Université Claude Bernard Lyon 1

Intelligence Bio-Inspirée
TP

Apprentissage par renforcement profond

Élèves :
Nathan Etourneau
Paul Flagel

Enseignants :
Alexandre Devillers
Frederic Armetta

GitHub : `https://github.com/paulflagel/IBI-tp2-deep_reinforcement_learning`

24 janvier 2022

Table des matières

Introduction

L'objectif de ce TP est de se familiariser avec l'apprentissage par renforcement profond, par la méthode du *Deep Q-Learning*. Nous utiliserons pour cela le framework PyTorch pour la partie agent, et les frameworks Gym (développé par OpenAI) et MineRL pour la partie environnement.

N.B : Tous les résultats présentés dans ce rapport ont été ajoutés sous forme de gif sur le repo github du projet afin de mieux visualiser les performances des agents.

1 Deep Q-Learning

Le Q-Learning est une technique d'apprentissage par renforcement qui vise à apprendre une politique pour un agent, dans un environnement décrit par un processus de décision Markovien (MDP).

Pour de tels processus, l'équation d'optimalité de Bellman s'écrit :

$$Q^*(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q(s',a')\right]$$

Ainsi, si l'on connaît une fonction de Q-valeur optimale, la stratégie optimale consiste à prendre l'action maximisant la valeur attendue pour $r + \gamma \max_{a'} Q(s',a')$.

Le principal problème est que l'on ne connaît pas la fonction de valeur Q, la seule chose que nous pouvons faire est l'estimer itérativement à partir d'observations des récompenses obtenues lors des transitions de l'agent.
Dans le cadre du Q-Learning, cette fonction est d'abord initialisée de manière aléatoire et est ensuite apprise itérativement à partir d'exemples, par descente de gradient. La règle de mise à jour est :

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$$

Et la politique ε-gloutonne associée est :

$$\pi(s) = \begin{cases} \underset{a \in A}{\text{argmax}}\, Q(s,a) & \text{avec probabilité } 1-\varepsilon \\ \text{action aléatoire de A} & \text{avec probabilité } \varepsilon \end{cases}$$

Le problème est que sauf cas très rare, il y a un nombre d'états bien trop important pour tous les explorer (explosion combinatoire), et qu'il est nécessaire de pouvoir reconstruire cette fonction à partir de features extraits des états : des états très similaires ne sont pas rigoureusement identiques et sont traités différemment par l'agent. Cela rend l'apprentissage très lent, d'autant plus que le stockage des états et de la table des valeurs requiert énormément de place en mémoire, comme le montre la figure 1.

Le Deep Q-Learning est une extension du Q-Learning, où la fonction donnant une valeur aux couples `(état, action)` n'est plus apprise, mais approximée à l'aide de réseaux de neurones. Cette valeur est approximée à partir de représentations des états, et de l'historique des transitions `(state, action)`.

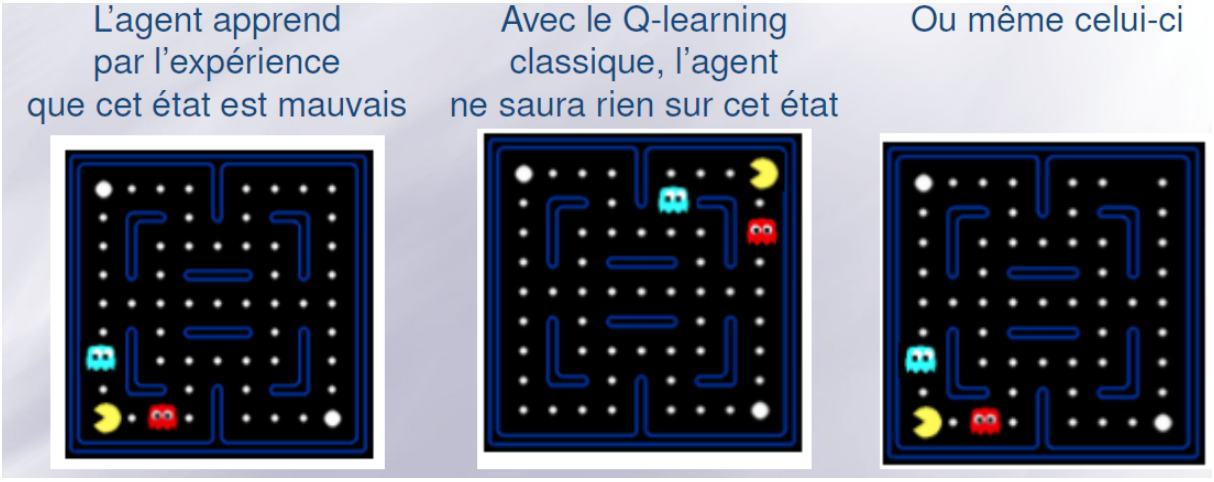

FIGURE 1 – Démonstration d'une difficulté du Q-Learning

2 CartPole-v1

Le premier environnement on nous avons essayé d'implémenter le Deep Q-Learning est l'environnement `Cartpole-v1` de Gym. Cet environnement consiste en un chariot sur un rail pouvant se déplacer horizontalement. Sur ce chariot, il y a un bâton, relié au chariot par une liaison pivot. Le chariot se déplace sans frotter sur son rail, et la liaison pivot est parfaite donc non dissipative.

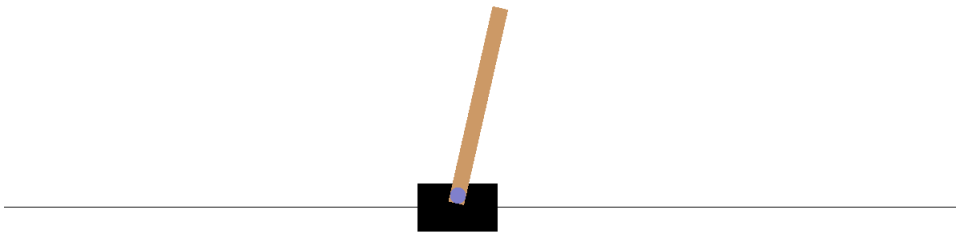

FIGURE 2 – Jeu du CartPole

Le but du problème est de maintenir le bâton vertical en contrôlant le mouvement vertical du chariot. Les équations sont :

$$\ddot{\theta} = \frac{g \sin\theta - \cos\theta \left(\frac{F + m_p l \dot{\theta}^2 \sin\theta}{m_c + m_p} \right)}{l(\frac{4}{3} - \frac{m_p \cos^2\theta}{m_c + m_p})}$$

$$\ddot{x} = \frac{F + m_p l(\dot{\theta}^2 \sin\theta - \ddot{\theta}\cos\theta)}{m_c + m_p}$$

Avec $m_p = 0.1$ kg (masse du baton), $m_c = 1$ kg (masse du chariot), $g = 9.8$ m/s^2 (accélération de la pesanteur), $l = 0.5$ m (demi-longueur du bâton). L'intégration a lieu avec la méthode d'Euler, avec un pas de 0.02 s.

Les actions possibles sont les suivantes : on peut appliquer une force de 10 N en direction de la gauche (action 0), ou en direction de la droite (action 1).

Les états sont de dimension 4, de la forme $(x, \dot{x}, \theta, \dot{\theta})$. La position x peut prendre des valeurs comprises entre -2.4 m et 2.4 m, et l'angle θ peut prendre des valeurs entre -15° et 15°. Autrement, l'épisode est terminé.

L'initialisation de la simulation est une initialisation d'un état aléatoire de manière uniforme : $(x, \dot{x}, \theta, \dot{\theta}) \sim \boldsymbol{U}([-0.05, 0.05]^4)$.

Un épisode se termine si l'angle ou la position dépasse les bornes mentionnées précédemment, ou si le bâton tient en équilibre dans les bornes autorisées pour 500 pas de temps. L'agent obtient une récompense de 1 à chaque pas de temps où le bâton est dans un domaine licite. Le problème est considéré résolu si l'agent obtient une récompense d'au moins 475 moyennée sur les 100 derniers épisodes.

2.1 Agent aléatoire

Nous implémentons d'abord un agent effectuant des actions aléatoires dans son environnement. Comme attendu, la courbe des performances de l'agent figure 3 montre de faibles résultats et aucune progression : environ 23 steps de récompense moyennée sur 100 épisodes.

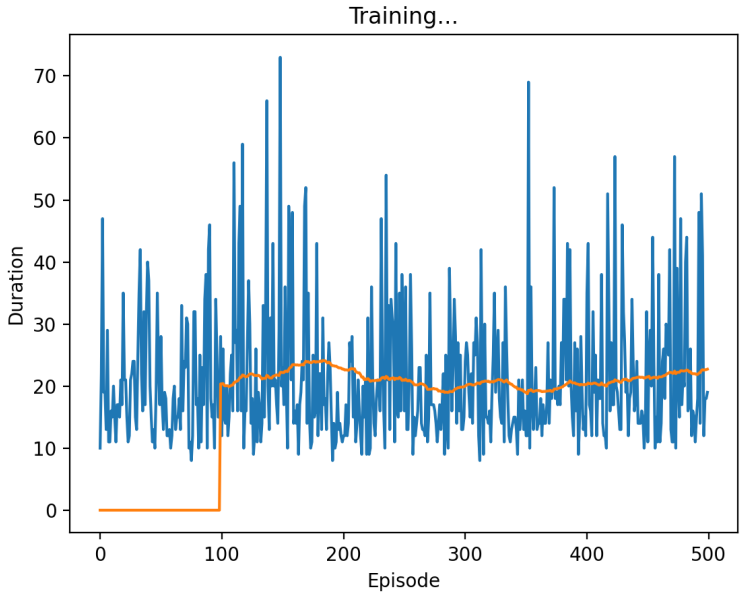

FIGURE 3 – Performances de l'agent random après 500 épisodes. La courbe bleue représente la performance de l'agent à chaque épisode, la courbe orange représente le score moyen sur les 100 derniers épisodes.

Cependant, la boucle de l'agent nous servira de base pour l'entraînement lors de la partie suivante.

2.2 Deep Q-Network

Pour que l'agent apprenne à maintenir le cartpole en équilibre, nous implémentons donc un deep Q-network constitué d'un buffer d'expériences, et d'un réseau de neurones dans un premier temps.

2.2.1 Experience Replay

Nous commençons par implémenter la mémoire de l'agent. Le DQN sauvegarde toutes dans la mémoire toutes les transitions entre états sous la forme d'un tuple (état, action, état suivant, récompense). Cette mémoire est un buffer circulaire ayant dans notre cas une taille maximale de 10000. Une fois le buffer rempli, les nouvelles interactions remplacent les plus anciennes. Cette mémoire est utilisée par la suite pour entraîner l'agent en tirant au hasard des lots d'expérience. Le but de ceci est de limiter au maximum la corrélation entre les données (puisque les différentes transitions au sein d'un même épisode sont très fortement corrélées) afin d'augmenter la capacité de généralisation de l'agent.

2.2.2 Premier réseau de neurones

Ce modèle est supposé être simple à apprendre. Le modèle que nous avons utilisé pour apprendre la fonction Q est un simple réseau de neurones fully-connected à une couche cachée. Les états étant de dimension 4, l'entrée est de dimension 4.

On passe ce vecteur dans une couche fully-connected à 256 unités. On applique la non-linéarité ReLU à chaque unité, puis on passe ce vecteur à 256 dimensions dans une couche fully-connected avec 2 unités en sortie : un nombre pour chaque action, avec une activation linéaire.

Par ailleurs, notre agent doit adopter une stratégie ϵ-greedy afin de visiter un plus grand nombre d'états et d'avoir une partie exploratoire lors de son apprentissage. Cela lui permet de découvrir de nouveaux états en effectuant des actions aléatoires avec une certaine probabilité . Nous réduisons ainsi la valeur de ϵ afin d'accélérer l'apprentissage au début et de faire en sorte que l'agent s'appuie ensuite sur ses connaissances de l'environnement pour apprendre les bonnes actions dans un état donné en calculant l'action de plus grande Q-valeur. Nous fixons donc un `eps_start`, `eps_end` et un `eps_decay` tels que le seuil pour prendre une action aléatoire ou conforme à la politique de l'agent est :

$$\epsilon = \epsilon_{end} + (\epsilon_{start} - \epsilon_{end}) \exp^{-\frac{t}{\epsilon_{decay}}}$$

avec t le nombre d'actions prises depuis le début de l'apprentissage.

2.2.3 Ajout d'un target network

Nous observons un apprentissage de notre réseau, mais celui-ci demeure très instable, puisque la fonction de coût n'est pas stationnaire. Lorsque le réseau apprend, la prédiction est modifiée, mais la cible aussi. Ainsi, nous ajoutons un second réseau (target network) partageant la même architecture que le premier mais pas les poids.

Il existe deux méthodes de mise à jour de ce réseau :
— La première consiste à mettre à jour les poids du target net tous les N épisodes d'apprentissage

— La seconde consiste à mettre à jour les poids du target net à tous les épisodes en utilisant la formule $w' = (1 - \alpha)w + \alpha w'$, avec w' les poids du target net et w les poids du policy net (*soft update*)

Nous avons donc testé les deux méthodes et avons constaté que la méthode de *soft update* semblait plus stable puisque la cible ne change pas radicalement au bout de N épisodes, ce qui peut amener des comportements très fluctuants. Nous avons dans la suite choisi un coefficient $\alpha = 0,01$.

2.3 Résultats

Après avoir itéré à de nombreuses reprises sur les hyperparamètres de notre agent, nous sommes arrivés à un très bon résultat, avec un score moyen de 495,15 après 360 épisodes, comme le montre la figure 4 où la courbe bleue montre la performance de chaque épisode et la courbe orange représente le score moyen sur les 100 derniers épisodes. Nous pouvons donc considérer le problème résolu (du moins pour la seed choisie dans notre cas pour les modules random, torch et gym, égale à 42).

Les hyperparamètres sont les suivants :

Parameter	**Value**
replay memory size	10000
batch size	128
γ	0,999
ϵ_{decay}	350
ϵ_{start}	0,9
ϵ_{end}	0,05
target soft update	0,05
number of episodes	360
optimizer	RMSprop
learning rate	10^{-2}
loss function	SmoothL1Loss (Huber loss)

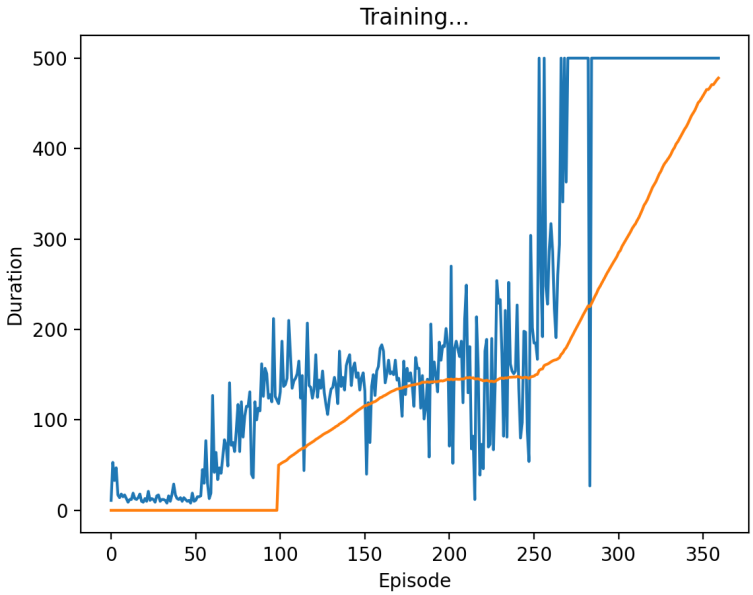

FIGURE 4 – Performances de l'agent après 360 épisodes. La courbe bleue représente la performance de l'agent à chaque épisode, la courbe orange représente le score moyen sur les 100 derniers épisodes.

3 MineRL

Une fois le code pour le Deep Q-Learning prêt, nous avons adapté ce code à un autre environnement. Le package Python `mineRL` permet l'apprentissage de modèles dans le jeu Minecraft.

3.1 Environnement Mineline-v0

L'environnement Mineline est un environnement où l'agent doit casser un bloc de diamant situé sur sa droite ou sur sa gauche. Il a la possibilité de se déplacer sur la droite, sur la gauche, ou d'attaquer le bloc.

Nous n'avons jamais pu afficher cet environnement précis, celui-ci retourne une erreur interne à MineRL.

3.2 Frame-stacking

Dans ces environnements issus de Minecraft, les états ne sont pas des features (comme la quantité de vie, la quantité de bois dans l'inventaire, ...) mais directement les frames du jeu. Ainsi, l'apprentissage effectué sur Cartpole-v1 précédemment est directement transposable ici, à condition de remplacer le réseau fully-connected par un réseau récurrent.

Toutefois, une image du jeu ne permet pas de définir uniquement l'état du joueur. On perd par exemple les notions de vitesse et de trajectoire du joueur. Une première solution consiste à considérer les états non pas comme une simple frame du jeu, mais comme une concaténation de la frame courante du jeu, et d'un certain nombre de frames passées. Par exemple, si l'on considère qu'un état est une concaténation de 4 frames, le premier état est $[f_1, f_2, f_3, f_4]$, le second est $[f_2, f_3, f_4, f_5]$, et ainsi de suite.

3.3 Modèle utilisé

Pour ce problème, nous avons utilisé un simple réseau à convolution, avec des activations ReLU et de la batch normalisation. Le réseau comporte à la fin une couche fully-connected. Le modèle complet est le suivant :

- Couche de convolution, noyau de convolution de taille 5×5, stride de 2, 4 channels en entrée, 16 en sortie. Image intermédiaire de taille (batch_size, 16, 40, 40).
- Batch normalisation à 2 dimensions, 16 channels.
- Activation ReLU.
- Couche de convolution, noyau de convolution de taille 5×5, stride de 2, 16 channels en entrée, 32 en sortie. Image intermédiaire de taille (batch_size, 32, 18, 18).
- Batch normalisation à 2 dimensions, 32 channels.
- Activation ReLU.
- Couche de convolution, noyau de convolution de taille 5×5, stride de 2, 32 channels en entrée, 64 en sortie. Image intermédiaire de taille (batch_size, 64, 7, 7).
- Batch normalisation à 2 dimensions, 64 channels.
- Activation ReLU.
- Layer Flatten qui convertit le tenseur intermédiaire de la dimension (batch_size, 64, 7, 7) à (batch_size, 3136)
- Couche fully-connected, 3136 features en entrée, 3 en sortie. Dimension de sortie (batch_size, 3).

3.4 Résultats obtenus

L'installation du package a été difficile (possible pour un seul des membres du binôme). Cela nous a laissé relativement peu de temps pour entraîner les modèles.‘

Nous pouvions entraîner sans problème les modèles, mais il a été impossible de visualiser les épisodes : la méthode `env.render` échoue avec mon installation de `MineRL` sur l'environnement `Mineline-v0`.

Nous avons tenté de nombreuses combinaisons d'hyperparamètres, en particulier les hyperparamètres liés à ε. Nous avons constaté que le frame-stacking n'était pas très utile pour ce problème simple. Toutefois, l'implémentation est fonctionnelle, mais n'a pas été utilisée dans notre meilleur hyperparamétrage, qui est le suivant :

Parameter	**Value**
replay memory size	10000
batch size	128
γ	0,99
ϵ_{decay}	300
ϵ_{start}	1
ϵ_{end}	0
target update	50
number of episodes	500
optimizer	Adam
learning rate	10^{-3}
loss function	SmoothL1Loss (Huber loss)
frame stacking	1

La courbe des récompenses en fonction du temps est la suivante :

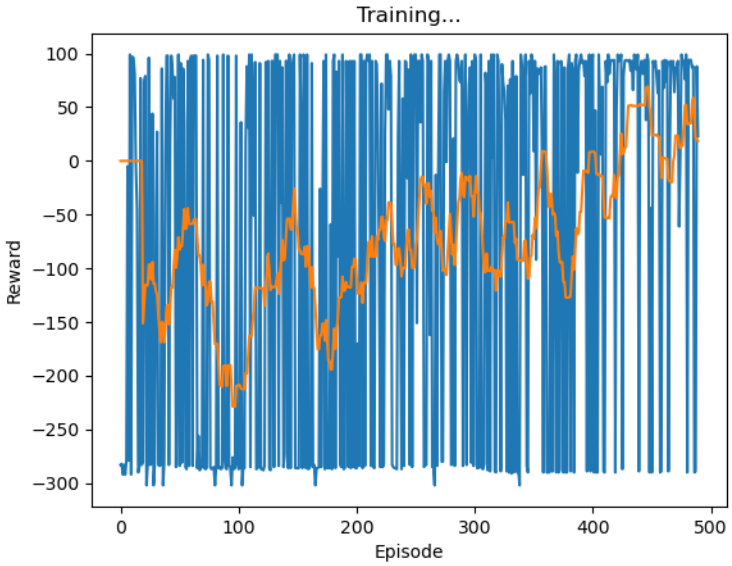

FIGURE 5 – Courbe d'apprentissage sur l'environnement Mineline-v0

On constate que les résultats sont plutôt mauvais. Ceci est dû au manque de temps pour optimiser davantage. La courbe en orange est la moyenne sur les 20 derniers épisodes. On constate que la récompense moyenne atteint les 70, ce qui montre que l'agent y arrive pratiquement systématiquement. L'allure de la courbe orange montre sans équivoque qu'il y a bien apprentissage. Pour comparaison, la récompense moyenne avec un agent aléatoire est située aux alentours de -70.

4 Bonus : Atari Breakout

4.1 Présentation de l'environnement

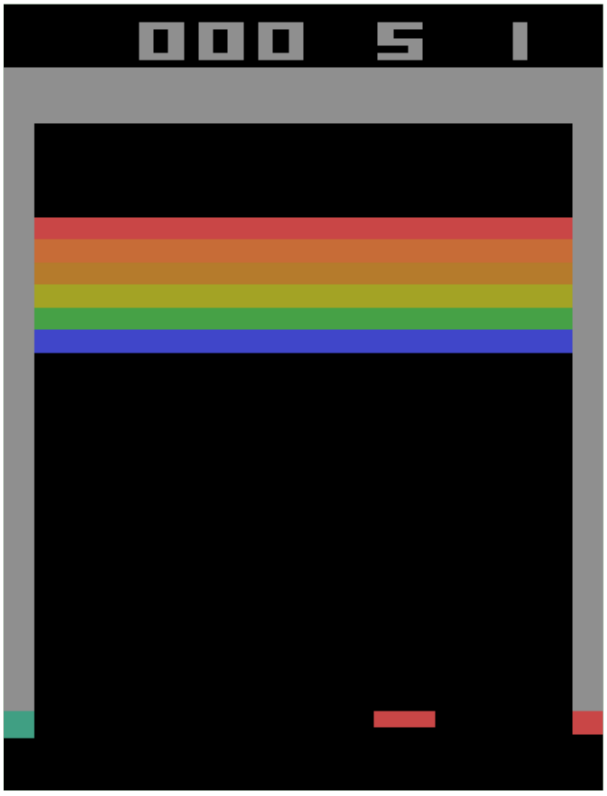

FIGURE 6 – Breakout

L'environnement ici est le jeu classique Atari appelé Breakout. C'est un jeu de casse-briques, avec des briques en haut de l'écran, une raquette similaire à Pong. Le but du jeu est d'envoyer la balle avec la raquette sur les briques, pour les casser. Si la balle n'est pas rattrapée avec la raquette lorsqu'elle tombe, le joueur perd une vie. Lorsque le joueur progresse, la balle va de plus en plus vite, rendant le jeu plus difficile.

Le jeu génère naturellement 60 frames par seconde. Ces frames sont des images RGB, de dimension (210, 160). supposons que $(s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, ...)$ soit la liste des frames. Lorsqu'une action est effectuée, elle est effectuée lors de 4 frames consécutives (ie pendant les frames s_1, s_2, s_3 et s_4). Un premier traitement effectué est de ne considérer qu'une frame sur 4. On ne garde que $(s_4, s_8, s_{12}, s_{16}, ...)$.

Pour être plus précis, on garde $(\max(s_3, s_4), \max(s_7, s_8), \max(s_{11}, s_{12}), \max(s_{15}, s_{16}), ...$, avec le max appliqué pixel par pixel.

Cette nouvelle liste de frames est ensuite convertie en noir et blanc, et la taille des frames est réduite de (210, 160) à un carré de taille 84×84. Un état est alors une succession de 4 frames consécutives de cette nouvelle collection de frames (ie ayant déjà été sous-échantillonnées par un facteur 4, avec une opération de max-pooling).

Ainsi, la dimension des états est $84 \times 84 \times 4 = 28224$ à comparer à 403 200 pour les états

non prétraités. Ces images seront passées en entrée d'un CNN à 2 dimensions, sous la forme (taille_batch, 4, 84, 84). L'empilement de frames est traité comme un channel. Un traitement de type max_pooling est effectué sur deux frames consécutives, mais nous avons trouvé relativement peu d'informations à ce sujet.

L'intérêt d'avoir comme états les 4 frames consécutives est de conserver assez d'information sur l'accélération de la balle et de la raquette. Une frame permettrait seulement la mesure de la position, deux frames la vitesse, il faut au moins 3 frames pour l'accélération. Comme une opération de pooling a été effectuée sur deux frames consécutives, il en faut une 4ème pour mesurer l'accélération.

Les actions possibles sont les suivantes :
— No-op : ne pas bouger la raquette.
— Fire : action qui sert uniquement à chaque fois que l'on perd une vie, pour lancer la balle. Cette action pose beaucoup de problèmes pour l'apprentissage.
— Right : déplacer la raquette à droite.
— Left : déplacer la raquette à gauche.

Lors d'une partie, l'agent possède 5 vies qui sont perdues quand la balle n'est pas rattrapée par la raquette. Autrement qu'avec le compteur et la détection du chiffre, ce nombre de vies n'est pas accessible facilement à l'agent lors de l'apprentissage. Ce fait entraîne de grandes difficultés à l'apprentissage, notamment à cause du fait qu'une action (Fire) n'est effectuée que après une perte de vie, entraînant sinon une partie statique car l'agent ne lance plus jamais la balle.

4.2 Modèle utilisé

Nos états étant des empilement de 4 images, il est naturel d'utiliser des réseaux à convolution pour ce problème. Nous avons utilisé un réseau alternant couches de convolution et batch normalisation.
— Couche de convolution, noyau de convolution de taille 5×5, stride de 2, 4 channels en entrée, 16 en sortie. Image intermédiaire de taille (batch_size, 16, 40, 40).
— Batch normalisation à 2 dimensions, 16 channels.
— Activation ReLU.
— Couche de convolution, noyau de convolution de taille 5×5, stride de 2, 16 channels en entrée, 32 en sortie. Image intermédiaire de taille (batch_size, 32, 18, 18).
— Batch normalisation à 2 dimensions, 32 channels.
— Activation ReLU.
— Couche de convolution, noyau de convolution de taille 5×5, stride de 2, 32 channels en entrée, 64 en sortie. Image intermédiaire de taille (batch_size, 64, 7, 7).
— Batch normalisation à 2 dimensions, 64 channels.
— Activation ReLU.
— Layer Flatten qui convertit le tenseur intermédiaire de la dimension (batch_size, 64, 7, 7) à (batch_size, 3136)
— Couche fully-connected, 3136 features en entrée, 4 en sortie. Dimension de sortie (batch_size, 4).

4.3 Résultats

Malheureusement dans le cas du breakout, nous n'avons pas réussi à déterminer les paramètres optimaux pour la résolution du problème. Nous avons choisi d'entraîner le modèle durant 50000 épisodes (parties avec chacune 5 vies, ce qui représente environ 2 jours sur les GPU de l'école

Centrale) avec une mise à jour du target network tous les 30 épisodes, et un batch size de 128. Les autres hyperparamètres sont les mêmes que pour le cartpole.

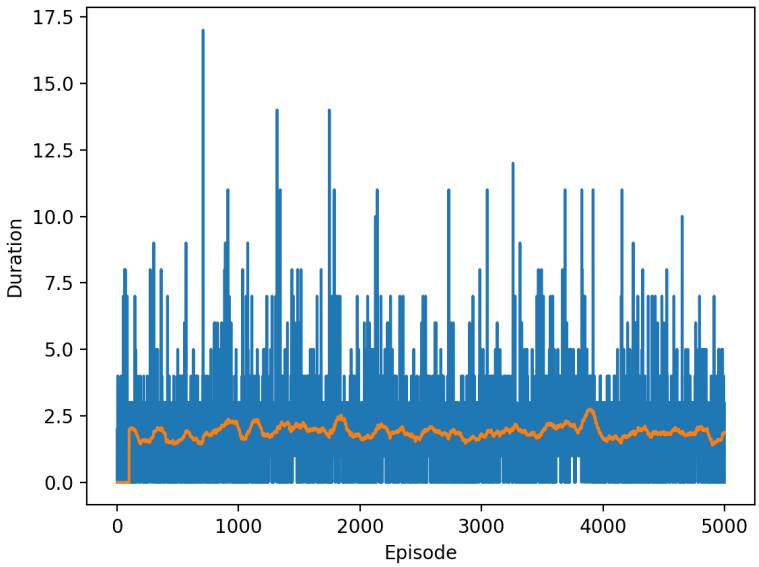

FIGURE 7 – Courbe d'apprentissage de notre modèle de breakout

Cette courbe montre bien la difficulté que nous avons à entraîner un tel modèle, même sur un très grand nombre d'épisodes. En effet, l'agent a atteint un pic de performance à 16 points mais la performance moyenne ne vaut que 2, ce qui signifie que l'agent ne fait rebondir la balle que 2 fois avant de perdre (sur 5 vies).

La prestation de l'agent lors du test est d'ailleurs assez mauvaise puisque la plateforme a tendance à se placer à gauche et à ne rien faire de la partie, ou encore à se placer aléatoirement sur l'aire de déplacement.

Plusieurs problèmes se posent : le premier est que vu le nombre d'itérations nécessaires pour que le modèle converge vers des résultats satisfaisants, nous sommes assez limités dans l'exploration de l'espace des hyperparamètres. Ensuite, le fait que la récompense ne soit pas négative mais nulle lorsque l'agent perd la balle serait peut-être à modifier. Enfin, le nombre d'états possibles rencontrés par le réseau rend très difficile l'apprentissage (par exemple deux frames identiques avec une brique en moins sur l'une des deux est perçue comme différente par le réseau convolutif). Nous pourrions, pour accélérer l'entraînement à la tâche "empêcher la balle de tomber", couper l'image au niveau des briques pour ne garder que la balle et la plateforme.

Conclusion

Ce TP nous a permis d'implémenter l'algorithme DQN sur des exemples ludiques et intéressants, de complexité croissante. Nous avons pu constater que l'apprentissage par renforcement est encore plus difficile à implémenter qu'un apprentissage supervisé classique. L'une des limitations de l'apprentissage par renforcement est que le problème sous-jacent d'apprentissage supervisé est un problème où la cible n'est pas exacte mais est une valeur estimée. La dimension de l'espace d'actions est usuellement très grande, et cela nécessite beaucoup d'épisodes afin d'avoir une

bonne couverture de l'ensemble des situations possibles. Ces épisodes nécessitent de simuler l'environnement, ce qui requiert une puissance de calcul bien plus grande qu'un simple apprentissage supervisé.

Nous aurions pu aller plus loin en implémentant d'autres algorithmes d'apprentissage supervisé, par exemple le Dual DQN. Nous aurions également pu implémenter un Prioritized Experience Replay pour sampler les transitions ayant entraîné les plus grandes erreurs de prédiction, et ainsi apprendre plus vite. Enfin, nous aurions pu, avec plus de temps, expérimenter plus profondément l'espace des hyper-paramètres et les différents optimiseurs et fonctions d'erreurs afin d'obtenir de meilleures performances pour nos agents.