
# Rapport du Projet Reinforcement Learning

**AI agent on Atari Breakout**

---

## Petignat Alexis, Promo 2026 SCIA

Enseignant : Pierre-Louis GUHUR

# 1 Introduction

Ce rapport détaillera l'implémentation de mon projet de Reinforcement Learning sur le jeu Breakout d'Atari, via la librairie python *gym*.

Pour rappel, ce projet consistait en l'entraînement non supervisé d'un agent d'Intelligence Artificielle sur le jeu Breakout d'Atari, afin d'obtenir un agent autonome capable de jouer au jeu et d'obtenir le meilleur score possible. L'implémentation globale de cet agent devait suivre l'implémentation classique d'un agent de Deep Q-Learning.

# 2 Implémentation

## 2.1 L'agent

Le début de l'implémentation reprend le code de l'agent de Q-Learning du TP précédent, bien-sûr modifié au besoin pour le cas d'usage. Il est évident qu'un simple agent de Q-Learning n'est en rien suffisant pour une tâche complexe comme le jeu de breakout, à commencer par une raison simple et suffisante : la valeur d'un état de jeu n'est pas triviale à calculer. En effet, là où d'autres jeux ont des possibilités dénombrables (jeu du Morpion, du taxi), le jeu de Breakout (ou casse-brique) possède un nombre quasi-indénombrable d'états, et même si ces états étaient dénombrables, il est difficile de calculer la récompense associée à un état S. C'est pourquoi la première étape nécessaire à l'implémentation de l'agent a été de créer un réseau de neurones convolutionnel chargé d'estimer la valeur associée à un état de jeu.

### 2.1.1 Le modèle

Ce modèle reprend les caractéristiques exactes du modèle utilisé par Google pour le même objectif, à savoir un réseau (détaillé en annexe) prenant en entrée un tenseur de taille 4x84x84 et donnant en sortie un tenseur 1xN, avec N le nombre d'actions à la disposition du joueur. Le tenseur d'entrée représente non pas les canaux de couleurs standards (r, g, b, a) mais une succession de 4 images du jeu, donnant ainsi une notion de mouvement cruciale à l'évaluation de l'état et des potentielles récompenses futures. La dimension 84x84 sans couleur implique une étape de preprocessing de chaque frame par par une mise en niveau de gris obtenue comme étant la moyenne sur la dernière dimension du tenseur, puis en la redimensionnant à l'aide du Pipeline de transformation de tenseurs fournie par Torch. Le résultat est stocké à chaque étape dans un buffer de l'agent et fait office d'état.

#### 2.1.2 Variations d'Epsilon

Epsilon est une variable associée à l'agent, comprise entre 0 et 1, qui définit la probabilité que l'agent ne choisisse une action aléatoire plutôt que la meilleure action recommandée par le modèle. L'implémentation d'un agent de Q-Learning Standard fixe cette valeur d'epsilon. Ici, le choix d'implémentation s'est porté sur un agent de Q-Learning avec Epsilon Decay. C'est à dire que la valeur d'epsilon n'est pas fixée, mais décroît au fur et à mesure des parties jouées. Sur le modèle final qui a été entraîné, Epsilon était fixé à 1 à l'initialisation, puis baissait sur 10 000 epochs pour atteindre 0,05, soit un taux de choix aléatoire de 5%. Cet epsilon decay a été couplé à une autre forme de variation que nous détaillerons ci-dessous.

#### 2.1.3 Stabilisation du modèle

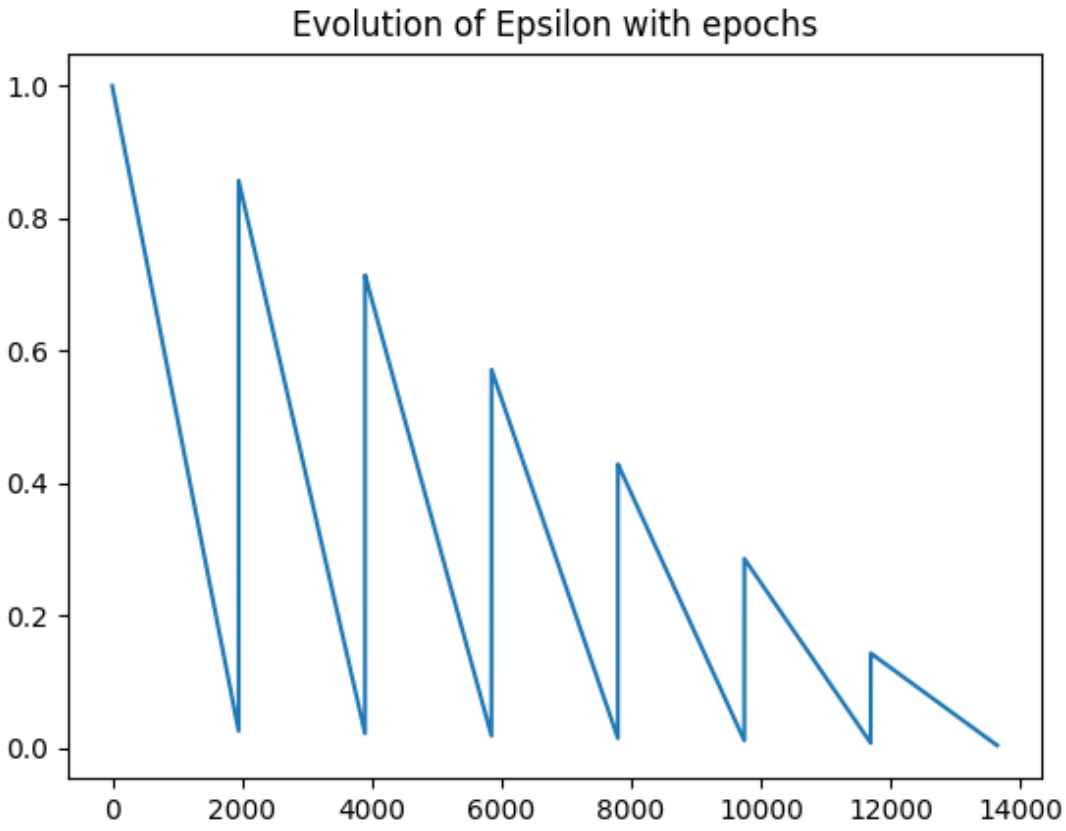
Le problème du Deep Q-Learning est la possibilité de divergence de l'agent, c'est-à-dire le fait que le réseau de neurones ne régresse pas malgré l'entraînement (puisque le modèle est entraîné sur une cible "mouvante", qui est en réalité lui-même). Ce danger a été ici abordé par la mise en place d'un réseau "cible", ou target network. Le rôle de ce réseau est de rendre statique la fameuse "cible" recherchée par le modèle. On définit ce modèle comme une copie du modèle entraîné, qui n'est pas mise à jour à chaque entraînement, seulement après un nombre fixé d'époques. Cela présente l'avantage d'une convergence des résultats vers une cible donnée et fixe (à court terme), permettant au modèle de comprendre profondément l'input et d'avoir un semblant de cohérence tout au long de l'entraînement.

## 2.2 Processus d'entraînement de l'agent

Le processus de l'agent peut à priori sembler simple, mais il est nécessaire de présenter plusieurs choix d'implémentation qui ont été faits tout au long du projets.

### 2.2.1 Variations d'Epsilon au cours de l'entraînement

Pour permettre au modèle d'être à même d'explorer suffisamment tout au long de l'entraînement, Epsilon est épisodiquement remis à une valeur importante lors de l'entraînement selon le graphe ci-dessous.




Ceci est une représentation approximative d'epsilon. Cette technique d'entraînement n'est en aucun cas une technique classique de Deep Q-Learning mais davantage une expérimentation de ma part, qui s'est avérée être fonctionnelle. Notez qu'une fois ces variations passées, Epsilon est fixé à 0.05 jusqu'à la fin de l'entraînement.

#### 2.2.2 Période d'entraînement

Entraîner le modèle est coûteux, c'est pour quoi nous avons ajouté un hyperparamètre **FRAME_BETWEEN_TRAIN** qui désigne le nombre de parties jouées avant de ré-entraîner le modèle de l'agent.

#### 2.2.3 Période de mise à jour du modèle cible

Un autre hyperparamètre a été introduit : **TARGET_UPDATE_INTERVAL**. Ce paramètre désigne le nombre de frames au bout duquel le modèle "cible" est remis à jour sur la version actuelle du modèle.

#### 2.2.4 Choix des hyperparamètres du modèle et de l'agent

Plusieurs hyperparamètres ont été choisis pour l'entraînement de l'agent :

- Le Leaning Rate du modèle est fixé à 0.00025 (optimiseur Adam)
- Le Epsilon de départ est fixé à 1
- Le Epsilon minimum est fixé à 0.05
- Gamma est fixé à 0.99 (importance des récompenses futures comparées à la récompense actuelle)
- **TARGET_UPDATE_INTERVAL** : Ce paramètre a été fixé à 10 000. C'est à dire que le modèle cible est remis à jour toute les 10 000 frames (et non pas parties).
- **FRAME_BETWEEN_TRAIN** : Ce paramètre a été fixé à 3. C'est à dire que l'entraînement ne se fait pas à chaque partie mais toute les 3 parties pour des raisons d'optimisation.
- N, la taille du buffer d'entraînement du modèle est fixée à 50 000 états en mémoire. On stocke donc 200 000 frames dans cette mémoire au total, puisque chaque état est composé de 4 frames.
- La loss utilisée pendant l'entraînement est la MSE, qui calcule l'écart entre la prédiction de la valeur d'un état par le modèle et la valeur estimée par l'équation classique de l'agent de Q-Learning.

## 2.3 Autres expérimentations

Certaines expérimentations ont été réalisées lors de la phase d'implémentation mais n'ont pas portées leur fruits. Il reste néanmoins pertinent de les mentionner.

### 2.3.1 Accélération de l'entraînement

L'entraînement d'un tel agent est très chronophage. En prenant en compte toutes les phases d'expérimentation au cours de l'implémentation du TP, on peut estimer ce temps à plus de 30h dans notre cas. Ainsi, une technique pour accélérer la convergence du modèle a été implémentée puis retirée.

Ce temps de convergence exceptionnellement long est explicable par la faible valeur des gradients, puisque la plupart des états du jeu donnent un résultat de 0. Ainsi dans une écrasante majorité de cas au début de l'entraînement, le modèle n'apprendra rien de l'état puisque sa récompense est nulle. Nous avons ainsi tenté d'accélérer le processus en ajoutant plusieurs fois un état dont la récompense présente est non nulle, pour faire augmenter le gradient et accroître les chances qu'un état "intéressant" soit sélectionné pour la backpropagation (puisque la backpropagation se fait sur un sample des états stockés).

Pour être plus précis, l'état était ajouté entre 1 et 10 fois selon la valeur de la TD error, c'est à dire que plus un état est "surprenant" pour le modèle, plus il est ajoutée dans la mémoire et plus il a de chances d'être sélectionné. Cependant, le calcul de la TD error pour chaque état ralentissait beaucoup l'exécution, doublant le temps par époque. La méthode ne semblait pas non plus améliorer la vitesse de convergence, ainsi elle a été retirée mais est toujours visible dans le code, commentée cependant.

### 2.3.2 Double Q-Learning

La méthode du double Q-Learning est un changement simple qui permet en théorie d'augmenter les performances du modèle en réduisant la surestimation de la récompense liée à un état. Elle se base sur la méthode de stabilisation implémentée précédemment et consiste, durant l'entraînement, à faire en sorte que ce soit le modèle cible qui prédise la récompense liée à un état futur plutôt que le modèle utilisé. Cette technique a été implémentée puis testée sur 3000 parties. Aucun résultat notable n'a été constaté, le modèle n'apprenait ni mieux ni moins bien. La version actuelle du projet implémente cette technique, mais elle ne sera ici que mentionné, en raison de l'absence de résultats.

#### 2.3.3 Backups manuelle du modèle

Lors des entraînements, il arrivait parfois que la récompense ne baisse de 4 points en quelques centaines d'époques. Une méthode pour pallier à ce "désapprentissage" avait été tentée, elle consistait simplement à faire des backups de modèles lorsqu'il est performant, puis de recharger ce même modèle lorsqu'un désapprentissage était constaté. Cependant, cela ne changeait en rien le problème, puisque une baisse de la récompense peut être simplement un manque de chance et non pas un désapprentissage. D'autre part, une récompense en baisse n'indique pas forcément un mauvais comportement mais possiblement une transition de comportement. Ainsi, par manque de certitudes, cette méthode a été abandonnée, mais certaines backups peuvent être trouvées dans le projet.

#### 2.3.4 Tensorboard

Tensorboard est une librairie python permettant de suivre en direct les performances d'un modèle en cours d'entraînement. Il permet de visualiser l'évolution de différentes métriques comme la loss, la récompense, la récompense moyenne... Tensorboard est désormais pris en charge dans ce projet, mais il a été mis en place seulement sur la toute fin de l'entraînement, c'est la raison pour laquelle peu de données ont été recueillies sur le début de l'entraînement du modèle.

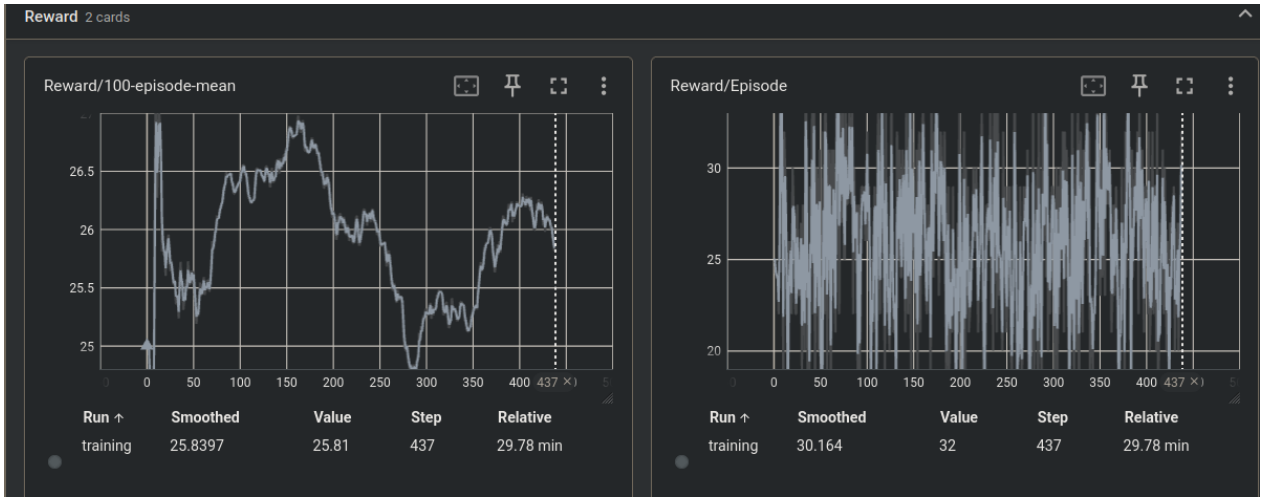
### 2.4 Limites de l'implémentation

Certaines pistes n'ont pas été explorées pendant l'implémentation de ce TP. Parmi ces pistes on peut citer :

- La quantization et pruning du modèle pour accélérer l'exécution
- Changements d'architecture du modèle par rapport à l'implémentation de Google
- Plus de temps d'entraînement

## 3 Résultats

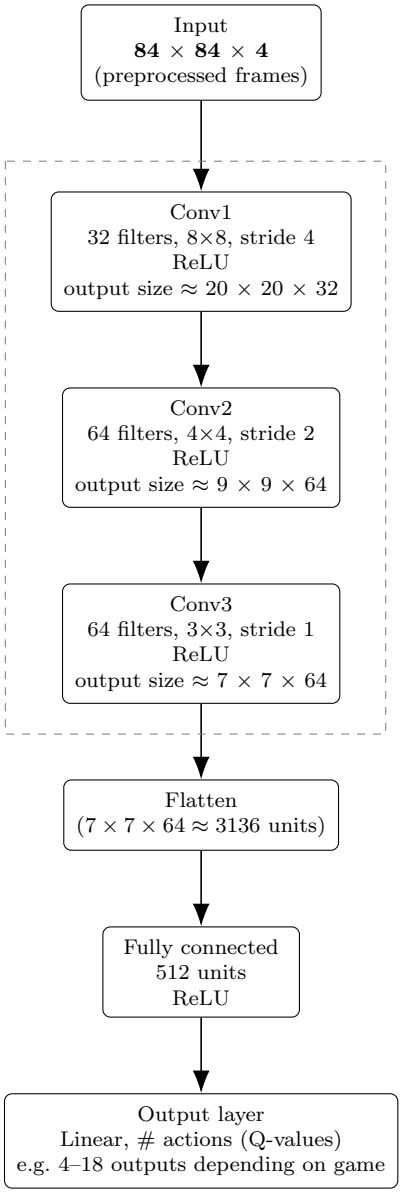
Le modèle a été entraîné sur une durée cumulée d'environ 24h pour un total estimé à plus de 14m de frames et 11000 parties. Le résultat démontre un apprentissage clair des règles et des comportements à adopter pour obtenir un bon score. La récompense moyenne se situe autour des 25 points par partie, avec un record à plus de 40 points ! Ci-dessous se trouve la courbe des récompenses sur 400 parties avec un modèle n'apprenant plus.




A gauche se trouve la moyenne sur 100 parties, tandis qu'à droite se trouve la récompenses brute à chaque étape. On constate une variation claire même en l'absence d'apprentissage, ce qui démontre une variance élevée des résultats obtenus par le modèle liée à l'aléatoire. Plus concrètement l'agent renvoie la balle la plupart du temps, mais il arrive qu'il la rate quand celle-ci va trop vite ou bien que l'agent prennent de mauvaises décisions tout simplement, notamment en restant bloque sur un des cotés de l'arène. Cependant on peut considérer que l'exercice d'implémenter un DQN fonctionnel sur le jeu d'Atari Breakout est un succès, malgré le fait que l'agent n'aie pas atteint un niveau de jeu "surhumain" comme celui de Google.

# 4 Annexe

## 4.1 Architecture du modèle utilisé pour le projet




Conv layer 1 : 32 @ 8x8, s=4

Conv layer 2 : 64 @ 4x4, s=2

Conv layer 3 : 64 @ 3x3, s=1

FC : 512 units (ReLU)