APPLICATION DU DOUBLE DEEP Q-LEARNING SUR SUPER MARIO BROS

A PREPRINT

Loïc Bouchery
loic.bouchery.1@ulaval.ca

Anh My Buis
anh-my.buis.1@ulaval.ca

December 21, 2022

ABSTRACT

L'apprentissage par renforcement est l'une des branches les plus étudiées dans le domaine de l'Intelligence Artificielle (IA) par sa grande capacité de prise de décision autonome face à différents types environnements. Elle joue un rôle majeur dans l'apprentissage de séquences d'actions par des agents autonomes, dont on peut tester l'efficacité dans des jeux vidéo, à travers des librairies de références comme la banque atari ou gym. Comme projet d'apprentissage par renforcement, nous avons ainsi décidé d'appliquer une stratégie de type Double Deep Q-learning sur un environnement dynamique, le premier niveau de Super Mario Bros de la SNES. L'objectif avec cet algorithme est de l'entrainer à battre le premier niveau de ce jeu le plus rapidement que possible, et ainsi de montrer l'efficacité d'un tel agent sur une tâche séquentielle complexe.

Keywords Double Deep Q-Learning · Apprentissage par Renforcement · Super Mario Bros

1 Environnement d'apprentissage

1.1 Implémentation de l'environnement

Afin d'implémenter l'algorithme Double Deep Q-Learning, nous avons besoin de créer notre environnement mentionné ci-dessus. Pour cela nous utilisons "gym super mario bros" une librairie fournie par OpenAI gym qui a une grande collection d'environnements test pour l'apprentissage par renforcement.

La taille de notre environnement Super Mario Bros est de (3 x 240 x 256), où (240 x 256) correspond à la dimension de la fenêtre contenant l'écran du jeu, et 3 aux les trois canaux de couleurs Rouge Vert Bleu. Ce sont des éléments cruciaux pour le fonctionnement de l'apprentissage automatique permettant à l'agent, Mario, de reconnaitre sa position, des obstacles, divers objets et ennemis mouvants. Cette étape est importante car cela permet à l'agent de comprendre qu'il doit éviter les obstacles et les ennemis afin d'atteindre son objectif sans mourir.

Ainsi, un prétraitement de l'environnement est effectué pour alléger les données. Tout d'abord, l'image a été convertie en niveaux de gris, ce qui donne en taille (1 x 240 x 256), seulement la reconnaissance des objets est nécessaire. Ensuite, la fenêtre a été redimensionnée en une dimension plus petite (84x84) pour faciliter le traitement des images.

En plus de cela, afin de préserver la propriété Markovienne, il est essentiel de récupérer plusieurs trames avant afin d'avoir les actions produites auparavant. Afin d'éviter de perdre d'informations, certaines trames intermédiaires sont ignorées durant la partie. En effet, comme les trames consécutives peuvent être similaires, les sauts de trames réguliers sont utilisés. Dans notre cas, on récupère 4 trames en niveaux de gris de taille (84x84) qui seront transférés dans notre réseau de neurone en tant que tenseurs.

Il doit aussi prendre en compte le temps affiché en haut à droite de l'écran.

Figure 1: Ecran du jeu de dimension (240x256)

1.2 Espace d'actions

L'espace d'actions dans Super Mario Bros correspond à tous les mouvements possibles de l'agent Mario. Il peut attendre, courir ou sauter à droite ou à gauche. L'espace d'actions défini avec OpenAI contient 256 actions. Mais il est possible de gagner du temps d'entrainement en limitant l'agent à quelques actions. Dans notre cas, notre agent est limité à allez seulement vers la droite ce qui réduit considérablement l'espace d'actions à 5 actions : Ne pas bouger, aller à droite, aller à droite et sauter, aller à droite et courir, aller à droite sauter et courir.

1.3 Fonction de récompense

Grâce au prétraitement défini ci-dessus, il est possible de définir la fonction de récompense. Elle suppose que l'objectif est de se déplacer vers la droite le plus rapidement que possible sans mourir.

Elle est déjà prédéfinie par la librairie d'OpenAI et est composée de 3 facteurs différents:
- la vélocité, prenant en compte la position de Mario avant et après le pas de temps, v = x1 - x0. Donc s'il se déplace vers la droite, v > 0 s'il ne bouge pas v = 0.
- le temps, affiché en haut à droite avant et après le pas de temps. il est important car si le temps est écoulé, c'est une fin de partie. C'est donc une pénalité qui l'empêche de rester immobile, c = c1 - c0
- le death penalty. On applique directement une pénalité à Mario s'il meurt quelques soit la cause, en touchant un ennemi, en tombant dans un trou ou par manque de temps. d = 0 s'il est en vie, sinon d = -15.

La récompense est alors calculée par la somme de ces 3 facteurs v + c + d.

2 Apprentissage

On a ici affaire à un espace d'action de très grande taille, puisqu'il contient les informations utiles aux déplacements dont la position de mario dans le stage. C'est pourquoi les approches tabulaires sont à proscrire ici : on utilise donc une stratégie de type Deep RL basée sur le Q-learning.

Q-learning est une stratégie off-policy de type Temporal Difference typique en apprentissage par renforcement. Elle consiste à estimer la fonction de valeur d'action Q à partir des connaissances de l'agent. Cette fonction est parametrée par des poids θ, que l'on met à jour selon les rewards observées lors des transitions d'états afin de pousser l'agent dans la direction à suivre.

2.1 Deep Q Learning

L'approche Deep Q-Learning emploie un réseau de neurones afin d'estimer la valeur de Q pour choisir quelle action a est à effectuer dans un état donné s.

Selon ces choix, l'agent observe puis stocke les transitions observées dans un buffer interne nommé *Replay buffer*. A partir de ces observations, on effectue une phase dite d'*Experience replay*.

2.1.1 Experience Replay

Dans notre implémentation, on utilise en tant que replay buffer des tenseurs de taille fixée et de type first in first out (FIFO). On remplace au besoin les anciennes valeurs par les nouvelles observations.

Dans cette phase, on échantillonne aléatoirement une transition dans le replay buffer et on met à jour le paramétrage θ du réseau en estimant les valeurs cibles :

$$Q^*(s_t, a_t) \approx r_t + \gamma \max_a Q_\theta(s_{t+1}, a)$$

On effectue ensuite la descente du gradient selon :

$$\Delta\theta = -\alpha(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))\nabla Q_\theta(s_t, a_t)$$

2.1.2 Politique epsilon-t greedy

La stratégie Q-Learning fait partie des approches Value based, c'est-à-dire que l'on utilise des fonctions de valeur afin de maximiser les rewards selon une politique implicite donnée.

L'approche décrite précedemment définit une politique exploitative selon la fonction de valeur Q. Pour ce qui est de l'exploration on doit donc forcer l'agent à chercher de nouvelles actions selon la politique epsilon-t greedy :

Avec une probabilité ϵ_t, on explore de nouvelles actions sélectionnées aléatoirement. A chaque épisode, on réduit ce taux d'exploration d'un facteur donné, jusqu'à un minimum que l'on spécifie à l'agent.

2.2 Double Deep Q Learning

Le rapport de [1] montre que pour des environnements complexes comme ceux de la librairie Atari, l'approximation des valeurs de Q est trop optimiste dans ses prédictions et pousse l'agent à effectuer des séquences d'actions qu'il estime pouvoir mener à l'état final, bien qu'elles soient en réalité sous optimales.

En effet, le principal problème rencontré avec l'approche précédente est la non stationarité des valeurs cibles. Pour y remédier, on utilise un deuxième réseau afin de prédire les valeurs cibles pour la mise à jour du réseau.

On a alors un réseau primaire parametré par θ et un réseau cible parametré par θ', qui sont identiques au départ. Afin d'alléger les notations, on va noter ces réseaux Q_P et Q_T dans la suite du rapport. On effectue toujours les prédictions dans la politique ϵ_t greedy à l'aide du réseau primaire.

Lors de la mise à jour des poids, on prédit les valeurs cibles avec le réseau cible et on met à jour les poids à partir de cette prédiction, selon un taux d'apprentissage α donné.

Enfin, tous les n pas de temps, on effectue la mise à jour des valeurs cibles. On a plusieurs approches envisageables : le fixed targets, qui consiste à copier les poids $\theta' \leftarrow \theta$ afin de simuler la stationarité des valeurs cibles ou le moyennage mobile selon un paramètre τ tel que $\theta' \leftarrow (1-\tau)\theta' + \tau\theta$.

En partant de ces principes, on implémente l'algorithme décrit ci-dessous :

Algorithm 1 Double Deep Q-Learning pour l'apprentissage d'une politique π

function AGENT DDQN
Initialiser les réseaux Q_P et Q_T, le replay buffer $\mathcal{D}$ et la fonction de perte $\mathcal{L}$
Répéter pour chaque épisode
Répéter pour chaque pas de temps $t \geq 1$ jusqu'à l'état terminal
Choisir a_t en fonction de s_t en utilisant π basé sur Q
Appliquer a_t et observer s_{t+1} et r_{t+1}
Stocker (s_t, a_t, r_t, s_{t+1}) dans le buffer $\mathcal{D}$
Répéter pour chaque étape de mise à jour
Echantilloner $s_t, a_t, r_t, s_{t+1} \sim \mathcal{D}$
Choisir l'action a' selon le réseau cible $a' = \arg\max_a Q_T(s_t + 1, a)$
Calculer les valeurs cibles $Q^*(s_t, a_t) \approx r_t + \gamma Q_P(s_{t+1}, a')$
Effectuer une descente du gradient pour Q_P selon $\mathcal{L}(Q^*(s_t, a_t), Q_P(s_t, a_t))$
Effectuer la mise à jour du réseau cible selon le critère choisi

3 Entraînement et résultats

3.1 Choix des hyperparamètres

En terme d'exploration, on veut que l'agent puisse analyser un maximum d'action au début afin d'avoir une intuition sur les séquences maximisant les estimations de Q. On utilise pour cela $\epsilon_0 = 1$, que l'on diminue d'un facteur de 0.99 à chaque épisode, jusqu'a $\epsilon_{min} = 0.1$.

On choisit un optimizer Adam avec une learning rate assez faible afin de garantir la stabilité de l'apprentissage, de l'ordre de 10^{-4}.

Plus l'on augmente la taille du replay buffer, plus les données vues par l'agent sont variées et les étapes d'experience replay sont significatives, mais plus l'agent va demander en ressources computationelles. On va donc se limiter à une taille de buffer de 20000 afin de pouvoir l'entrainer en un temps raisonnable.

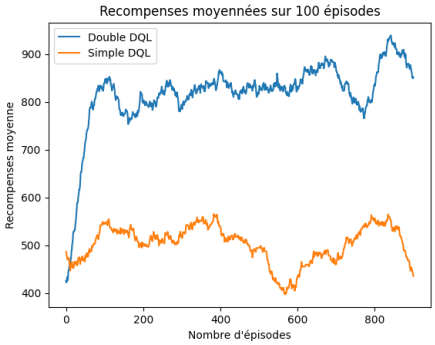

Figure 2: Performances des agents DQN et DDQN

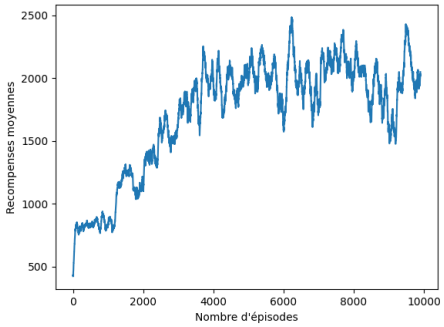

Figure 3: Entrainement de l'agent DDQN

3.2 Performances et démarche de l'agent

Comme le montre la figure 2, l'agent utilisant la stratégie DDQN selon fixed targets est bien plus performant que l'agent DQN simple.

On choisit donc de continuer l'entrainement avec l'agent DDQN, dont les résultats moyennés sur 100 épisodes sont illustrés figure 3. Lors des premiers épisodes, l'agent commence à apprendre comment faire face à l'ennemi présent au début du stage, mais a des difficultés à franchir les obstacles. Lorsqu'il franchit un nouvel obstacle, l'agent doit à nouveau explorer afin de déterminer comment le passer de manière plus optimale, et comment faire face aux obstacles suivants.

Après les avoir franchi aux alentours de 1000 épisodes, l'agent fait face à un obstacle alors inconnu : les trous. Ce nouvel obstacle va lui nécessiter une grande exploration puisque l'agent doit d'abord pouvoir arriver jusqu'au trou sans manquer de temps, puis explorer les éventualités depuis ce point.

En pratique, l'agent arrive dans de très rares cas à partir de 5000 épisodes à finir le niveau. Aux alentours de 6400 épisodes, on obtient le plus haut pic de récompenses et l'agent est capable de finir l'instance dans plus de 20% des cas en entrainement. Si l'on passe son taux d'exploration à 0 et que l'on passe à une stratégie purement exploitative en phase de test, on observe que l'agent est en effet capable de résoudre l'environnement.

Au dela, on observe que l'agent redeviens moins efficace en termes de récompenses pures, mais si on le déploie à nouveau en test, on remarque qu'il est bien plus rapide qu'avant. Il nécessite en effet moitié moins de pas de temps qu'a 3000 épisodes pour atteindre la moitié du stage, jusqu'a atteindre à nouveau la completion du stage vers 9000 épisodes.

3.3 Analyse de la démarche de l'agent

A chaque nouvel obstacle franchi, l'agent doit revenir au point où il en était à partir de sa politique exploitative puis revenir à une phase plus exploratoire afin de pouvoir avancer dans le niveau. Cela incombe des durées d'entrainement de plus en plus longues, puisque l'agent arrive de plus en plus loin au fil des épisodes.

Dans le niveau exploré, l'agent est souvent ralenti par cette phase exploratoire, lorsqu'il doit trouver des séquences d'actions précises, par exemple afin de traverser les tuyaux, qui nécessitent de maintenir l'action de saut pendant X frames selon la taille, puis de se déplacer à droite au bon moment afin de pouvoir franchir cet obstacle, sans quoi il devra recommencer cette séquence depuis le début sans avoir d'information directe de la part de l'environnement sur la marche à suivre.

En effet, lorsqu'il meurt face à un ennemi ou un trou, l'agent subit la pénalité de mort et devient à partir de ces informations capable d'esquiver ces obstacles assez vite. La pénalité de mort lorsque l'agent n'arrive pas à franchir un tuyau est liée au temps : cette pénalité est la moins informative de toute, car l'agent doit alors revenir à cet endroit, puis essayer des séquences de manière aléatoire jusqu'à trouver *l'unique séquence* capable de lui faire passer l'obstacle.

3.4 Généralisation

Pour tester les performances de l'agent en généralisation, on le déploie sur le stage 1-2. Comme on pourrait s'y attendre, cela n'offre pas de bons résultats avec l'agent pré-entrainé : il faudrait alors ré-entrainer l'agent sur le nouveau stage.

Il est logique que l'agent ait du mal à obtenir de bonnes performances sur un nouvel environnement avec une politique exploitative, puisqu'un nouvel état observé ne donne aucune information présumée sur le comportement des ennemis qui peut différer de celui des autres stages. Seule l'exploration de ce nouveau stage est capable de donner à l'agent ces informations.

On peut cependant envisager des approches de transfer learning en copiant les poids des réseaux tout en augmentant le taux d'exploration initial pour cette nouvelle tâche, ce qui revient à utiliser un réseau pré-entrainé.

References

[1] Hado van Hasselt, Arthur Guez, and David Silver. Deep Reinforcement Learning with Double Q-learning, 2015.

[2] Christian Kauten. Super Mario Bros for OpenAI Gym. GitHub, 2018.

[3] Andrew Grebenisan. Play Super Mario Bros with a Double Deep Q-Network, 2020.

[4] Howard Wang Steven Guo Yuansong Feng, Suraj Subramanian. Train a Mario-playing RL Agent, 2020.