Apprentissage profond par renforcement

Claire Pillet & Cyril Reymond

1 Introduction

Ce projet a pour objectif de créer un agent qui soit capable de résoudre un problème donné dans un environnement précis à l'aide d'algorithmes d'apprentissage profond par renforcement. L'agent apprend les actions à effectuer à partir de ses expériences avec l'environnement de sorte à optimiser sa récompense.

Les environnements que nous étudions dans ce projet sont le Cartpole-v1 et le BreakoutNoFrameskip-v4 de OpenAI Gym. Ce framework offre la possibilité de tester les algorithmes d'apprentissage profond sans avoir à coder l'environnement dans lequel l'agent évolue. Ainsi, un même agent peut être utilisé sur différents environnements.

Cartpole Une tige est accrochée verticalement à un poids. Cette tige doit rester droite et ne pas passer en dessous de la ligne verticale. L'agent obtient une récompense de 1 à chaque épisode si la tige reste droite. Un épisode, i.e un essai de l'agent sur l'environnement, s'arrête si le poids s'est trop éloigné du centre ou que la tige possède un angle trop important. L'agent possède deux actions qui sont de pousser le poids vers la gauche ou vers la droite.

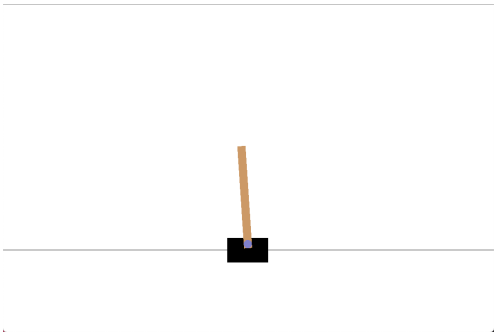

Figure 1: Interface de l'environnement Cartpole-v1

Atari Breakout Cet environnement est nettement plus complexe car un état est une image avec des pixels RGB. Pour simplifier l'apprentissage, nous traitons l'image en amont en supprimant les couleurs, et en ne considérant uniquement les quatre dernières frames. Nous obtenos ainis des états multidimensionels de taille 4x84x84. L'agent doit arriver à jouer sur cet environnement, c'est-à-dire casser toutes les briques du niveau.

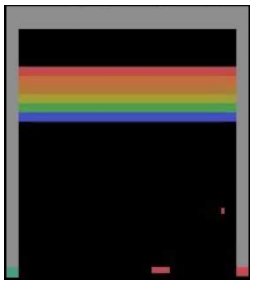

Figure 2: Interface de l'environnement Breakout

2 Code

Ce projet est réalisé avec le language Python. Le code comporte quatre fichiers Python : AgentCartpole, AgentAtari, ReplayMemory et Network.

Pytorch La réalisation des réseaux de neurones et algorithmes est faite avec la bibliothèque Pytorch. Elle offre tous les modules nécessaires à la construction d'un réseau de neurones entièrement paramétrable, de faire de l'apprentissage et de sauvegarder un modèle entraîner pour le réutiliser ultérieurement. Les modèles sauvegardés sont situés dans le répertoire *Model* du code source.

Q-Network Nous utilisons deux réseaux de neurones pour faciliter l'apprentissage de l'agent. Le premier, *eval network*, permet de calculer les Q valeurs de l'état courant et le second, *target network*, sert à calculer les Q valeurs de l'état suivant. Progressivement, nous mettons à jour l'état du *target network* avec la formule $w^{(target)} = (1 - \alpha)w^{(target)} + \alpha w^{(eval)}$, où w est le poids du réseau de neurones et α le pas de mise à jour.

Classes Agent Nous avons décidé de séparer en deux agents pour simplifier la compréhension du code. Toutefois, le code est très similaire. La seule différence se fait au niveau du réseau de neurone utilisé. Le DQN implémenté est un double DQN comme suggérer dans les bonus. Les deux méthodes principales de l'agent sont $act()$ et $learn()$. La première permet à l'agent de choisir son action en fonction de la Q valeur calculée par le réseau. La seconde méthode sert à l'apprentissage de l'agent. Nous y implémentons l'équation de Bellman et la rétropropagation de l'erreur entre les différentes couches du réseau de neurones. Nous avons choisis la politique d'exploration epsilon greedy

Classe ReplayMemory Cette classe symbolise la mémoire de l'agent. Après chaque action, une interaction est stockée en mémoire avec pour données l'état courant, l'action effectuée, l'état suivant, la récompense associée et un booléan indiquant si l'épisode est terminé ou non. Dans une première version de cette classe, nous avons implémenter l'interaction sous la forme d'un tuple. Toutefois, la structure figée d'un tuple en Python ne nous permettait pas de gérer les états à plusieurs dimensions, comme c'est le cas pour l'environnement Breakout. Ainsi, nous avons modifié la structure de la mémoire en créant un Tensor de la bonne taille pour chaque information stockée. Les méthodes $push(state, action, n_state, reward, done)$ et $pop()$ permettent respectivement d'ajouter une interaction en mémoire et de choisir aléatoirement un échantillon de la mémoire.

3 Deep Q-network sur Cartpole

3.1 Implémentation

Pour l'environnement Cartpole, nous avons définis un réseau de neurones simple. En effet, un état est composé de quatre informations regroupées en une seule dimension, il n'est donc pas nécessaire de créer un réseau à convolutions. Ainsi, notre réseau est composé de deux couches linéaires, c'est-à-dire que tous les neurones sont reliés entre eux. En entrée, la première couche accepte quatre éléments et un nombre de données en sortie à faire varier. Entre les deux couches, nous utilisons un rectifieur linéaire pour augmenter les performances du réseau. La deuxième couche donne en résultat les Q valeurs pour chaque action, donc retourne deux valeurs.

```
class DQN(nn.Module):
def __init__(self, hidden_dim):
    nn.Module.__init__(self)
    self.l1 = nn.Linear(4, hidden_dim)
    self.l2 = nn.Linear(hidden_dim, 2)

def forward(self, x):
    x = F.relu(self.l1(x))
    x = self.l2(x)
    return x
```

Les paramètres de l'algorithme d'apprentissage profond sont décisifs pour que l'agent apprenne efficacement. Les tableaux suivants résument notre paramétrage.

Dimension cachée	Mémoire	Minibatch	Episodes	Etapes max par épisode	Actions
50	100000	32	1000	200	2

Table 1: Paramètres utilisés pour l'agent

3.2 Résultats

Afin d'augmenter les performances de l'algorithme tournant sur nos machines, nous avons limité le nombre de récompense maximal à 200. Cette valeur nous sert donc d'indicateur pour savoir si l'agent à réussi l'épisode en cours. Les résultats que nous présentons sont réalisés sans modèle préalablement sauvegardé.

Epsilon	Epsilon Minimum	Epsilon Decay	Gamma	Alpha	Learning rate
0.9	0.1	0.99	0.9	0.005	$1e-3$

Table 2: Paramètres utilisés pour l'apprentissage

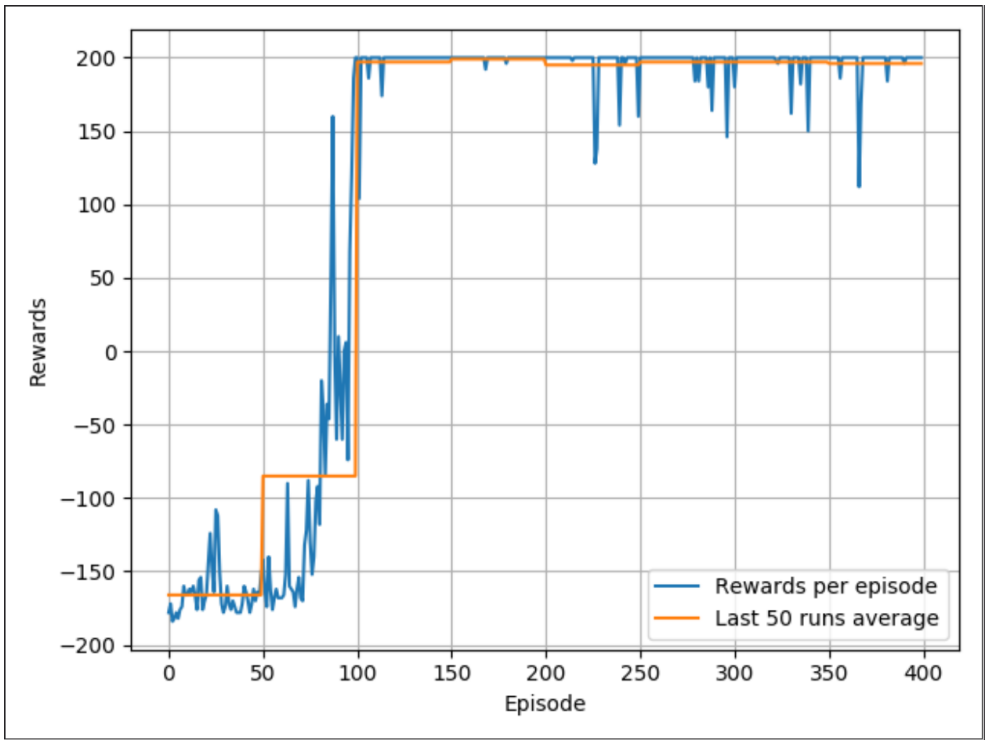

Figure 3: Résultats obtenus pour 400 épisodes

La courbe bleue montre l'évolution des récompenses de l'agent par épisode, et la jaune la moyenne des récompenses sur les 50 derniers essais. Cette dernière courbe permet de montrer que l'agent atteint la récompense maximale pour notre expérience après 100 épisodes.

4 Breakout Atari

4.1 Implémentation

L'implémentation générale est similaire à l'agent pour l'environnement Cartpole. Toutefois, la différence se trouve au niveau de la multidimensionalité des états et donc du réseau de neurones. Dans cette partie, nous utilisons un réseau de neurones à convolutions. L'image est traitée par petites parties, que l'on peut appeler tuile. Chaque tuile est analysée par un neurone. L'intérêt de cette technique réside dans le fait que tous les neurones ont le même paramétrage.

```
class Convolutional(nn.Module):
def __init__(self):
    super(Convolutional, self).__init__()
    self.conv1 = nn.Conv2d(4, 32, 8, 4)
    self.conv2 = nn.Conv2d(32, 64, 4, 2)
    self.fc1 = nn.Linear(5184, 512)
    self.fc2 = nn.Linear(512, 4)

def forward(self, x):
    x = F.relu(self.conv1(x.reshape([-1, 4, 84, 84])))
    x = F.relu(self.conv2(x))
    x = F.relu(self.fc1(x.view(x.size(0), -1)))
    return self.fc2(x)
```

Le réseau est composé de quatre couches : deux couches à convolutions, et deux couches linéaires. De la même manière que précédemment, nous utilisons la fonction d'activation ReLU entre les couches.

Pour ce qui est l'implémentation lié aux actions nous gérons nous-même l'envoie de la balle après une défaite et au lancement de la partie. De plus, nous avons choisi de faire des tests avec uniquement 2 actions possibles à notre agent, RIGHT et LEFT. En effet lors de nos tests si nous laissions les 4 actions disponibles relançais sans cesse la balle et nos résultats convergeais toujours vers un score de 0. En réduisant le nombre d'actions nous espérons aussi avoir des résultats avec moins d'itération.

Implementation à 4 actions (self.p['N$_A CTIONS$'] = 4)

```
def act(self, state):
    r = random.random()

    if r > self.p['EPSILON']:
        x = torch.FloatTensor(state).to(device)
        q_value = self.eval_cnn(x)
        action = torch.argmax(q_value).item()
        return action
    else:
        action = random.randint(0, self.p['N_ACTIONS'] - 1)
        return action
```

Implementation à 2 actions (self.p['N$_A CTIONS$'] = 2)

```
def act(self, state):
    r = random.random()

    if r > self.p['EPSILON']:
        x = torch.FloatTensor(state).to(device)
        q_value = self.eval_cnn(x)
        max=2;
        if torch.abs( q_value[0][3]) > torch.abs(q_value[0][2]):
            max=3
        action = max
        return action
    else:
        action = random.randint(0, self.p['N_ACTIONS'] - 1) +2
        return action
```

Voici les paramètre utilisé pour l'apprentissage de notre réseau à base de convolution.

Mémoire	Minibatch	Episodes	Etapes max par épisode	Actions
10000	64	200	200	4

Table 3: Paramètres utilisés pour l'agent

Epsilon	Epsilon Minimum	Epsilon Decay	Gamma	Alpha	Learning rate
0.9	0.01	0.99	0.9	0.005	$1e-3$

Table 4: Paramètres utilisés pour l'apprentissage

4.2 Résultats

Faute d'ordinateur suffisamment performant, nous avons fais des tests sur un nombre d'épisodes petit pour percevoir une tendance mais insuffisant pour avoir des résultats concluants.

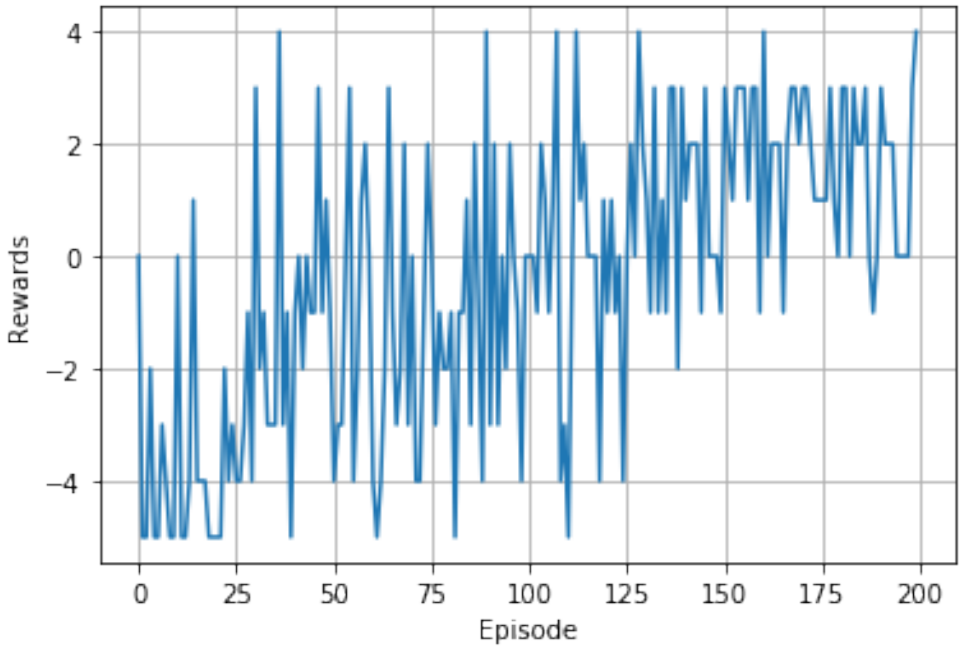

Figure 4: Résultats obtenus pour 200 épisodes avec 4 actions

Ici les résultats ne sont pas tres concluant et le réseau apprend peu sur aussi peu d'épisodes mais un apprentissage est tout de même visible .Et on constate que l'erreur de notre réseau décroit , laissant pensez qu'elle va finir par converger.

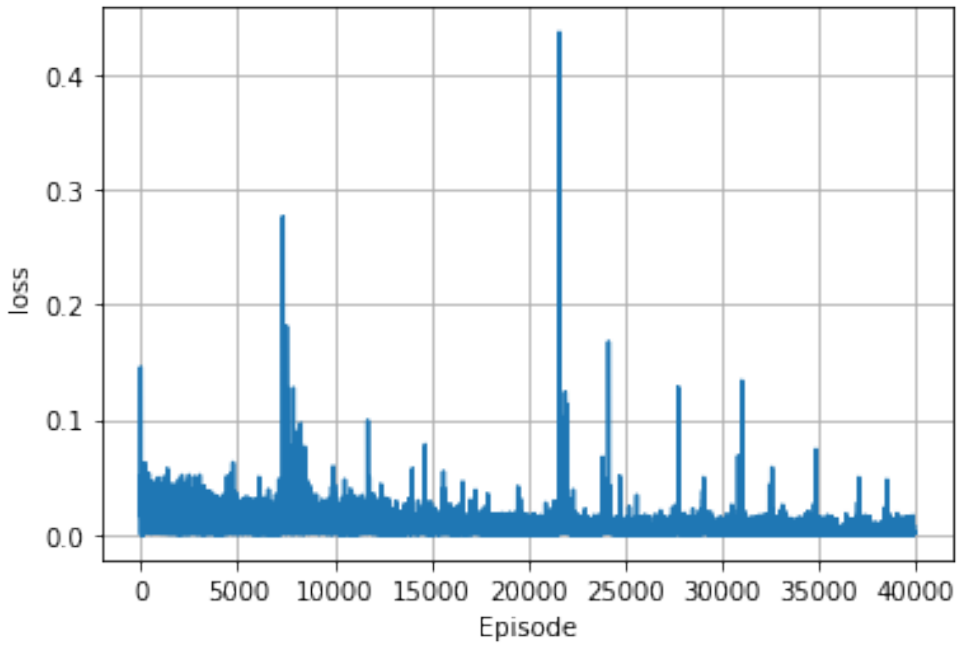

Figure 5: Erreur obtenus pour 200 épisodes avec 4 actions

Les résultats avec 2 actions sont au final moins encourageant, car les récompenses augmentent moins même si la chance d'exploration baisse. De même l'erreur semble moins converger. Nous retenons donc la solution avec 4 actions.

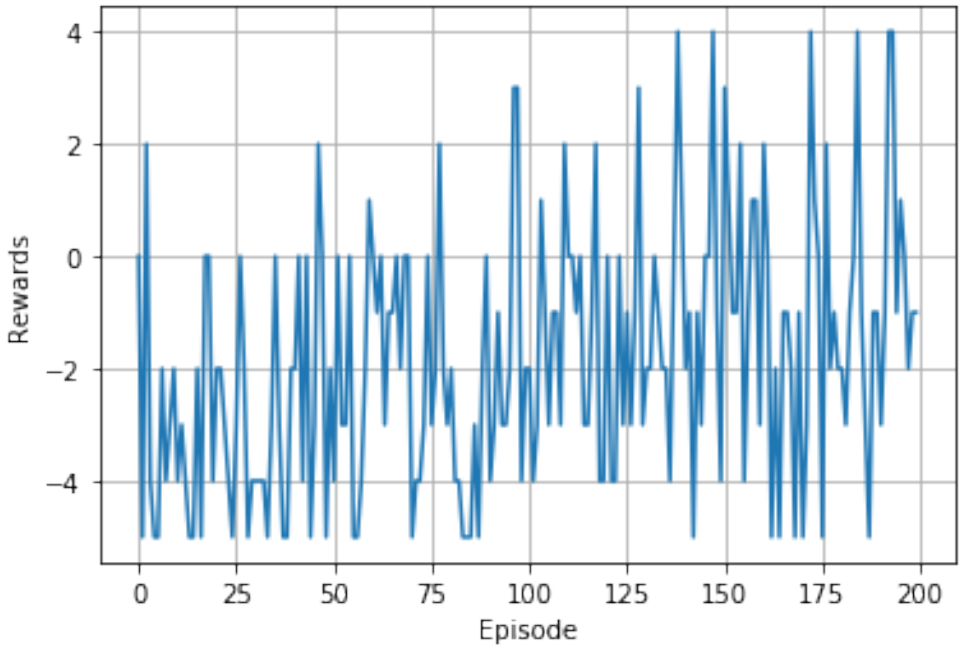

Figure 6: Résultats obtenus pour 200 épisodes avec 2 actions

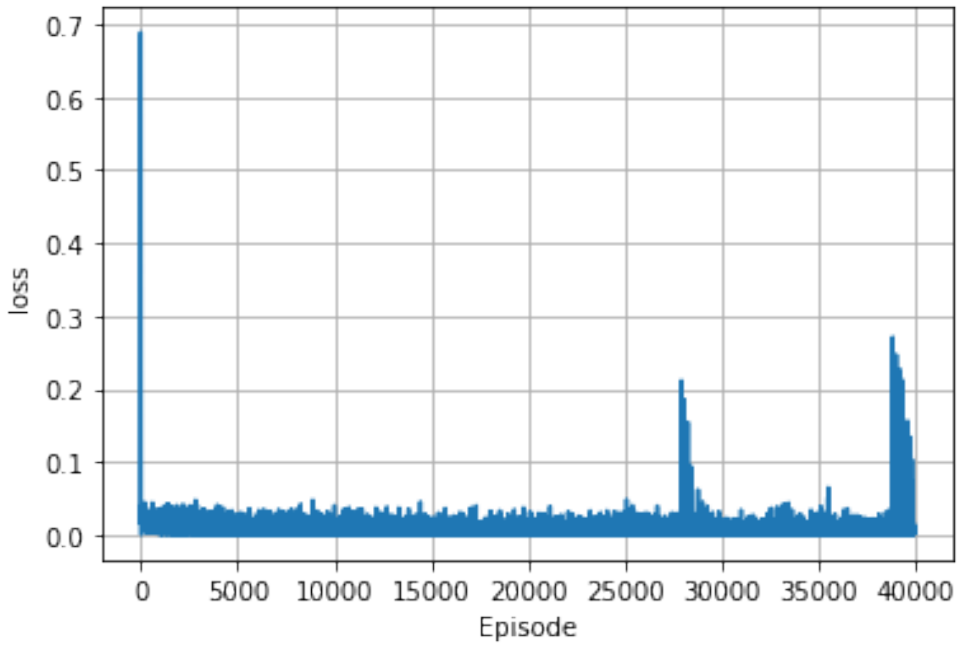

Figure 7: Erreur obtenus pour 200 épisodes avec 2 actions

5 Conclusion

L'apprentissage profond par renforcement est un outil intéressant pour résoudre des problèmes fermés. Les deux environnements proposés dans ce projet sont différents. Le premier ne nécessite que très peu de ressources car les actions sont limitées. L'avantage de cet environnement Cartpole est qu'il nous permet de tester rapidement nos algorithmes. L'apprentissage profond par renforcement montre très vite ces limites. Pour que l'agent puisse jouer au jeu Atari Breakout, il faudrait que le réseau soit entraîné pendant plusieurs heures. Ce sont donc des techniques intéressantes mais qui nécessitent de grandes puissances de calculs.