M2-IA

Rapport Projet

Bio-Inspired Machine Learning

Apprentissage par renforcement profond

Groupe:

NESR Mohamad

p1811466

[GIT du projet]

Enseignant:

DEVILLERS Alexandre

Année universitaire **2022/2023**

Table des matières:

I. Introduction:

Ce TP est réalisé dans le cadre de l'UE d'intelligence Bio-Inspirée et a pour objectif de nous familiariser avec l'apprentissage par renforcement profond. Le travail est réalisé à l'aide de PyTorch dans des environnements Gym pour la partie cartopole et des environnements MineRL pour la partie minecraft.

II. Organisation:

Structure du code et des fichiers:

- Cartpole:
    - dqn.py : reseau de neurones
    - experienceReplay.py: buffer
    - qAgentCartPole1.py: agent cartpole avec l'action et l'apprentissage
    - main.py: fichier ou on lance le training + test + l'agent aletoire
    - model.pth: fichier de poids
- Minecraft:
    - minerl/agent.py: agent avec l'action et l'apprentissage
    - minerl/cnn.py: reseau de neurones + buffer
    - minerl/main.py: training + test
    - minerl-model.pth: fichier de poids
- Le dossier perfs contient quelques-unes des meilleures performances obtenues.

III. Qu'est ce que le deep q-learning?

Le deep q-learning est une approche de l'apprentissage par renforcement qui utilise des réseaux de neurones profonds pour approximer la fonction de valeur d'une action dans un environnement donné. Cette fonction de valeur, appelée fonction Q, détermine la valeur d'une action en fonction de l'état actuel de l'environnement et de la récompense future attendue pour cette action. En utilisant un réseau de neurones profond pour approximer cette fonction, le deep q-learning peut généraliser les valeurs d'action à des états que l'agent n'a pas encore rencontrés, ce qui le rend particulièrement adapté aux environnements complexes et non stationnaires.

Le deep q-learning utilise également une stratégie d'exploration-exploitation pour choisir les actions à effectuer dans un environnement donné. L'idée est que l'agent doit explorer différentes actions pour découvrir celles qui ont la plus grande valeur, tout en exploitant les actions qu'il connaît déjà pour maximiser la récompense. Cette stratégie est généralement implémentée en utilisant une fonction d'échauffement, qui permet à l'agent de se concentrer davantage sur l'exploration au début de l'apprentissage et de se concentrer davantage sur l'exploitation à mesure qu'il acquiert de l'expérience.

En résumé, le deep q-learning est une approche de l'apprentissage par renforcement qui utilise des réseaux de neurones profonds pour approximer la fonction de valeur d'une action et une stratégie d'exploration-exploitation pour choisir les actions à effectuer dans un environnement donné. Cette approche est largement utilisée dans les environnements complexes pour résoudre des tâches de contrôle.

IV. CartPole-v1:

Notre premier environnement est l'environnement *'CartPole-v1'* de Gym. Il simule un système physique consistant en un chariot sur lequel est monté un mât vertical. L'objectif de l'agent dans cet environnement est de maintenir le mât en équilibre en déplaçant le chariot de gauche à droite.

Cet environnement offre deux actions possibles à l'agent : déplacer le chariot vers la gauche ou vers la droite. L'agent doit choisir l'action à effectuer en fonction de l'état actuel de l'environnement, qui est représenté par 4 variables : la position et la vitesse du chariot, ainsi que l'angle et la vitesse angulaire du mât. L'agent reçoit une récompense positive à chaque pas de temps où il maintient le mât en équilibre, et une récompense négative si le mât tombe. L'objectif de l'agent est donc de maximiser la récompense totale qu'il reçoit au cours de son interaction avec l'environnement.

Agent aléatoire :

Notre premier agent est un agent aléatoire. On s'attends ici à des résultats plutôt mauvais et aucun apprentissage après chaque épisode. Normalement, la courbe qu'on obtient ne doit pas être croissante. Elle devrait avoir beaucoup de fluctuations

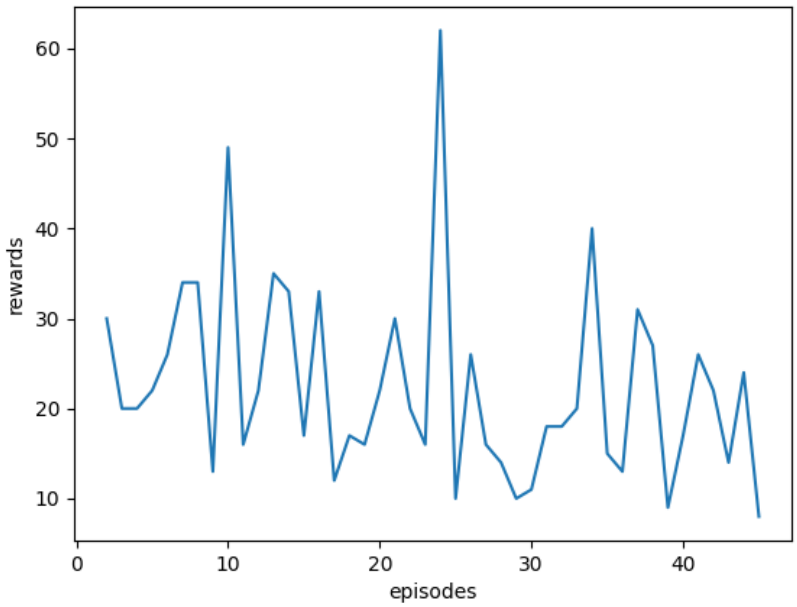

Figure 1: evolution des récompenses en fonction des épisodes pour un agent aléatoire dans l'environnement 'CartPole-v1'

Comme on peut le voir, notre agent n'apprend pas et effectue seulement des mouvements sans vrai objectif derrière. Sur 45 épisodes, la meilleure récompense est de 60 et la pire récompense est inférieure à 10.

Agent Intelligent:

Cet agent a été implémenté en plusieurs étapes qu'on va détailler par la suite.

Experience Replay:

L'expérience replay est une technique utilisée dans les algorithmes d'apprentissage par renforcement pour améliorer l'efficacité de l'entraînement des réseaux de neurones. Elle consiste à enregistrer les expériences de l'agent dans un buffer de replay, puis à utiliser ces expériences enregistrées pour entraîner le réseau de neurones de l'agent.

L'expérience replay permet à l'agent de revoir et d'apprendre à partir de ses expériences passées, ce qui peut améliorer sa performance en lui permettant de corriger les erreurs qu'il a commises dans le passé. Cette technique peut également aider à prévenir le sur-apprentissage en fournissant à l'agent un échantillonnage aléatoire des expériences passées, ce qui l'empêche de se concentrer uniquement sur les expériences les plus récentes.

Caractéristiques de mon buffer replay:

- **Buffer** de taille 100000
- **Tuple** {état, action, prochain état, récompense, fin}
- Mécanisme de **sauvegarde** d'un tuple
- Mécanisme **d'échantillonnage** aléatoire

Reseau de neurones:

Pour le réseau de neurones j'ai créé un DQN simple comportant plusieurs couches. La première couche prend en entrée des données de dimension 4 (caractéristiques de l'environnement) et renvoie des sorties de dimension 32. La deuxième et la troisième couches prennent en entrée les sorties de la couche précédente (dimension 32) et renvoient également des sorties de dimension 32. La dernière couche prend en entrée les sorties de la couche précédente (dimension 32) et renvoie des sorties de dimension 2 (action à réaliser). Entre chaque couche on a une fonction d'activation **ReLU**. La fonction ReLU est souvent utilisée comme fonction d'activation dans les couches cachées des réseaux de neurones, car elle permet d'améliorer la convergence de l'apprentissage et de réduire le temps de calcul.

On a également défini une fonction de perte **MSELoss** pour mesurer l'écart entre les prédictions du modèle et les valeurs réelles. Cette fonction de perte calcule la moyenne des erreurs au carré entre les prédictions du modèle et les valeurs réelles, ce qui permet de mesurer la différence entre ces deux valeurs. La structure du réseau de neurones ressemble donc à: **DQN[4, 32, 32, 2]**

Strategie d'exploration:

On va appliquer une stratégie **e-greedy** avec **decay**. Cette stratégie consiste à choisir aléatoirement une action avec une probabilité égale à la valeur epsilon (e), qui est une hyper paramètre contrôlant l'exploration. Si la valeure aléatoire est inférieure a e, on fait une action aléatoire. Sinon, on choisit notre meilleure action possible. Le decay implique de définir une valeur initiale pour epsilon (e) et

de la réduire progressivement au fil du temps. Cela permet de contrôler **l'exploration** de l'agent en début d'apprentissage, lorsqu'il n'a pas encore beaucoup d'expérience, et de favoriser **l'exploitation** de son expérience acquise en fin d'apprentissage. Cette stratégie permet d'atteindre un bon compromis entre exploration et exploitation, ce qui peut améliorer les performances de l'agent dans un environnement donné.

```
# greedy policy
if random.random() < self.epsilon:
    # random action
    return self.act_space.sample()
else:
    # best action
    return q_values.argmax().item()
```

```
# define epsilon
self.epsilon = max(self.epsilon * EPS_DECAY, EPS_MIN)
```

Figure 2: code de l'implémentation de la stratégie e-greedy avec decay

Target network:

La raison de créer un target network est de stabiliser la fonction de valeur en utilisant des valeurs cibles stables pour calculer la fonction de perte et ajuster les poids du modèle principal. Sans un target network, la fonction de valeur peut être instable et fluctuer fortement au fil du temps, ce qui peut entraîner des oscillations dans les prédictions du modèle et rendre l'apprentissage plus difficile.

J'ai implémenté 2 méthodes de mise à jour du réseau de neurones.

- **Soft update:** consiste à mettre à jour les poids du target network en utilisant une combinaison des poids du modèle principal et des poids actuels du target network. Cela peut se faire en utilisant la formule suivante :

  *target_weights = (1 - tau) * target_weights + tau * model_weights*

- **Hard update:** consiste à mettre à jour immédiatement les poids du target network avec les poids du modèle principal. Cela peut être utile si les poids du modèle principal convergent vers une solution optimale et que l'on souhaite mettre à jour immédiatement les valeurs cibles utilisées pour calculer la fonction de perte.

En résumé, la soft update permet de mettre à jour progressivement les poids du target network en utilisant une combinaison des poids du modèle principal et des poids actuels du target network. La hard update permet de remplacer complètement les poids du target network par les poids du modèle principal. La méthode à utiliser dépend des besoins de l'application et des caractéristiques des données (j'ai codé les deux méthodes mais j'utilise qu'une seule).

Resultats:

hyperparamètres	valeur
eta	0.01
batch_size	128
episodes	300
epsilon	1.0
gamma	0.99
Loss function	MSELoss()
tau	0.01
Epsilon decay	0.995
Epsilon minimum	0.001
optimiseur	Adam
Taille memoire	100000

Figure 3: tableau représentant nos réglages des hyperparamètres

Après avoir réalisé plusieurs tests avec différents hyperparamètres, ceux fournis ci-dessus ont été les plus optimaux dans notre cas avec notre réseau de neurones. De plus, on a obtenu de meilleurs résultats avec le soft update plutôt que le hard update. On a donc procédé en utilisant cette méthode.

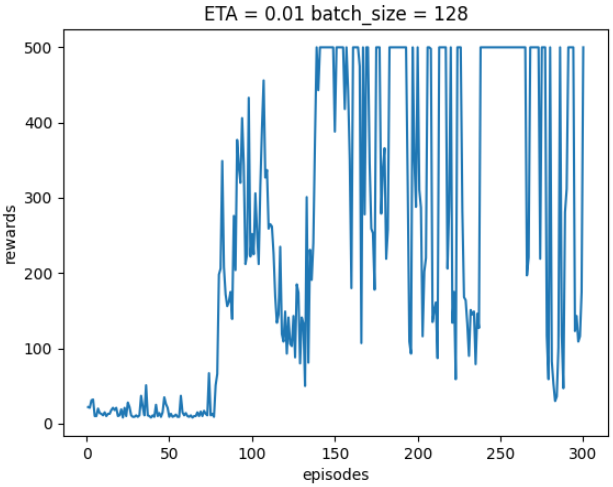

Figure 4: évolution des récompenses en fonction des épisodes pour l'apprentissage

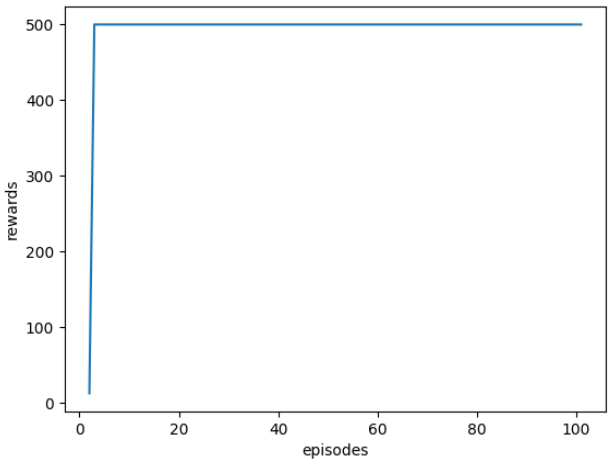

Figure 5: évolution des récompenses en fonction des épisodes pour le test

V. MineRL:

Une bonne partie du code est identique à celle du cartpole. J'ai donc commencé par la duplication du code et ensuite l'adaptation pour qu'il fonctionne avec l'environnement MineLine-v0.

Environnement MineLine-v0:

Dans cet environnement, l'objectif de l'agent est de casser un bloc de diamants situé à l'un de ses deux côtés. Il existe 3 actions possibles {attack, left, right}. L'agent ne peut effectuer qu'une action à la fois.

Experience Replay:

Exactement la même procédure que pour CartPole.

Reseau de neurones:

Le réseau se compose de **4 couches**;

- La première couche est une couche de convolution avec 16 filtres, une taille de noyau de 8, un pas de 4 et une activation ReLU. Cela est suivi d'une couche de max pooling avec une taille de noyau de 2 et un pas de 2.
- La seconde couche est une autre couche de convolution avec 32 filtres, une taille de noyau de 4, un pas de 2 et une activation ReLU. Cela est suivi d'une couche de pooling adaptatif moyen avec une taille de sortie fixe de 7, et une couche de aplatissement qui convertit le tenseur de sortie en un tenseur 1D.
- La troisième couche est une couche fully-connected (dense) avec 128 unités et une activation ReLU.
- La quatrième et dernière couche est une autre couche dense avec le nombre d'unités égal à la taille de sortie du réseau, et une activation ReLU.

Le réseau utilise la perte en erreur quadratique moyenne (**MSE**) pour l'entraînement et l'optimiseur **Adam** pour mettre à jour les paramètres du réseau.

La méthode **forward** définit comment les données traversent le réseau. Elle prend le tenseur d'entrée, applique les couches définies et renvoie le tenseur de sortie.

Strategie d'exploration:

Même procédure que pour CartPole mais bien sûr, les tenseurs et leurs tailles ont été adaptés à ce problème. Cependant, l'idée reste exactement la même.

Target network:

Même procédure que pour CartPole.

Pretraitement:

J’ai choisi de convertir mon image en GrayScale. En théorie, la conversion en niveaux de gris peut réduire considérablement la taille des données d'entrée, ce qui peut accélérer le processus d'entraînement et économiser des ressources de calcul. En outre, la conversion en niveaux de gris peut éliminer les informations de couleur redondantes dans les images, ce qui peut améliorer les performances du modèle en termes de généralisation. Enfin, la conversion en niveaux de gris peut également simplifier le traitement des données d'entrée, en particulier pour les modèles de machine learning qui ne sont pas conçus pour traiter les images couleur. De plus, une étape de redimensionnement a été faite. Cependant, aucune amélioration notable a été aperçue dans mon modèle. J’ai donc décidé de continuer mon travail avec les images de base. J’ai retrouve la fonction que j’avais appliqué dans un ancien commit:

```
def process_state(self, state):
    # convert to grayscale
    state = rgb2gray(state)
    # resize
    state = resize(state, (self.im_height, self.im_width), anti_aliasing=True)
    # normalize
    state = state.astype(np.float32)
    state /= 255.0
    return state
```

Figure 6: code de prétraitement des images

Resultats:

hyperparamètres	valeur
eta	0.001
batch_size	64
episodes	100
epsilon	1.0
gamma	0.99
Loss function	MSELoss()
tau	0.01
Epsilon decay	0.995
Epsilon minimum	0.01
optimiseur	Adam
Taille memoire	100000

Figure 7: tableau représentant nos réglages des hyperparamètres au départ

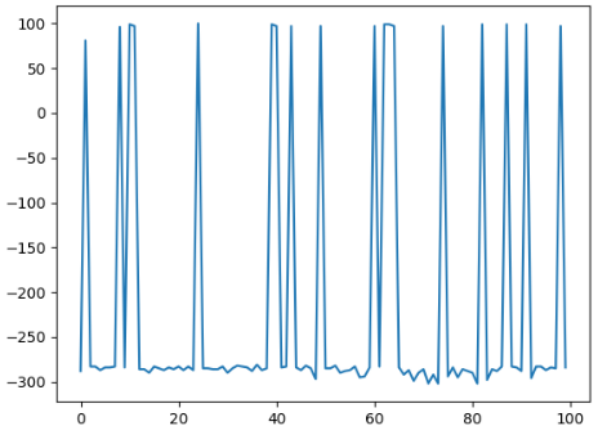

Figure 8: évolution des récompenses en fonction des épisodes pour l'apprentissage

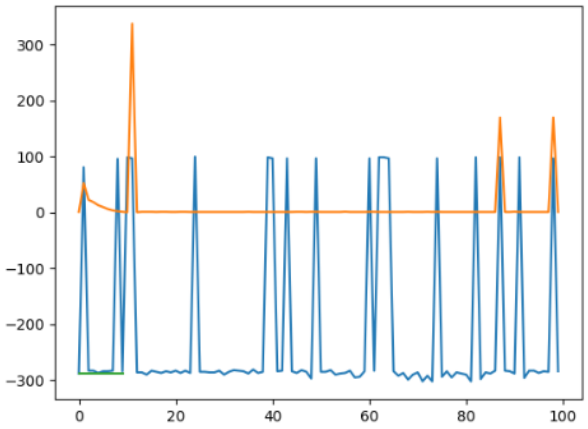

Figure 9: évolution des récompenses en fonction des épisodes pour le test

Comme on peut le voir, les résultats ne sont pas impressionnants. On voit que l'agent n'apprend pas vraiment et ne réussit que par pur hasard. Nous devons donc modifier nos hyperparamètres afin d'obtenir de meilleures performances. A mon avis, c'est peut etre du a un mauvais couplage exploration-exploitation lors de l'apprentissage.

Problemes:

Sans raison spécifique, ma VM n'avait plus d'espace pour travailler. Ceci doit être dû à une erreur de ma part que je n'arrive pas à identifier. Il était déjà trop tard pour demander de l'aide.

```
etudiant@tp-drl-17:~/apprentissage-par-renforcement-profond$ xvfb-run -a python3 minerl/main.py
mktemp: failed to create directory via template '/tmp/xvfb-run.XXXXXX': No space left on device
```

Figure 10: erreurs sur la VM

Solutions:

Cela ne m'a pas empêché de continuer à avancer au maximum. Je suis passé sur google colab et j'ai rechargé tout l'environnement afin de continuer mon travail.

La première idée pour améliorer l'apprentissage ici c'était de donner plus de liberté pour faire de l'exploration. Pour cela, j'ai augmenté **epsilon minimum** à 0.3 et **eta** a 0.1.

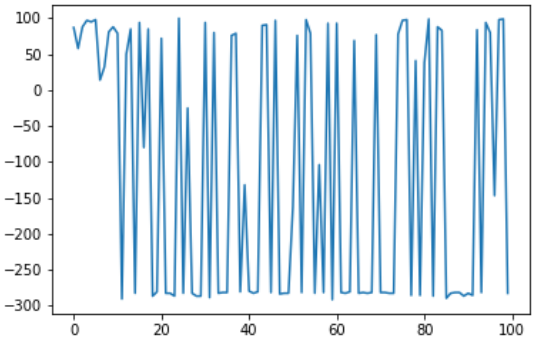

Figure 11: évolution des récompenses en fonction des épisodes pour l'apprentissage

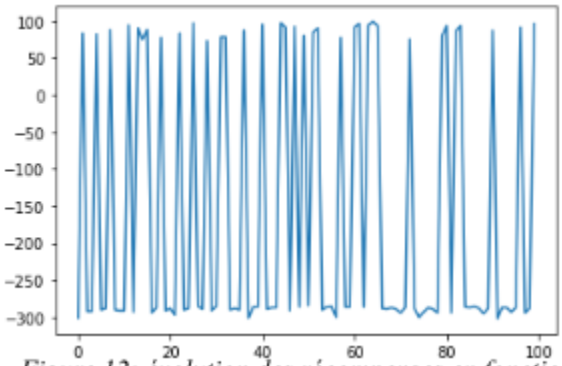

Figure 12: évolution des récompenses en fonction des épisodes pour le test

Comme on peut voir, avec plus de capacité d'exploration et donc, plus d'aléatoire, nos résultats s'améliorent significativement mais notre courbe ne converge pas.

Une dernière tentative d'amélioration que j'ai fait était la suivante:

- Epsilon decay: 0.99995
- ETA: 0.01
- Kernel de ma première couche: 6
- Epsilon minimum: 0.1

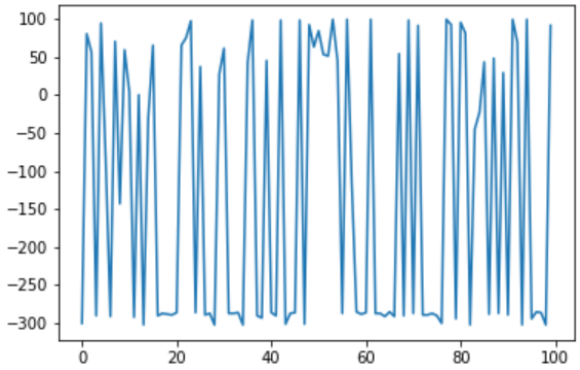

Figure 13: évolution des récompenses en fonction des épisodes pour l'apprentissage

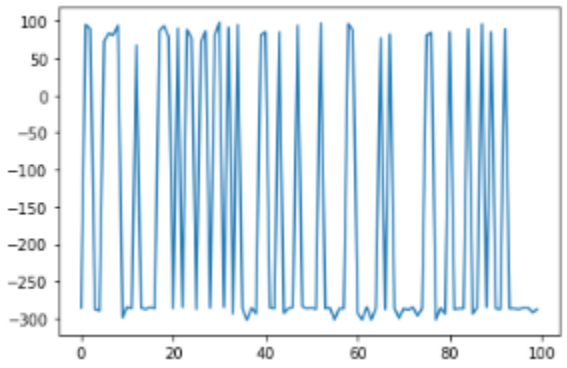

Figure 13: évolution des récompenses en fonction des épisodes pour le test

On remarque que nos résultats restent assez similaires. On va donc rester sur notre dernière combinaison d'hyper paramètres.

*** Le commit du travail fait sur google colab a été ajouté à la forge manuellement. ***

FrameStacking:

Je n'ai pas eu le temps de faire cette partie mais je me suis renseigné sur le fonctionnement du frame stacking, ses objectifs et ses contraintes.

Fonctionnement:

Le frame stacking consiste à concaténer ou empiler plusieurs frames ou images consécutives d'une séquence temporelle pour en créer une seule et unique entrée pour un modèle de machine learning. Cette technique peut être utilisée pour les modèles de reconnaissance de l'action dans les vidéos, les modèles de traitement du langage naturel ou les modèles de reconnaissance de la parole.

Objectifs:

Le frame stacking permet de capturer les relations spatiales et temporelles entre les différentes frames d'une séquence, ce qui peut améliorer les performances du modèle en termes de précision et de généralisation.

Contraintes:

Le frame stacking peut également augmenter considérablement la taille des données d'entrée et nécessiter plus de temps de calcul pour l'entraînement du modèle.

Autres environnements:

Malheureusement, je n'ai pas eu le temps de tester d'autres environnements.

VI. Conclusion:

Avant le début du TP, j'avais aucune connaissance sur le fonctionnement des DQN. Ce projet a donc été très enrichissant au niveau théorique et pratique sur le fonctionnement des réseaux de neurones en générale et l'apprentissage par renforcement profond plus spécifiquement. J'ai également eu l'occasion d'utiliser les environnements Gym que je peux encore exploiter et tester dans mon temps personnel.

Concernant les résultats, la partie cartpole a été complètement finie et les résultats sont satisfaisants. Cependant, la recherche de bons hyperparamètres a pris énormément de temps. Vu la contrainte du temps, je n'ai pas pu assez expérimenter pour obtenir de meilleurs résultats.

Finalement, la partie MineRL a été compliquée à implémenter comparé à la partie cartpole vu que les erreurs ne sont pas toujours évidentes. Mais cela dit, ça reste une expérience très enrichissante.