TP du 10/12/2020 et du 17/12/2020

Apprentissage profond par renforcement

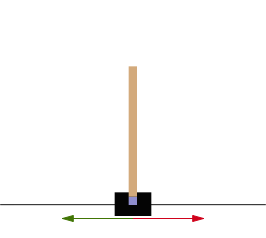
# Introduction

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 : METTRE LE GIT

# 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 :

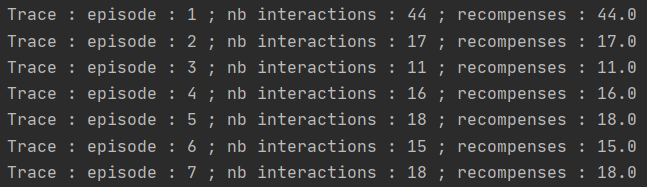
env = gym.make('CartPole-v1')

### 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 :



## 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 :



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.

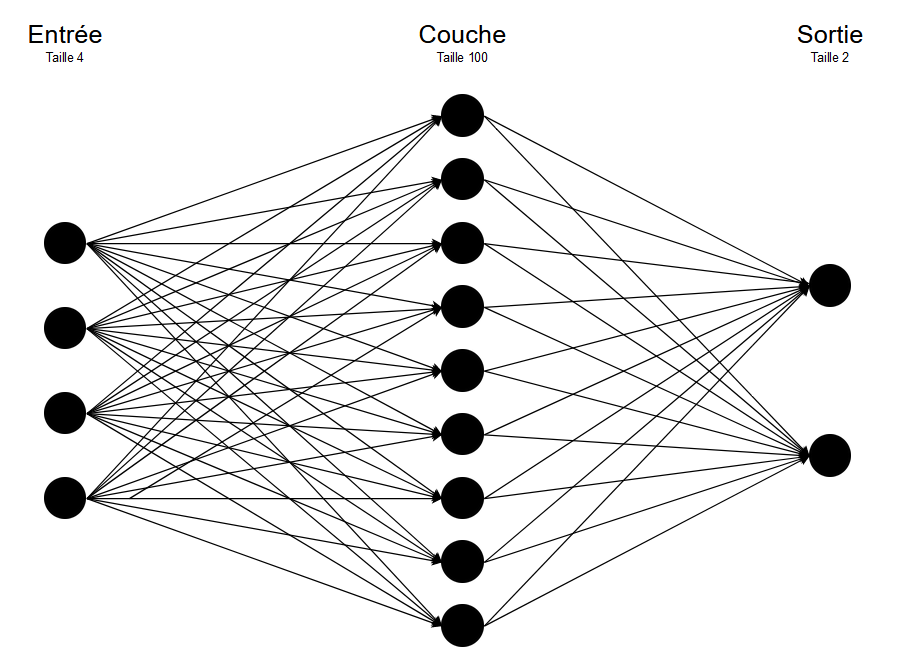
### Question 5

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

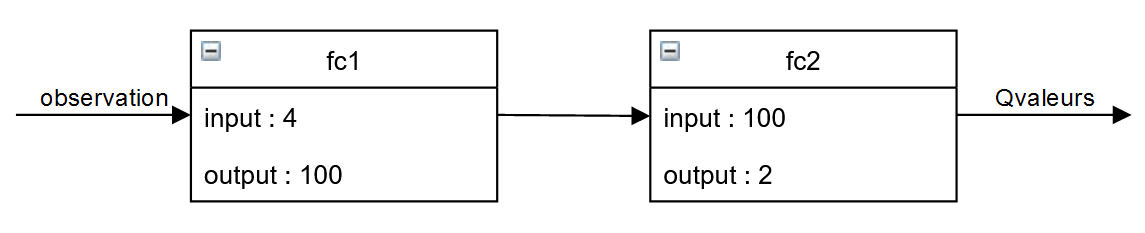
L’état courant de l’environnement s’obtient grâce à la méthode env.reset() ou env.step(action) qui permettent d’obtenir une observation de l’environnement composée d’une liste de 4 valeurs.

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.

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



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

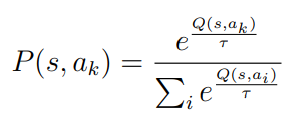
L’utilisation du model ainsi que le choix des actions se fait dans la classe Neural\_Network.

## 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 ak avec la probabilité suivante :



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



Nous mettrons donc xi = Q(s,ak)/τ

## 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.

## Question 8

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

* model\_anticipation correspondant à Qθ(s, a), pour prédire l’action à réaliser
* model\_apprentissage correspondant à Q’θ(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 : θ’= (1−α)θ’+αθ

où θ’ sont les poids du duplicat et θ les poids du réseau original. α est le pas de mise à jour, nous choisissons α = 0.01.