Thibaud PERRIN 11707841
Arnaud DAILLER 11711599

TP
Apprentissage profond par renforcement

PARTIE 1 : CART POLE

Le deep-Q network sur le Cartpole est implémenté dans le fichier “TP2_Cartpole.py”. Le fichier contient deux classes : RandomAgent() qui représente l’agent et la classe MutipleLayer() qui représente le réseau de neurone utilisé.

Réseau de neurone

Le réseau de neurone nommé model contient 2 couches cachés (entrée : 4 * 100 neurones, 2e : 100 * 100) et une couche sortie (100 * 2 nombres d’actions). Ce réseau de neurone utilise un Optimiser Adam avec un Learning Rate de 0.001, et utilise une erreur quadratique pour la backpropagation.

Mémoire

L’agent utilise une mémoire de taille 10000 contenant l’état (4 valeurs), l’action (1 valeur), l’état suivant (4 valeurs), la reward (1 valeur) et le done (boolean). On ajoute une ligne à la mémoire à chaque fois que l’agent effectue une action (fonction remember()). les fonctions showMemory() et getMemory() permettent respectivement de voir et de récupérer la mémoire.

Batch

La taille du Batch influe sur la qualité de l’apprentissage mais aussi sur le temps d'exécution, nous avons donc fixé la valeur 32 car elle donne des résultats satisfaisants et que le temps d'exécution est correct. Le batch est une partie de la mémoire construite avec la fonction random.sample().

Apprentissage

La fonction retry() effectue l’apprentissage de l’agent et est déclenché à chaque fois que l’agent effectue une action et que la taille de la mémoire est supérieur à la taille du batch. Pour chaque ligne du batch, on applique l’algorithme de Bellman, en calculant la Qvaleur de l'état courant à l’aide du réseau de neurone de base, puis en calculant la Qvaleur de l’état

suivant à l'aide du réseau de neurone dupliqué. Enfin on calcul l'erreur à partir de ces deux Qvaleur et on la rétro-propage dans le réseau de base.

Action

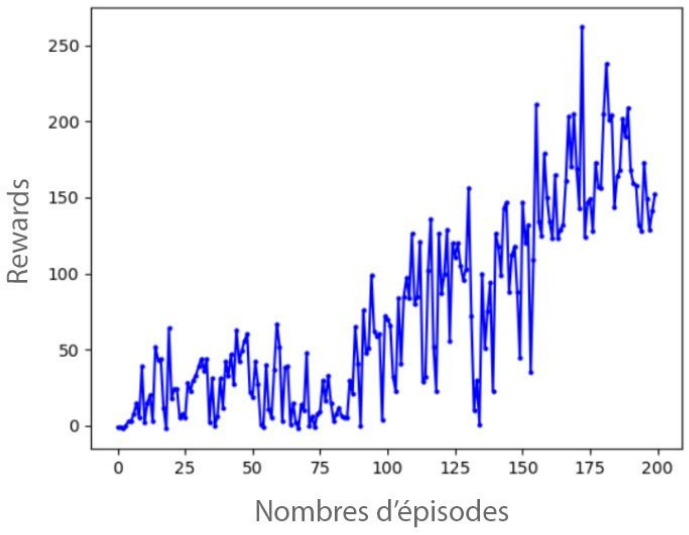
Pour l'action (fonction act()) nous avons utilisé la méthode greedy avec un epsilon fixé à 0.1 (paramètre qui fait varier l'exploration pour trouver la solution optimale, mettre un epsilon trop haut induit un nombre d'erreur trop importante de l'agent).

Target Network

Pour améliorer la stabilité des résultats de l'agent sur plusieurs épisodes nous utilisons un Target Network pour calculer la Qvaleur de l'état suivant. Nous le mettons à jour tous les 10 000 états dans la fonction retry() (fonction updateModel()). Mettre à jour le target Network trop souvent ne permet pas d'améliorer l'apprentissage.

Résultats

Nous avons lancé notre agent sur 200 épisodes, et nous observons les résultats suivants :



Le DQN fonctionne donc puisqu'on voit une augmentation du nombre de reward au fil des épisodes, les résultats restent très variables étant donnés l'epsilon pour l'exploration et le nombre d'épisode d'entraînement. Nous avons fixés une reward maximum de 300 points pour

réduire les écarts entre chaque résultats. On peut observer qu'au bout de 200 épisodes on arrive à une moyenne des récompenses stables.

PARTIE 2 : ATARI

Le Deep-Q Network sur l'environnement Atari est implementé dans le fichier "TP2_Atari.py". Il contient 3 classes :

- AtariPreprocessing : prétraitement des frames atari
- ConvModel : Contient le réseau de neurones utilisés
- RandomAgent : Agent

Pour les différents paramètres utilisés, nous avons utilisés ceux présentés dans l'article proposé dans le TP.

Preprocessing

Pour que accélérer l'apprentissage du réseau de neurone, nous avons intégré une phase de pré processing des frames du jeu Atari. Nous avons modifié le preprocessing du github gym pour que a chaque action de l'agent, on effectue cette action 4 fois (Frame-Skip) et on stocke les frames dans un Buffer de Frames de taille (4*84*84). On effectue ensuite des étapes de traitements de frame (reshape, greyscale, normalisation) puis on retourne un état correspondant aux 4 frames traités. Le greyscale et la récupération des frames sont implémentés dans la fonction step(), et le reshape et la normalisation sont implémentés dans la fonction _get_obs de la classe AtariPreprocessing.

Réseau de neurone

Le réseau de neurone est en deux parties (convolutionnel et linéair). On a donc 3 couches convolutionnelle avec les paramètres suivants :

- couche d'entrée : 4, 32, kernel_size=8, stride=4
- 2ème couche : 32, 64, kernel_size=4, stride=2
- 3ème couche : 64, 64, kernel_size=3, stride=1

et deux couches linéaires :

- 4ème couche : feature_size, 512
- couche sortie : 512, 4 (nombres d'actions)

Nous avons utilisé un Optimizer RMSprop avec un learning rate de 0.00025, un momentum de 0.95 et un epsilon de 0.01.

L’ensemble du réseau de neurone est implémenté dans la classe ConvModel.

Mémoire

La mémoire est la même que pour la partie 1, l’état et l’état suivant sont simplement de dimensions différentes (4*84*84).

Batch

On utilise un batch similaire à celui du cartpole.

Apprentissage

La fonction retry() effectue l’apprentissage de l’agent et est déclenché à chaque fois que l’agent effectue une action. On calcule la Q-valeur de l’ensemble des états courants du batch puis on calcul la Qvaleur suivant de l’ensemble des états suivant du batch avec des états ayant pour dimension (32*4*84*84 où 32 correspond à la taille du batch). On peut ensuite calculer l’erreur quadratique entre ces deux valeurs pour la rétro potager au réseau de neurone.

Action

Pour l’action (fonction act()) nous avons utilisé la méthode greedy avec un epsilon non fixé, l’epsilon est fixé à 1 au départ pour permettre beaucoup d’exploration et décrémente jusqu’à 0.1 au fil des épisodes. L’update de l’epsilon est implémenté dans la fonction changeEps().

Target Network

Le target Network reprend le même principe que celui de la partie 1 et est mis à jour tous les 10000 états comme recommandé dans l’article.

Résultats

Les premiers tests sur 600 épisodes n’ont pas été concluants et très long. Nous avons donc corrigé l’environnement qui paraissaient défaillant mais nous n’avons pas eu le temps de tester les résultats.