la récompense totale que l'agent accumule au fil du temps. Le deuxième graphe (celui de droite) est un histogramme des récompenses totales de l'agent pour l'ensemble des épisodes. Les lignes rouges sur les deux graphiques sont notre récompense objective (elle est de 500 sur un épisode).

Comme prévu, il est impossible de résoudre cet environnement (ou n'importe lequel) en effectuant des actions aléatoires. L'agent n'apprend pas de son expérience. Bien qu'il soit parfois chanceux (obtenant une récompense de près de 85), sa moyenne de récompenses est aussi faible : environ 20 pas par épisode seulement.

2. Agent CartPole avec un Réseau Profond Q

Fichier : `dqn_agent.py`

Enregistrements chaque 25 épisodes : `episodes/dqn_agent/`

Nous avons basé notre implémentation du Deep Q-Network sur le document de recherche intitulé « Human-level control through deep enhancement learning », publié en 2015.

L'implémentation de la classe DQNetwork (`src/dqn.py`) consiste en un simple réseau de neurones qui a deux méthodes principales, `predict()` et `update()`. Le réseau prend l'état actuel de l'environnement en entrée et en sortie les Q valeurs pour chacune des actions. La valeur Q maximale est sélectionnée par l'agent pour effectuer l'action suivante. Nous avons implémenté un agent qui peut donner une approximation de en s'appuyant sur notre réseau de neurones, la stratégie d'exploration ϵ-greedy (avec facteur de décroissance) et une fonction de perte.

Pour avoir un apprentissage plus stable, nous avons mis en place un deuxième réseau Target (`src/double_dqn.py`). Les Q valeurs seront extraites de ce nouveau réseau, censé refléter l'état du DQN principal (réseau Policy). Cependant, ce nouveau n'a pas de poids identiques, car il n'est mis à jour qu'après un certain nombre d'épisodes (TARGET_UPDATE).

L'ajout du réseau Target peut ralentir l'apprentissage puisque ce réseau n'est pas continuellement mis à jour. Néanmoins, il devrait avoir une performance plus robuste dans le temps.

Hyperparamètres

Après avoir fait des tests avec les différents hyperparamètres de notre réseau, nous

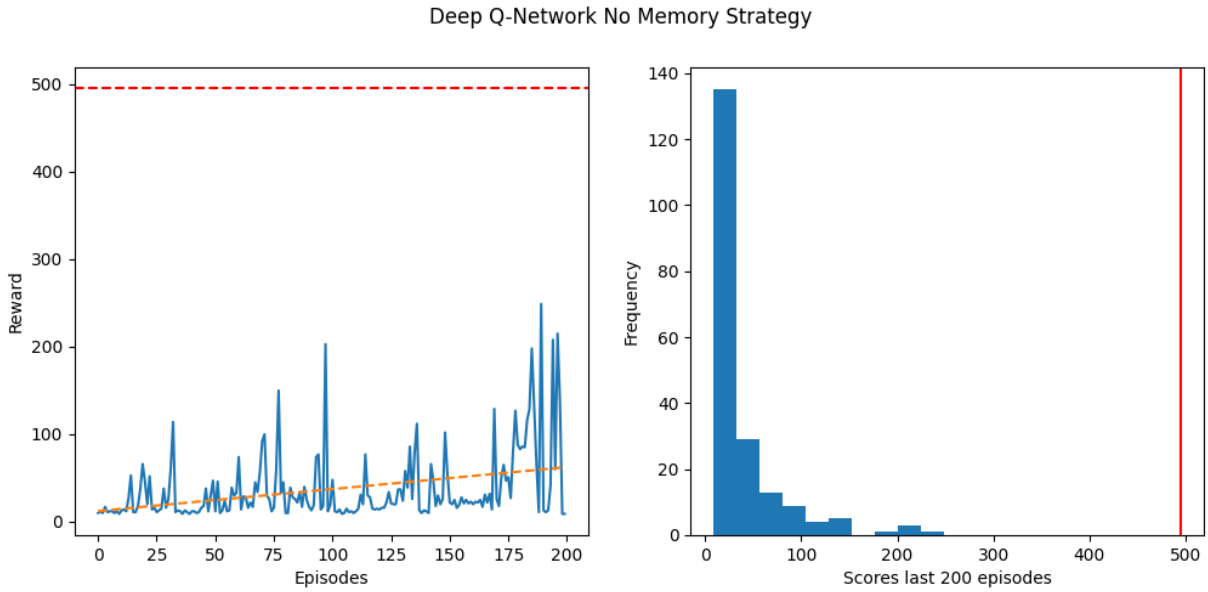
considérons que ces paramètres nous donnent les meilleurs résultats.

- Taux d'apprentissage (éta) = 0.01
- Nombre d'épisodes = 250
- Gamma = 0.999
- Epsilon = $\{start: 0.3, min: 0.01, decay: 0.99\}$
- MemoryReplay = $\{batch\_size: 20, memory\_size: 100000\}$
- Nombre de neurones des couches cachées = 64
- TARGET_UPDATE = 20 (mise à jour des poids du réseau Target par les poids du réseau Policy)

2.1. Apprentissage sans mémoire

Tout d'abord, nous avons implémenté et testé notre agent DQN sans implémenter l'Experience Replay (mémoire des anciennes observations/interactions). L'agent met à jour ses Q-valeurs en fonction de l'observation la plus récente de l'environnement. Comme l'agent n'a pas de mémoire, il apprend d'abord en explorant l'environnement, puis en diminuant progressivement la valeur d'epsilon, ce qui l'obligera à prendre des décisions informées.

Résultats obtenus



Nous pouvons voir que les performances de l'agent se sont considérablement améliorées. L'agent est maintenant capable de marquer jusqu'à 250 unités de récompense en un seul épisode, score qui était impossible à obtenir avec la stratégie aléatoire. La ligne orange montre

la pente des récompenses de l'agent. Dans la première stratégie, on voit que la pente est quasiment parallèle à l'axe des abscisses. Avec le DQNetwork, on voit que c'est positif, et la moyenne de récompenses a aussi augmenté : environ 40 pas par épisode.

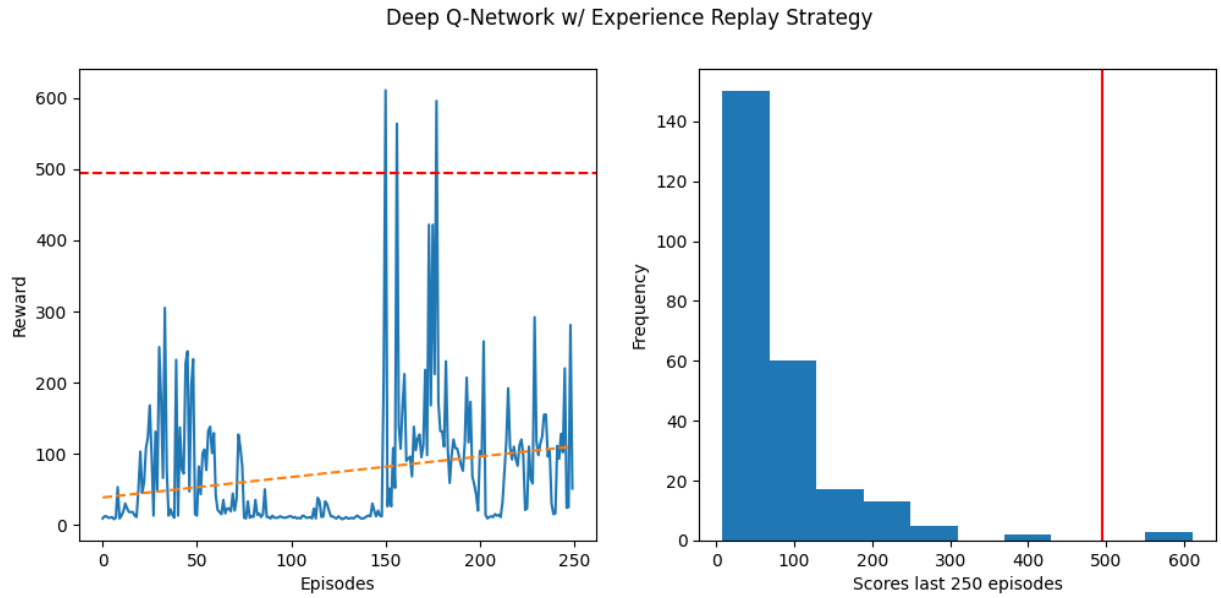
2.2. Apprentissage avec Experience Replay

Comme nous l'avons vu dans la section précédente, le DQNetwork a permis à l'agent de doubler la moyenne des récompenses par épisode, et aussi d'obtenir un score beaucoup plus élevé dans un seul épisode. Cela dit, l'approximation des Q valeurs en utilisant une observation d'environnement à la fois n'est pas très efficace. La mise en œuvre d'une mémoire d'expériences aidera à améliorer la stabilité du réseau et à s'assurer que les interactions précédentes ne soient pas rejetées, mais utilisées dans l'entraînement du réseau.

L'implémentation de la classe ReplayMemory (`src/memory_replay.py`) consiste en un buffer circulaire (`collections.deque`) de taille bornée qui contient les expériences observées récemment. Cette classe dispose de deux méthodes principales : `push()` qui permet de rajouter une nouvelle expérience dans notre buffer et `sample()` qui permet de tirer aléatoirement un lot d'expériences (de taille donnée en paramètre).

Pour que cela fonctionne, nous avons dû créer un tuple nommé `Experience` représentant une seule expérience dans notre environnement. Ce tuple contient les 5 éléments de notre espace d'observation $\{ state, action, next\_state, reward, done \}$.

Résultats obtenus



Comme prévu, le DQNetwork avec la mémoire semble beaucoup plus stable et robuste par rapport au DQN précédent qui ne se souvient que de la dernière interaction. Comme vous pouvez le voir sur les graphiques, l'agent a réussi à obtenir une récompense totale de plus de 500 (en trois épisodes), après environ 150 épisodes d'entraînement. Grâce à la mémoire d'expérience, les récompenses moyennes que l'agent obtient par épisode sont passées à environ 75 pas.

La récompense totale la plus élevée que nous ayons vue par épisode était de 788,0.

```
episode: 188, total reward: 131.0, epsilon: 0.11989742871998742
episode: 189, total reward: 114.0, epsilon: 0.11855014688946754
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! episode: 190, total reward: 788.0
```

Sauvegarde des poids à la fin de l'apprentissage

Fichier démo : `demo.py`

Poids de notre modèle : `pretrained_models/dqn_model.pth`

Nous avons pu enregistrer et charger les poids de notre modèle après l'apprentissage, mais l'agent dans le script de démonstration répète sans cesse le même mouvement (ce qui n'est pas le cas de notre agent lors des épisodes d'entraînement).

Nous sommes tout à fait conscients que notre réseau de neurones n'apprend pas dans le cas de ce script, mais nous pensions que l'agent essaierait au moins différentes actions. C'est pourquoi nous avons essayé de changer le seed de l'environnement et de torch, ce qui n'a rien changé. Nous avons également essayé de charger notre modèle en utilisant différentes méthodes proposées par le tutoriel de PyTorch, mais toutes ont abouti au même comportement.

3. Agent MineRL avec un Réseau Convolutif Profond Q

Nous avons eu des difficultés à tester MineRL sur la VM depuis le début, nous avons alors décidé d'implémenter un réseau de neurones convolutifs et de le tester sur l'environnement CartPole. Une fois ce réseau fonctionnel, nous avons ensuite pu l'adapter pour le faire fonctionner sur les environnements MineRL.

Alternative : Agent CartPole avec un Réseau Convolutif Profond Q

Fichier : `conv_dqn_agent.py`

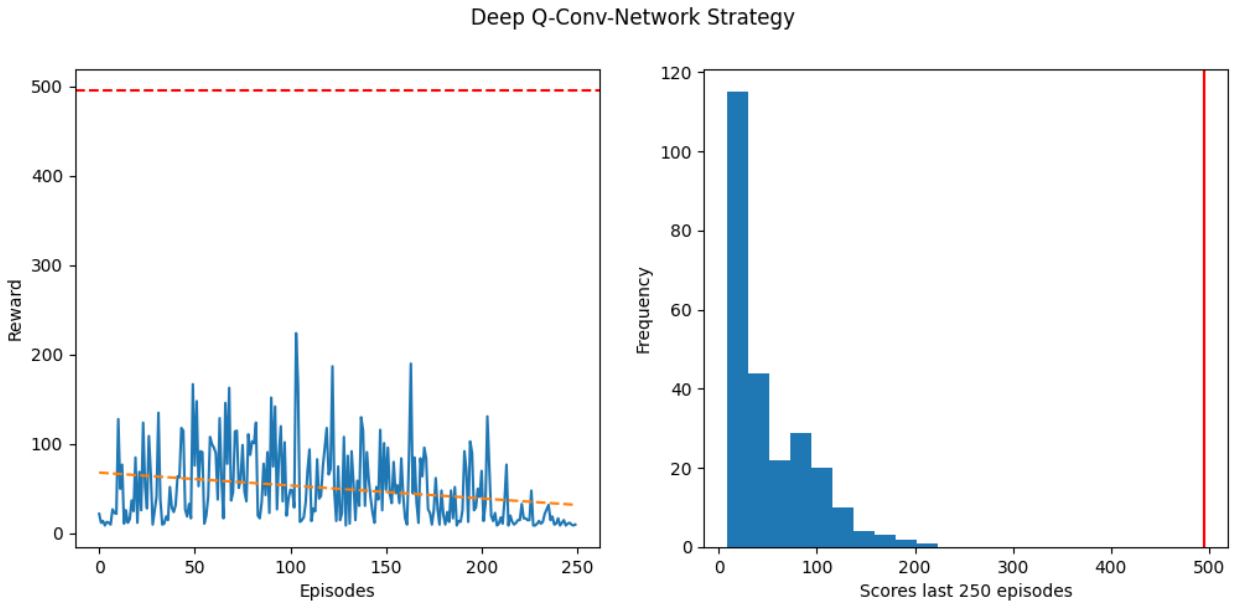
Enregistrements chaque 25 épisodes : `episodes/conv_dqn_agent/`

L'implémentation de la classe DQCN (`src/conv_dqn.py`) consiste en un simple réseau de neurones convolutifs qui a une seule méthode principale, `forward()` qui permet de prédire la ou les prochaines actions (plusieurs en cas de batch).

Ce réseau fonctionne différemment de celui précédemment mis en place, puisqu'il prend en entrée des images et non un tuple d'observation. Nous avons utilisé `torchvision` pour extraire et traiter les images rendues de l'environnement, puis utilisé `PyTorch` pour convertir l'écran en tenseurs que nous pouvons alimenter dans notre réseau convolutif.

Pour avoir un réseau plus stable, nous avons implémenté dans cet agent une mémoire Expérience Replay, des réseaux de politiques et cibles, la stratégie d'exploration ϵ-greedy (avec facteur de décroissance) et une fonction de perte.

Résultats obtenus



Nous pouvons voir que les performances de l'agent, avec un réseau convolutif, sont bonnes, mais ne dépassent pas les performances obtenues précédemment. Avec notre réseau, l'agent est capable de marquer jusqu'à 300 en un seul épisode, ce qui s'approche des résultats que nous avons obtenus en utilisant notre DQNetwork sans mémoire.

En revanche, la pente dans le graphique de droite montre une baisse de récompenses par épisodes avec une moyenne de récompenses d'environ 60 pas par épisode.

État actuel de l'agent MineRL

Fichier : `minerl_agent.py`

(Pas d'enregistrements)

Après avoir réussi l'implémentation du réseau convolutif, nous avons pris l'exemple précédent comme base pour commencer à travailler sur les environnements customisés, plus précisément ; MineLine.

Pour commencer, nous avons dû retraiter les images rendues (le P.O.V de notre agent) de l'environnement. Pour ce faire, nous avons enveloppé notre environnement avec plus de `gym.Wrappers` (par exemple : Skip Frame, Resize Observation, GrayScale Observation et, Frame Stack). Ces wrappers nous ont permis de passer d'images de taille 224x224x3 à 84x84x4 ce qui permet d'avoir un temps d'exécution plus rapide.

Il était maintenant temps d'introduire ces images dans notre réseau convolutif.

Malheureusement, après avoir rencontré de nombreuses difficultés avec les tenseurs de torch, nous avons eu plusieurs problèmes liés aux dimensions. De plus, la machine virtuelle met beaucoup de temps à exécuter le code, nous avons été obligés de nous arrêter ici et n'avons pas pu finir l'implémentation.

Conclusion

Nous avons réussi à implémenter plusieurs types d'agents dans l'environnement CartPole. Nous avons pu nous attaquer au même problème, avec nos différents agents, voir les résultats des différentes approches (Stratégie aléatoire, double réseaux de neurones sans/avec mémoire, réseau de neurones convolutifs) et, le plus important, nous avons pu les comparer.

Pour nous, ce projet était l'occasion de prendre en main de nouveaux outils comme Gym d'OpenAI et mettre en œuvre nos connaissances en Machine Learning à l'aide d'une bibliothèque, assez connue et utilisée, comme PyTorch.