SOLOMON Maria
TREGER Pauline

TP du 10/12/2020 et du 17/12/2020
Apprentissage profond par renforcement

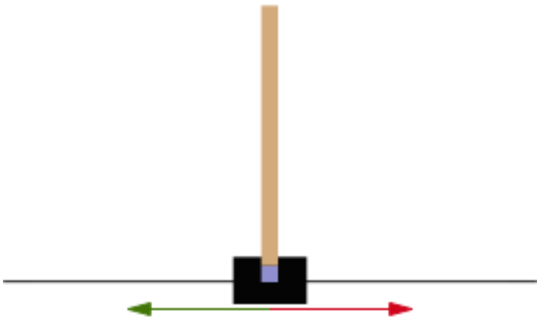
Introduction

Ce rapport est un compte-rendu du TP de l'UE Bio-Inspired Intelligence mettant en place des techniques d'apprentissage profond par renforcement. Nous décrirons donc ici toutes nos implémentations et expérimentations.

Notre code correspondant à ce TP se trouve sur le dépôt GitHub suivant :
https://github.com/pltreger/TP_bio_inpired

1. Partie 1 : Deep Q-network sur CartPole

1.1 Introduction au problème



L'objectif de cette partie est de créer un agent capable d'apprendre par tâtonnements et de résoudre finalement le problème du CartPole.
L'objectif est que l'agent (ici le rectangle noir) arrive à faire tenir le bâton à l'équilibre (le rectangle marron) en bougeant de droite à gauche.

1.2 Début

Question 1

Il nous est demandé, en nous inspirant du code fourni, de faire interagir un premier agent de manière aléatoire avec l'environnement CartPole-v1.
Pour cela nous modifions la ligne de l'initialisation de l'environnement en mettant :

```
env = gym.make('CartPole-v1')
```

Question 2

Nous cherchons maintenant à évaluer notre agent, en suivant l'évolution de la récompense obtenue à chaque épisode. Pour cela, nous proposons de suivre l'évolution de la somme de récompense par épisodes en fonction du nombre d'interaction de l'agent avec l'environnement en affichant des logs à chaque épisode. Nous modifions la partie sur l'implémentation des épisodes comme ci-dessous :

```
for i in range(episode_count):
    ob = env.reset()
    nbInteraction = 0
    totalReward = 0
    while True:
        action = agent.act(ob, reward, done)
        ob, reward, done, _ = env.step(action)
        nbInteraction = nbInteraction + 1
        totalReward = totalReward + reward
        if done:
            break
    print("Trace : episode : " + str(i) + " ; nb interactions : " + str(nbInteraction) + " ; recompenses : "
+str(totalReward))
```

Nous obtenons ainsi une partie des traces suivantes :

```
Trace : episode : 1 ; nb interactions : 44 ; recompenses : 44.0
Trace : episode : 2 ; nb interactions : 17 ; recompenses : 17.0
Trace : episode : 3 ; nb interactions : 11 ; recompenses : 11.0
Trace : episode : 4 ; nb interactions : 16 ; recompenses : 16.0
Trace : episode : 5 ; nb interactions : 18 ; recompenses : 18.0
Trace : episode : 6 ; nb interactions : 15 ; recompenses : 15.0
Trace : episode : 7 ; nb interactions : 18 ; recompenses : 18.0
```

1.3 Experience replay

L'objectif maintenant est d'implémenter le buffer pour stocker des données. Cela permettra à l'agent de stocker en mémoire toutes les interactions qu'il reçoit. Le buffer a une taille maximale et lorsqu'elle est dépassée, les nouvelles expériences remplacent les plus anciennes. L'agent va apprendre des expériences stockées dans son buffer, il choisira aléatoirement un minibatch d'expériences dans son buffer.

Question 3

Nous commençons par créer la classe buffer qui stock les données reçues.
Le buffer, permettra donc de garder en mémoire l'état de l'agent, son action, son état suivant, sa récompense, sa fin d'épisode.
Nous créons donc une liste de taille prédéfinie pour chaque élément à sauvegarder. Dès que le buffer a rempli sa mémoire, les nouvelles expériences remplacent les plus anciennes.

Question 4

Nous devons créer une méthode qui permet de récupérer un minibatch de données aléatoires dans le buffer. Cette méthode aura comme paramètre la taille que devra faire le minibatch.

La modélisation UML de la classe Buffer est donc la suivante :

SOLOMON Maria
TREGER Pauline

Buffer
states actions nextStates rewards done
add(data) get_minibatch(int sizeBatch)

Le buffer implémenté est donc indépendant de l'environnement, et stocke les données voulues.

1.4 Deep Q-learning

Nous allons maintenant utiliser des réseaux de neurones pour approximer l'espérance de récompense cumulée Q(s, a). Ensuite nous appliquerons l'équation de Bellman.

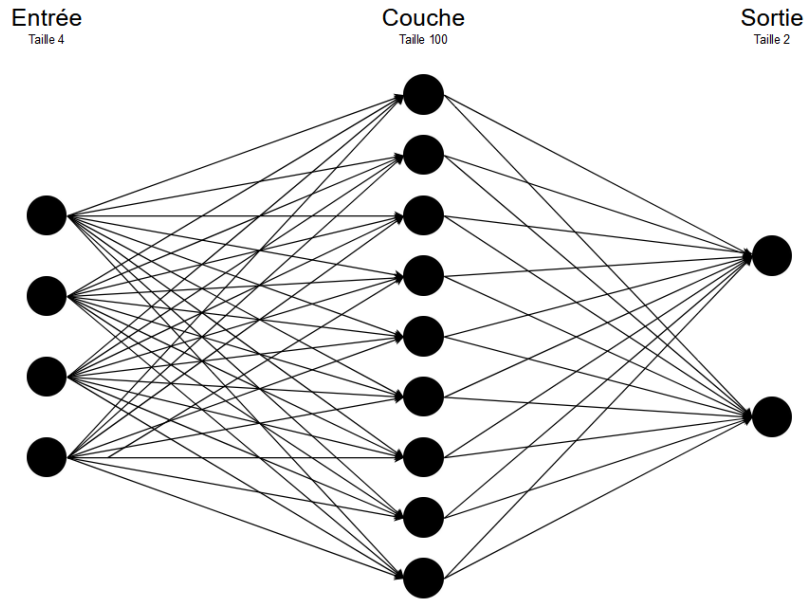
Question 5

A cette étape nous créons donc un réseau de neurone qui aura en entré l'état courant de l'environnement et fournira en sortie le score associé à chacune des actions.

Nous avons réalisé ce réseau de neurones dans le fichier Net.py

L'état courant de l'environnement s'obtient grâce à la méthode `env.reset()` ou `env.step(action)` qui permettent d'obtenir une observation de l'environnement composée d'une liste de 4 valeurs.

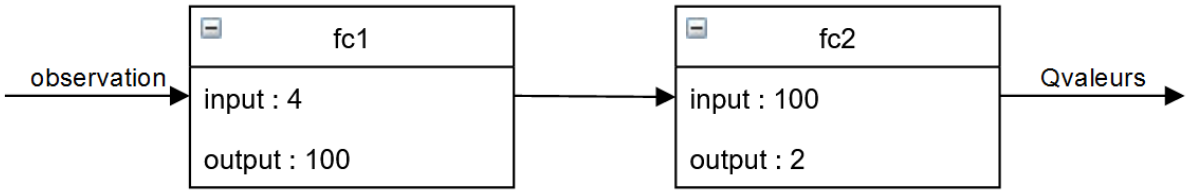
Le réseau de neurone est donc ici très simple et ne se compose que d'une couche :



En entrée le réseau recevra donc les 4 valeurs correspondant à l'état actuel. Puis parcoura une couche de 50 neurones. Pour fournir une sortie de taille 2 correspondant à la Q-valeur pour chacune des actions possibles.

SOLOMON Maria
TREGER Pauline

Pour cela, nous mettons en place deux couches fully-connected linéaires la manière suivante :



Pour l'instant, nous choisissons l'action de manière aléatoire.

L'utilisation du model ainsi que le choix des actions se fait dans la classe Neural_Network.

Question 6

Nous allons maintenant étoffer notre réseau de neurones en ne choisissant plus de manière aléatoire les actions, mais en utilisant une approximation de Q pour choisir les actions. Nous allons donc calculer la Q-valeur de chacune des actions pour chaque nouvel état. Nous mettrons également en œuvre deux stratégies d'exploration pour découvrir les nouveaux états.

Nous allons dans un premier temps mettre en place la méthode ε-greedy. Nous appliquons donc une politique ε-greedy qui permet d'obtenir une action par le réseau de neurone selon une probabilité. Avec une probabilité (1 - ε), le réseau donne la meilleure action, et avec une probabilité ε, le réseau donne une action aléatoire.

Ensuite, nous mettons maintenant en place l'exploration Boltzmann. Cette stratégie permet de choisir une action a_k avec la probabilité suivante :

$$P(s, a_k) = \frac{e^{\frac{Q(s,a_k)}{\tau}}}{\sum_i e^{\frac{Q(s,a_i)}{\tau}}}$$

Pour cela nous utilisons la méthode softmax du package torch.nn.functional, qui permet de réaliser l'équation suivante :

$$\mathrm{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Nous mettrons donc $x_i = Q(s,a_k)/\tau$

Question 7

A présent, nous allons permettre à notre agent d'apprendre. Pour cela, à chaque interaction l'agent choisira un minibatch aléatoirement dans son buffer et calculera les Q-valeurs. Nous appliquerons l'équation de Bellman et rétropropagerons l'erreur dans le réseau de neurone.

Nous ne commençons l'apprentissage seulement lorsqu'il y a assez d'expériences sauvegardées dans le buffer pour constituer un minibatch.

SOLOMON Maria
TREGER Pauline

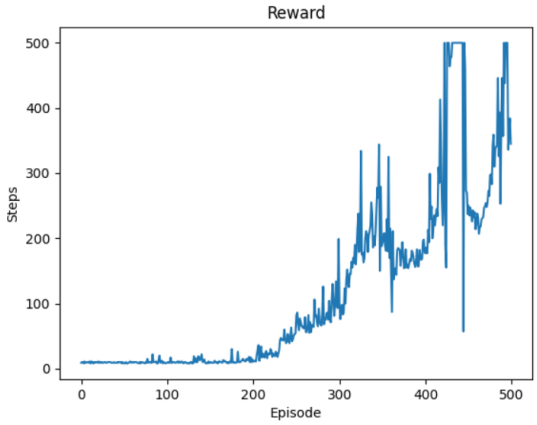
Question 8

Nous mettons maintenant en place deux modèles de réseaux de neurone :

- model_anticipation correspondant à $Q_\theta(s, a)$, pour prédire l'action à réaliser
- model_apprentissage correspondant à $Q'_\theta(s', a)$,pour l'apprentissage

Les poids anticipés seront au fur et à mesure mis à jour avec les poids définis par l'apprentissage. A chaque étape les poids du duplicat seront mis à jour à l'aide de la formule suivante : $\theta' = (1-\alpha)\theta' + \alpha\theta$ où θ' sont les poids du duplicat et θ les poids du réseau original. α est le pas de mise à jour, nous choisissons $\alpha = 0.01$.

Nous obtenons donc un agent qui apprend. Voici un exemple de résultat que nous pouvons obtenir avec les paramètres suivants :
Gamma = 0.99
Alpha = 0.005
Nous utilisons la méthode e-greedy avec epsilon = 0.01



Nous pouvons voir ici que l'agent apprend à maximiser ses récompenses.

2. Partie 2 : Environnement plus difficile : VizDoom

Nous allons maintenant essayer de voir si notre agent peut réussir à apprendre dans un environnement plus compliqué. Pour cela nous allons le tester dans l'environnement VizDoom.

VizDoom est basé sur le jeu vidéo classique de tir à la première personne, Doom. Il permet de développer des bots qui jouent au jeu.

Question 1

Dans un premier temps, nous construisons un nouvel environnement VizdoomBasic-v0 en remplaçant la ligne correspondante dans le code.

Ainsi la ligne que nous avions défini pour la première partie :

```
env = gym.make('CartPole-v1')
```

Devient :

```
env = gym.make('VizdoomBasic-v0')
```

SOLOMON Maria
TREGER Pauline

Nous allons également réduire la résolution des images de l'environnement d'une résolution initiale de 320x240, nous allons la réduire à 112x64. Nous mettons également l'image de l'observation en noir et blanc. Les observations étant définis par des captures d'image du jeu, cela permettra de faciliter l'apprentissage.
Pour cela, nous utilisons le code fourni dans le sujet du TP.

Question 2

Nous allons maintenant adapter notre code. Nous pouvons remarquer que seul l'observation est modifiée, en effet, dans l'environnement Cartpole, l'observation était composée d'une liste de 4 valeurs, dans ce nouvel environnement plus complexe VizdoomBasic-v0, nous avons maintenant une matrice de 4 dimensions : 1x1x112x64. Nous allons donc devoir modifier le Buffer pour stocker toutes ces valeurs.

De plus, nous pouvons observer que dans ce nouvel environnement, il n'y a plus 2 mais 3 actions possibles.

Question 3

Il nous faut à présent modifier le réseau de neurone pour réaliser un réseau de neurone convolutionnel.

Malheureusement, nous n'avons pas réussi à implémenter un réseau de neurone convolutif. Nous avons perdu beaucoup de temps à faire fonctionner l'environnement Vizdoom que nous n'avons pas pu accorder le temps nécessaire à nos recherches pour ce réseau de neurone.

Vous trouverez dans notre code, dans le fichier Net.py tous les tests que nous avons pu effectuer pour élaborer ce réseau de neurone entouré de balises ### DEBUT TESTS ### et ### FIN TESTS ###