TP : Apprentissage profond par renforcement

Chupin Pierre-Henri
Jassigneux Marie

Partie 1 : Deep Q-network sur CartPole

Pour cette première partie nous avons comme mission d'implémenter une première version du Deep Q-network (DQN). Pour faire cela nous commençons sur un environnement simple avant d'aller vers un environnement plus compliqué. Notre premier agent est une plateforme qui a pour objectif de faire tenir en équilibre un bâton le plus longtemps possible

L'agent à 2 actions différentes. Il peut soit aller à gauche soit aller à droite.
L'espace d'observation elle a 4 valeurs (positions du bâton et de la plateforme)

Nous avons dû créer un réseau de neurones simple. Nous avons décidé de faire un réseau de neurones totalement connecté avec deux couches de neurones invisibles. Il a pour entrée les valeurs observables de l'environnement la première couche invisible à une taille d'entrée choisi ici 64 et de sortie choisi ici 64. La dernière couche a un nombre de sortie équivalant aux actions. On prend alors la valeur maximale entre toutes les valeurs pour avoir l'action à faire par l'agent.

Nous avons dû créer un Buffer de taille fixé par nous (50.000) afin d'y mettre ce qui sa passe après l'action fait par l'agent. En mettant ces informations dans un Buffer on permet à l'agent d'apprendre des expériences stockées dans celui-ci. l'agent va alors choisir un minibatch d'expérience afin de lui permettre d'être plus performant dans son apprentissage.

Pour faire attention au dépassement de mémoire dans le buffer on garde un paramètre position pour remplacer correctement les valeurs les plus anciennes une fois le buffer plein.

```
def push(self, value):
        if len(self.memoire) < self.cap:
                self.memoire.append(value)
        else:
                self.memoire[self.position] = value
        self.position = (self.position + 1) % self.cap
```

Nous avons utilisé comme stratégie d'exploration « epsilon greedy » car il est très simple à mettre en place et son efficacité est satisfaisante. Le fonctionnement d'epsilon greedy est très simple. On choisit avec une probabilité epsilon de prendre une action aléatoire et avec une probabilité 1-epsilon de choisir la meilleure action. l'intérêt de cette stratégie est que cela permet de forcer l'agent à tenter de nouvelle action qui peut s'avérer meilleur. Afin de rendre cette stratégie meilleure nous avons décidé de faire décroître l'epsilon jusqu'à une valeur minimale cela permet au départ de parcourir tous l'espace d'action possible pour être sur de prendre la meilleure route possible.

```
if from_file:
        self.EPS_START = 0.004
else:
        self.EPS_START = 1
self.EPS_DECAY = 0.997
self.EPS_END = 0.004
```

Voici la mise à jour du epsilon et comment on le fait décroître

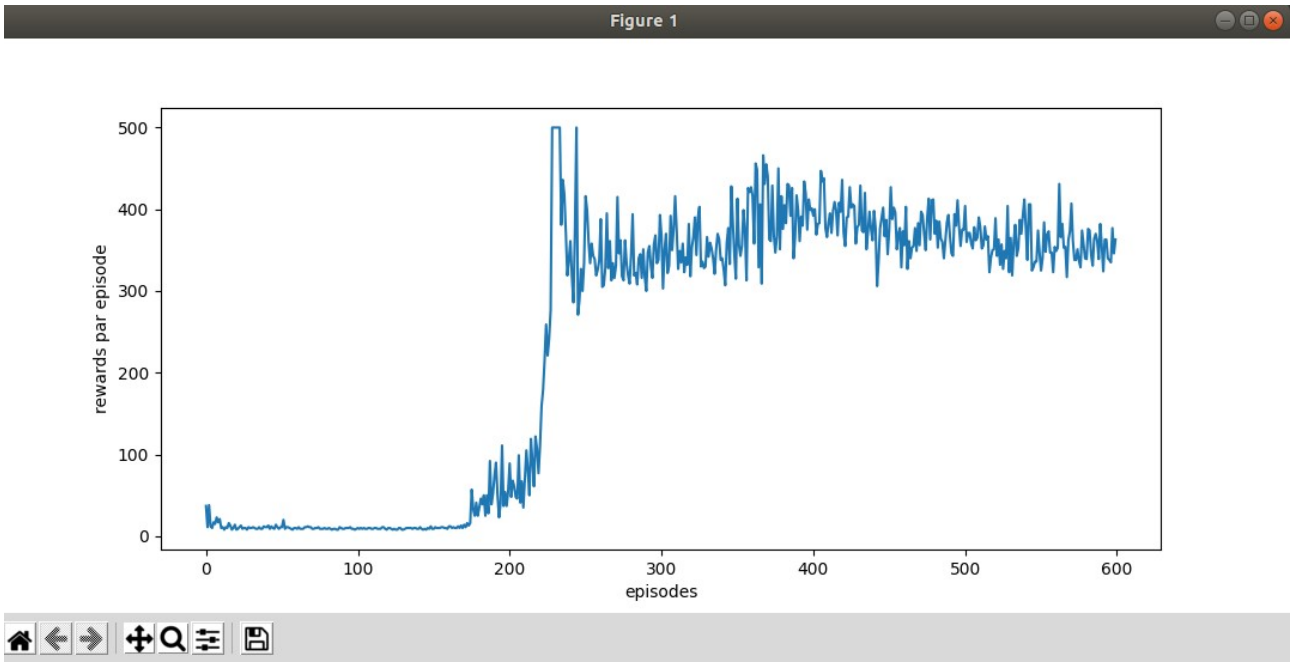
```
if self.EPS_START > self.EPS_END:
        self.EPS_START *= self.EPS_DECAY
```

Nous avons utilisé un target network (réseau cible dans le code) afin de stabiliser l'apprentissage. Il y a deux manières de mettre à jour le target network. Nous avons choisi de mettre à jour petit à petit le duplicat à chaque étape d'apprentissage. Nous utilisons un alpha = 0,005 pour cela.

```
self.reseau_cible = rNeurones(self.espace_observation.shape[0],
                              self.espace_action.n,64,64)
```

```
for t_param,l_param in zip(self.reseau_cible.parameters(), self.r_Neurones.parameters()):
        t_param.data.copy_(self.alpha * l_param.data + (1 - self.alpha) * t_param.data)
```

Voici ce que donne notre agent lors de son entraînement :



Nous avons fait en sorte de pouvoir enregistrer l'apprentissage de notre agent afin de ne pas avoir à relancer tout l'apprentissage à chaque fois qu'on aimerait utiliser notre agent. Pour cela nous avons défini une fonction qui utilise la fonction save de torch

```
def save_param(self):
        torch.save(self.r_Neurones.state_dict(), "saved_params/cart_pole.pt")
```

Pour filmer les comportements de notre agent nous avons utilisé comme dit dans le sujet wrapper mais nous avons un souci car une la vidéo créée nous n'arrivons pas à les lancer et nous ne savons pas pourquoi.

Partie 2 : Deep Q-network sur Vizdoom

Pour le deuxième environnement plus complexe il s'agit de Vizdoom. Ici la première chose à faire est de ne pas oublier à faire un pre-processing sur l'image de l'environnement pour n'avoir que du noir et blanc et une résolution plus petite. Cela est indispensable car notre machine n'était pas du tout assez puissante pour réussir à apprendre dans un environnement aussi massif.

```
def preprocess(self, img):
        img = img[0]
        img = skimage.transform.resize(img, self.resolution)
        #passage en noir et blanc
        img = skimage.color.rgb2gray(img)
        #passage en format utilisable par pytorch
        img = img.astype(np.float32)
        # print(img.shape)
        img = img.reshape((1, self.resolution[0], self.resolution[1]))
        return img
```

Pour ce nouvel environnement nous avons du changer le réseau de neurones et le faire passer en convolutionnel.