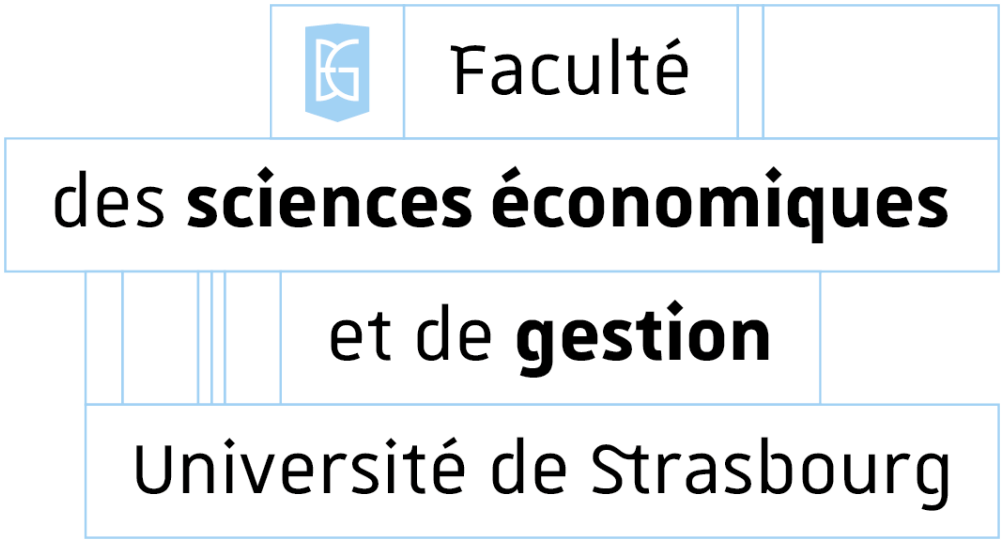
Comparaison de Méthodes d'Apprentissage par Renforcement : *Q*-Learning, DQN et DDQN dans un Environnement Pong

Auteurs: Valentin Barthel*
Adrien Busche†
Supervisé par: Bertrand Koebel

8 novembre 2024



*Magistère Génie Economique 3 et Master 2 Data science pour l'Economie et l'Entreprise, Faculté des sciences Economiques et de Gestion, Université de Strasbourg, 61 avenue de la Forêt Noire, 67000 Strasbourg, France, valentin.barthel@etu.unistra.fr, apprenti Data Scientist au Crédit Agricole Alsace-Vosges.
Github: https://github.com/valentinb67

†Magistère Génie Economique 3 et Master 2 Data science pour l'Economie et l'Entreprise, Faculté des sciences Economiques et de Gestion, Université de Strasbourg, 61 avenue de la Forêt Noire, 67000 Strasbourg, France, adrien.busche@etu.unistra.fr, apprenti Data Analyst à StromAI.
Github: https://github.com/abusche
Script Python du Projet: https://github.com/valentinb67/Pong-player-DQN-agent

Table des matières

1 Introduction

Pong est l'un des premiers jeux vidéo jamais créés, simulant une partie de tennis de table dans laquelle deux joueurs se renvoient une balle à l'aide de raquettes. Sa simplicité, à la fois en termes de mécanique de jeu et de conception graphique, en fait un terrain de jeu idéal pour expérimenter des algorithmes d'apprentissage par renforcement. En effet, Pong permet de tester des agents dans un environnement dynamique, avec des règles claires, tout en présentant des défis intéressants liés à la coordination des actions et à l'anticipation des mouvements de l'adversaire.

Pour ce projet, nous avons choisi de développer et de comparer trois agents d'apprentissage par renforcement qui tentent de maîtriser le jeu de Pong : un agent basé sur le Q-Learning (Watkins and Dayan, 1992), un deuxième sur le Deep Q-Network (DQN) (Mnih, 2013) et enfin un dernier sur le Double DQN (DDQN) (Van Hasselt, Guez and Silver, 2016). Ces trois approches, bien que fondamentalement basées sur la même logique de maximisation de récompenses, diffèrent en termes de complexité d'architecture et de capacité à apprendre dans des environnements continus aux dimensions élevées.

Dans ce rapport, nous verrons que le Q-Learning est une méthode d'apprentissage par renforcement relativement simple à mettre en œuvre et à interpréter suivant un Processus de Décision Markovien (MDP). Il fonctionne en utilisant une table Q qui stocke les valeurs d'état-action pour chaque combinaison possible dans l'environnement. Cependant, cette approche est limitée pour des espaces d'états important ou continu, comme dans le cas de Pong en haute résolution. En raison de la taille de l'espace des états, la méthode tabulaire est innadaptée pour l'apprentissage efficace du modèle. Les algorithmes DQN et DDQN sont des extensions du Q-Learning qui utilisent des réseaux de neurones pour approximer la fonction Q. Cela permet de traiter des espaces d'états continus et de grande dimension sans utiliser de table Q. Grâce à leur architecture, ces modèles sont capables d'apprendre de manière plus rapide et plus stable, en utilisant des techniques comme le replay buffer et des réseaux distincts afin de stabiliser l'apprentissage. Le DDQN, inspiré du Double Q-learning (Hasselt, 2010), corrige le biais de surestimation observé dans le DQN en séparant les réseaux utilisés pour choisir l'action et pour évaluer sa valeur, améliorant ainsi la robustesse de l'agent.

Ce rapport vise à décrire les différentes méthodes mises en œuvre, les architectures des modèles utilisés et les résultats obtenus pour chacun des trois agents en environnement discrets et continus. Chaque approche apporte une perspective différente sur la manière de faire face aux défis de l'apprentissage dans un environnement dynamique comme Pong, avec une attention particulière portée à l'interprétabilité du modèle, la complexité de l'entraînement, et la performance de l'agent à travers des métriques d'évaluation rigoureuses.

2 Notre Environnement Pong

2.1 Présentation de l'Environnement, des Actions et du Système des Récompenses

Pour ce projet, nous avons choisi de travailler sur un environnement Pong que nous avons développé à l'aide de la bibliothèque Pygame de Python. La version du jeu Pong est un environnement en deux dimensions 640×480 dans laquelle deux joueurs s'affrontent. Une balle sera lancée aléatoirement depuis le point $(0, 0)$ dans l'une des deux directions données par les vecteurs $(\vec{v}_1; \vec{v}_2) = ((-\cos 45°, \sin 45°); (-\cos 45°, -\sin 45°))$ *(cf. Annexes, Notre Environnement Pong, figure 3)*. Dans le cadre de notre entraînement, le joueur 1 (J_1), à gauche de l'environnement est un ordinateur contrôlant la raquette 1 (r_1). Il a pour unique fonction de traquer la balle sur la hauteur y, de telle sorte à ce que $y.r_1 = y.balle$, afin de renvoyer la balle au modèle que l'on souhaite entrainer afin d'automatiser l'entraînement. L'algorithme, joueur 2 (J_2) que nous souhaitons entraîner par différentes méthodes d'apprentissage (Q-learning, DQN et DDQN) évoluera sur la partie droite de l'environnement et obtiendra une récompense. L'objectif du jeu étant de ne pas laisser passer la balle. Les deux joueurs se renverront la balle en évoluant uniquement sur leur position y, pour un point x. L'agent obtiendra une récompense de "+1" lorsque sa raquette (r_2) rentre en collision avec la balle et obtiendra une récompense de "-10" lorsque la balle n'est pas réceptionnée. Ainsi, $\forall i \in J, A_j = \{\text{UP, STAY, DOWN}\}$. J_1 est doté d'un traqueur lors de l'entraînement du modèle pour suivre la balle. Il verra donc ses actions être guidées par le mouvement de balle. Par conséquent, il sera rendu imbattable afin de faciliter l'entraînement de J_2 tout en limitant le coût computationnel de l'entraînement. Cela conduit ainsi à une récompense automatique de "-10" au compteur dans chaque épisode. Enfin, il sera naturellement possible de remplacer l'ordinateur J_1 par un joueur réel pour qu'il puisse affronter notre modèle entraîné stocké dans un fichier ".pth".

2.2 Discrétisation de l'Environnement

Dans un environnement 640×480 pixels, on a $x \in [-320; 320]$ et $y \in [-240; 240]$. La balle pouvant se déplacer de manière positive ou négative sur les axes x et y, nous avons deux possibilités pour chacun des axes avec la position de r_2 sur l'axe y. De ce fait, en partant d'un produit cartésien, nous obtenons une estimation du nombre d'états possibles dans l'espace continu [1] : $|\text{Espaces d'états}| = |x_{\text{position}}| \times |y_{\text{position}}| \times |x_{\text{vitesse}}| \times |y_{\text{vitesse}}| \times |r_{\text{agent}}| \approx 589\,824\,000$ *(cf. Annexes, Taille de l'Espace d'Etats en Fonction de la Dimension de l'Environnement, figure 4)*. La taille de l'espace d'états augmentant de manière exponentielle en fonction des dimensions de l'environnement, nous pouvons discrétiser l'environnement afin d'alléger l'espace des

1. Pour simplifier le calcul, nous supposons qu'il n'existe qu'un seul état par pixel au lieu des 2^{32} états possibles avec une précision float32.

états. En optant pour une discrétisation de l'environnement, nous pouvons réduire l'environnement au format 85×85, ainsi, la position x et y de la balle sera discrétisée en 85 intervalles. Par ailleurs, la position de r_2 contrôlée par l'agent est elle aussi discrétisée en 85 intervalles.
Nous verrons dans les sections suivantes que la discrétisation de l'environnement se présente comme une solution alternative face aux limites que présente l'entraînement d'un agent Q-learning. Celui-ci n'est pas en mesure d'apprendre dans un cadre continu en raison de son utilisation de table contrairement au DQN et DDQN qui utilise des réseaux. Nous verrons également que nous pourrons exploiter la discrétisation de l'environnement afin d'alléger le coût computationnel et par conséquent l'entraînement des agents. Cela représente quand même une perte de précision puisqu'en effet, deux états différents dans le jeu peuvent être regroupés dans la même catégorie discrète, ce qui peut limiter la capacité de l'agent à apprendre des stratégies plus fines.

3 Apprentissage Famille Q

3.1 Q-Learning (Reinforcement Learning)

Le Q-Learning est une méthode d'apprentissage par renforcement qui permet à un agent d'apprendre une stratégie optimale pour maximiser une récompense cumulative en interagissant de manière répétée avec son environnement (Watkins and Dayan (1992)). Il s'agit d'une méthode d'apprentissage hors politique, dans laquelle l'agent apprend de manière autonome, sans connaissance préalable des règles optimales et indépendamment de la politique suivie par l'agent pendant l'entraînement. Dans notre cas, l'agent est représenté par la raquette du joueur 2 et l'environnement est tel que nous l'avons décrit auparavant. L'objectif est de trouver la fonction de valeur d'action optimale, $Q^*(s, a)$, qui donne la valeur maximale des récompenses futures cumulées pour un état $s \in S$ et une action $a \in A$. S représente l'ensemble des états que peut observer l'agent. Cela comprend la position de la balle, sa vitesse et la position de sa raquette. A représente les actions disponibles pour l'agent à savoir monter, descendre ou ne pas bouger sa raquette. La récompense R est un signal donné à chaque action pour guider l'agent vers un comportement optimal.
La fonction de valeur notée $Q(s, a)$ estime la qualité de chaque action a dans un état s. La mise à jour de la fonction de valeur est basée sur l'équation de Bellman et se définit comme ci-dessous :

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] \tag{1}$$

où α est le taux d'apprentissage qui contrôle la rapidité de mise à jour de la fonction Q. γ est le facteur

d'actualisation qui pondère l'importance des récompenses futures par rapport aux récompenses immédiates. s' est l'état atteint après avoir pris l'action a dans l'état s. $\max_{a'} Q(s', a')$ représente l'estimation de la meilleure récompense future possible à partir de l'état s'.
L'erreur de Différence Temporelle (TD error), notée δ, mesure l'écart entre la valeur actuelle estimée Q(s,a) et la cible. Elle est définie par l'expression suivante : $r + \gamma \max_{a'} Q(s', a') - Q(s, a)$ où $Q(s, a)$ est l'estimation actuelle de la valeur de l'état-action, $r + \gamma \max_{a'} Q(s', a')$ est la cible mise à jour pour cette estimation. Un δ négatif signifie que l'agent sous-estime la récompense future, tandis qu'un δ positif indique une surestimation. De ce TD error en découle la fonction de perte (Loss), qui grâce à son terme quadratique, pourra être minimisée telle que :

$$Loss = \frac{1}{2}(r + \gamma \max_{a'} Q(s', a') - Q(s, a))^2 = \frac{1}{2}\delta^2$$

Pour chaque action prise, l'agent met à jour $Q(s, a)$ en fonction de la récompense obtenue et de l'estimation des valeurs futures. L'objectif est de converger vers une fonction Q qui permet à l'agent de prendre les meilleures décisions à long terme. Dans le jeu de Pong, l'objectif est d'apprendre une politique qui permet à l'agent de maximiser la récompense cumulative en renvoyant la balle le plus de fois possible. Pour réussir à apprendre de façon optimale, l'agent doit naviguer entre deux stratégies : l'exploration et l'exploitation. L'exploration consiste à essayer de nouvelles actions pour découvrir de nouvelles stratégies et l'exploitation qui est le fait d'utiliser une politique connue pour maximiser la récompense attendue. Dans notre cas, nous opterons pour une stratégie d'actions exploratoire avec un paramètre $\epsilon = 0.9$, permettant à l'agent d'effectuer une action aléatoire avec une probabilité de 90% et une action qui maximise la récompense connue avec une probabilité de 10%. Nous analyserons également l'effet de l'intégration d'un facteur de décroissance ϵ_{decay} sur l'apprentissage qui a pour objectif de transitionner entre une stratégie d'exploration ($\epsilon = 0.90$) et une stratégie gourmande ($\epsilon = 0.10$).
Cependant, l'apprentissage d'un agent par Q-learning pour jouer à Pong n'est pas optimale. En effet, l'agent apprend en temps réel à chaque interaction avec l'environnement, ce qui peut entraîner une forte corrélation entre les transitions successives et rendre l'apprentissage instable, voir bloquer l'agent dans un optimal local. Nous verrons dans les sous-sections suivante que l'utilisation de réseaux permettra d'obtenir un apprentissage stable qui converge rapidement à l'aide d'une gestion de mémoire d'expérience (replay buffer) efficace.

3.2 Deep Q-Network (Deep Reinforcement Learning)

L'apprentissage profond (LeCun, Bengio and Hinton, 2015) est un outil qui permet de modéliser des environnements complexes et de traiter des données d'entrée de grande dimension. L'objectif de cette sous-section est de comparer l'entraînement d'un agent qui joue au jeu Pong en utilisant les algorithmes DQN et DDQN. Ceux-ci utilisent des réseaux de neurones convolutionnels (Fukushima (1969)) dans le cadre d'une analyse d'image à haute résolution (Krizhevsky, Sutskever and Hinton (2012)). Cette comparaison nous permettra de mettre en évidence les avantages des approches basées sur l'apprentissage profond pour l'approximation des fonctions de valeur, notamment dans un environnement continu et dynamique comme Pong, par rapport aux limitations du Q-learning en raison de son approche tabulaire. Contrairement à l'approche tabulaire du Q-learning, le DQN est plus adapté à des environnements complexes de haute dimension, comme le jeu Pong (avec une résolution de 640×480 pixels). Grâce à l'utilisation d'un réseau de neurones, le DQN peut approximer la fonction de valeur $Q(s, a)$ sans avoir besoin de stocker explicitement chaque état comme le ferait un agent Q-learning tabulaire. Nous verrons dans cette section que le DQN et le DDQN stockent les nouvelles expériences dans un replay buffer dans lequel un échantillon aléatoire de cette mémoire est utilisé pour entraîner le réseau de neurones via une optimiseur mini-batch. En généralisant l'espace des états via cette gestion de mémoire, l'agent DQN va apprendre une représentation abstraite des états pour mieux réagir à des états jamais rencontrés avant du fait que ces états puissent être similaires à ceux déjà rencontrés dans les données d'entraînement. Ainsi l'utilisation d'un réseau permet de manipuler des espaces d'états continus sans mémoriser explicitement chacun des états dans l'univers des possibles.

3.2.1 Le Deep Q-Network

Dans le cadre de l'algorithme DQN, le réseau de neurones est composé d'une couche d'entrée qui est un vecteur d'états contenant cinq éléments. Ils représentent les informations permettant au réseau de prédire l'action optimale en couche de sortie. Premièrement, il y a la position de la balle sur les axes x ($etat_{\mathrm{x}}$) et y ($etat_{\mathrm{y}}$), normalisée par rapport aux dimensions de la fenêtre de jeu. Ensuite, nous incluons la vitesse de la balle sur les axes x ($etat_{\mathrm{vx}}$) et y ($etat_{\mathrm{vy}}$), normalisée par rapport à une vitesse maximale. Enfin, la position verticale de la raquette contrôlée par l'agent (r_{posi}). Cette représentation de l'état, notée $s_t \in \mathbb{R}^5$, capture toutes les informations nécessaires qui permettent au réseau de calculer l'action optimale à chaque étape de l'épisode *(cf. Annexes, Schéma de Nos Réseaux de Neurones DQN et Double DQN, figure 6)*. Le réseau DQN se compose de trois couches entièrement connectées. La première couche est une couche linéaire qui prend les 5 caractéristiques de l'état comme entrée et produit une sortie de dimension 128. Une autre couche linéaire de 128 neurones, activée par la fonction ReLU (Rectified Linear Unit). Et une couche

de sortie linéaire de 3 neurones correspondant aux trois actions possibles, à savoir, monter, descendre, ou rester sur place. Chaque couche cachée est activée par une fonction ReLU, définie telle que : $f(x) = \max(0, x)$. Cette non-linéarité est essentielle, car elle permet au réseau d'apprendre des représentations complexes des états pour optimiser les actions de l'agent. La sortie du réseau est un vecteur $Q(s, a) \in \mathbb{R}^3$ contenant les Q-values estimées pour chaque action a possible dans l'état s_t. Autrement dit, ces Q-values représentent l'estimation de la valeur attendue de chaque action donnée l'état actuel. L'action optimale a^* est sélectionnée par la politique π telle que : $a_t^* = \arg\max_a Q(s_t, a)$. La valeur de $Q(s, a)$ est mise à jour en fonction de la récompense reçue r, de l'état suivant s', et de l'action qui maximise la valeur attendue à partir de cet état.

$$Q_{\text{principal}}(s, a) = r + \gamma \cdot \max_{a'} Q_{\text{cible}}(s', a') \tag{2}$$

Le DQN est constitué de deux réseaux, un réseau principal et un réseau cible. Le réseau principal est entraîné à chaque itération en utilisant les transitions (s, a, r, s') de l'environnement et vise à générer des valeurs Q estimées pour l'état actuel et permet ainsi de choisir l'action optimale. Le réseau cible est quant à lui utilisé pour calculer la Q-value de l'état suivant (s') et sera synchronisé avec le réseau principal pour un intervalle donné qui sera discuté en fin de section.
La valeur cible (y) est calculée en prenant en compte la récompense reçue r et la meilleure valeur estimée à partir de l'état suivant s', mais à l'aide du réseau cible Q_{cible}. L'utilisation du réseau cible permet de stabiliser l'apprentissage : $y = r + \gamma \max_{a'} Q_{\text{cible}}(s', a')$.
Nous retrouvons γ, le facteur d'actualisation, qui pondère l'importance des récompenses futures par rapport aux récompenses immédiates : $Q_{\text{principal}}(s, a) = r + \gamma \max_{a'} Q_{\text{cible}}(s', a')$. Nous définissons l'erreur de Bellman (δ) par la différence entre la valeur cible calculée (y) et la valeur actuelle estimée par le réseau ($Q(s, a)$). Elle permet d'ajuster les poids du réseau de manière à minimiser la fonction de perte : $Loss = \mathbb{E}\left[\left(Q_{\text{principal}}(s, a) - (r + \gamma \cdot \max_{a'} Q_{\text{cible}}(s', a'))\right)^2\right]$.

3.2.2 Le Double Deep Q-Network

La principale différence dans le DDQN par rapport au DQN réside dans son utilisation du réseau principal et du réseau cible. En effet, de manière identique au DQN, le DDQN utilisera le réseau principal pour mettre à jour chaque étape d'apprentissage. Cependant, le DDQN va également sélectionner l'action optimale à l'état suivant s' au lieu d'utiliser le réseau cible. De ce fait, le réseau principal servira à déterminer l'action (a_t^*) qui maximise notre valeur Q future (en s'). Le réseau cible va quant à lui calculer la valeur Q associée à cette action optimale. Cette différence, inspirée du Double Q-learning, vise à réduire le biais de surestimation des Q-values. C'est un problème inhérent aux méthodes Q-learning et DQN, en raison de l'utilisation de la même fonction Q (réseau cible) qui est utilisée à la fois pour choisir l'action optimale

et pour évaluer la valeur de cette action. Cela conduit alors souvent à une surestimation des valeurs Q.

Équation Double Deep Q-Network

Comme expliqué précédemment, le DDQN utilise deux réseaux de neurones $Q_{\text{principal}}$ et Q_{cible}, mis à jour alternativement, pour estimer les valeurs d'actions. Les valeurs Q sont mises à jour selon les équations suivantes :

$$Q_{\text{principal}}(s,a) = r + \gamma \cdot Q_{\text{cible}}\left(s', \arg\max_{a'} Q_{\text{principal}}(s',a')\right) \tag{3}$$

Ici, le réseau $Q_{\text{principal}}$ est utilisé pour choisir l'action optimale dans l'état suivant s', mais la valeur de cette action est évaluée par le réseau Q_{cible}. Cette évaluation croisée réduit le risque de surestimation des valeurs Q. La valeur cible est calculée de manière à réduire le biais de surestimation :
$y = r + \gamma Q_{\text{cible}}\left(s', \arg\max_{a'} Q_{\text{principal}}(s',a')\right)$.
Dans cette équation, le réseau $Q_{\text{principal}}$ est utilisé pour sélectionner l'action qui maximise la récompense future, tandis que le réseau Q_{cible} est utilisé pour estimer la valeur de cette action. Cette séparation entre la sélection de l'action et l'évaluation de la valeur permet d'obtenir une estimation plus fiable de la valeur Q.
L'erreur de Bellman est définie comme la différence entre la valeur cible (y) et la valeur actuelle estimée par le réseau $Q_{\text{principal}}$: $\delta = y - Q_{\text{principal}}(s,a)$ auquel découle la fonction de perte :
$Loss = \mathbb{E}\left[\left(Q_{\text{principal}}(s,a) - \left(r + \gamma \cdot Q_{\text{cible}}\left(s', \arg\max_{a'} Q_{\text{principal}}(s',a')\right)\right)\right)^2\right]$ qui est ensuite utilisée pour ajuster les poids du réseau de manière à réduire l'erreur d'estimation.

3.2.3 Optimisation et Hyperparamètres

En appliquant une descente de gradient mini-batch, nous trouvons un équilibre entre variance et vitesse de convergence, ainsi qu'entre mémoire et performance, comparé à la descente de gradient par batch complet ou stochastique. La descente par batch complet est trop coûteuse en mémoire pour un environnement dynamique comme Pong. La descente stochastique, qui utilise un seul exemple à chaque mise à jour, introduit du bruit en raison d'une variance élevée. Cela conduit à un apprentissage moins stable et plus lent à converger. Ainsi, la descente de gradient mini-batch permet d'éviter le blocage dans des minima locaux sans compromettre l'efficacité computationnelle en utilisant un sous-ensemble de données d'entraînement, soit un batch de 128 dans notre cas. Étant donné que l'environnement Pong est dynamique et que les situations varient entre chaque épisode, notamment par la direction aléatoire de prend la balle à son lancement, nous utilisons l'optimiseur stochastique Adam pour permettre à notre agent de s'adapter progressivement sans attendre d'accumuler une grande quantité de données d'entraînement.

Cela permet de faire des bonds en s'adaptant à la raideur de la pente. En se basant sur des mini-batches tirés de données d'entraînement, il aide à gérer l'instabilité des gradients et à accélérer la convergence vers des politiques de jeu optimales (Kingma, 2014).

Le DQN s'appuie également sur une mémoire d'expérience (le replay buffer) qui stocke les transitions (s, a, r, s') et permet de prélever aléatoirement des échantillons pour briser la corrélation entre les expériences successives et stabiliser l'apprentissage. L'entraînement se fait par mini-batchs de taille 128, équilibrant ainsi la variance et la vitesse de convergence. La taille des mini-batchs permet de profiter des avantages des méthodes stochastiques tout en conservant une efficacité computationnelle. Dans notre cas, à chaque itération, un mini-batch de taille 128 est prélevé aléatoirement dans le replay buffer ayant une mémoire de 1'000'000. Il aide à réduire la corrélation entre les transitions successive.

Comme pour le DQN, un facteur d'actualisation élevé (typiquement 0.99) est utilisé pour donner une grande importance aux récompenses futures. Il incite l'agent à prendre des actions qui maximisent les gains à long terme. Dans notre cas, ce facteur d'actualisation permettra à l'agent de positionner sa raquette en avance à la coordonnée estimée dans laquelle la balle arrivera. Le taux d'apprentissage détermine la vitesse de mise à jour des poids du réseau. Un taux d'apprentissage typique est de 0.0001, assurant une convergence lente mais plus stable. Notre réseau cible est mis à jour toutes les 10 itérations. Cela permet de stabiliser les cibles utilisées pour la mise à jour des poids du réseau principal. En termes de stratégie d'exploration pour nos modèles de réseaux, nous opterons pour une stratégie epsilon-decay qui permet d'intégrer une probabilité d'exploration qui régresse au fil des épisodes afin de transiter d'une stratégie exploratoire ($\epsilon - exploratory$) à une stratégie cupide ($\epsilon - greedy$) à l'aide d'un facteur de décroissance (ϵ_{decay}). Cette stratégie aura pour conséquence d'allonger la durée de convergence mais avec pour finalité d'augmenter l'efficacité du modèle entraîné par rapport à un modèle avec stratégie $\epsilon - greedy$ constante, tel que $\epsilon = 0.1$ (Sutton (2018), p.132). Dans notre cas, à partir d'un $\epsilon_{\text{init}} = 0.9$, nous appliquerons un $\epsilon_{\text{decay}} = 0.975$ par épisode afin d'atteindre $\epsilon_{\text{min}} = 0.1$ telle que $\epsilon_{\text{init}} \times \epsilon_{\text{decay}}^{n} = \epsilon_{\text{min}}$, c'est-à-dire $0.9 \times 0.975^{n} = 0.1$. Dans notre cas, pour $\epsilon_{\text{decay}} = 0.975$, il faudra atteindre 87 épisodes pour atteindre $\epsilon_{\text{min}} = 0.1$. Notons qu'il serait plus efficace de choisir une valeur ϵ_{decay} plus élevée telle que $\epsilon_{\text{decay}} = 0.995$; cependant, cela aura pour effet d'augmenter de manière exponentielle le nombre d'épisodes pour atteindre $\epsilon_{\text{min}} = 0.1$ *(cf. Annexes, Courbes de Décroissances Exponentielles pour Différents valeurs ϵ_{decay}, figure 5)*. De ce fait, pour des raisons computationnelles évidentes, nous nous limiterons à une valeur $\epsilon_{\text{decay}} = 0.975$.

Ces hyperparamètres jouent un rôle essentiel dans la performance de l'agent DDQN. Ils aident à équilibrer entre exploration et exploitation, à stabiliser l'apprentissage grâce au réseau cible, et à améliorer la précision des estimations des valeurs Q. En utilisant deux réseaux, le DDQN atténue efficacement le biais de surestimation inhérent au DQN classique et permet d'obtenir des agents plus robustes et stables.

4 Résultats

4.1 Métriques pour évaluer le modèle

Dans le contexte de l'évaluation de nos modèles, nous exploiterons les métriques de Temporal Difference Error (TD Error) et de Fonction de Perte (Loss) afin de comparer la vitesse et le niveau de convergence entre nos différents modèles en supplément de la TD Error qui nous permettra d'identifier le potentiel surajustement pour certains modèles.
Nous avons vu dans la section précédente que nous pouvons déduire la fonction de perte (Loss) à partir du TD error. Notre jeu étant dynamique, nos agents apprennent au fil des épisodes en se basant sur les données d'entraînements obtenues sur différents frames [2] qui composent les épisodes, ainsi nous obtenons la fonction de perte suivante : $\text{Loss}_{\text{frame}} = \frac{1}{|\text{frame}|} \sum_{i=1}^{|\text{frame}|} \frac{1}{2}\delta_i^2$
Ainsi, si nous souhaitons déterminer la fonction perte moyenne par épisode, il faudra prendre en compte le nombre de frame par épisode (celui-ci étant par nature croissant avec la durée de l'épisode). Dans le cadre du DQN, nous avons sélectionné un batch de 128 transitions, de ce fait, nous retiendrons 128 frames par épisode. Afin d'obtenir la perte moyenne pour chaque épisode nous prendrons la somme des 128 Loss obtenus que nous diviserons par le nombre de transition dans le batch.

4.2 Analyse des Métriques d'Evaluation

Nous nous retrouvons avec les modèles suivant : Q-learning (Environnement discret, discret avec récompenses aléatoires dans la table et continu), DQN (Environnement discret et continu) et DDQN (Environnement discret et continu).

Lecture Modèles Q-learning

Le Q-learning tabulaire n'étant pas naturellement adapté aux environnements dynamiques complexes, la fonction de perte (Loss) n'est pas directement interprétable. En effet, durant les premiers frames, la fonction de perte retournera des valeurs faibles, voire nulles, car le modèle n'arrive pas à découvrir des stratégies efficaces, ce qui ne reflète en rien sa performance réelle. En observant les 100 premiers épisodes, nous pouvons lire que les valeurs Q ne s'ajustent pas et que les True Values sont supérieures aux Estimates Values. Cela indique un sous-apprentissage. Pour pallier à cette difficulté, nous avons tenté d'intégrer du bruit dans le modèles Q-learning en implémentant des récompenses aléatoires dans la table (Q-learning

2. Un frame désigne une seule image dans une séquence d'images, comme dans une vidéo. Chaque frame capture un instantané de l'environnement ou de la scène observée. Les frames peuvent être analysés séquentiellement pour extraire des informations, suivre des objets en mouvement, ou détecter des changements.

Rdm). L'algorithme Q-learning tabulaire aura beaucoup de difficulté à s'adapter au lancement aléatoire de balle et aura tendance à rester dans l'un des deux coins de l'environnement afin d'avoir une chance sur deux de touche la balle une première fois. De ce fait, nous observons une évolution constante de nombre de touches cumulée, montrant que l'agent ne tire que peu d'expérience au fil de son apprentissage. Nous pouvons cependant remarquer quelques éléments pertinents. D'une part, nous pouvons voir que conformément à nos attentes, le Q-learning continu aura plus de difficulté à toucher la balle que dans un environnement discret en raison de la complexité de l'environnement continu. Nous pouvons lire qu'au cours de son entraînement il n'aura touché la balle qu'à 609 reprises, soit un peu plus de la moitié du temps. D'une autre mesure, si nous nous concentrons sur l'entraînement des modèles Q-learning dans l'environnement discret, il est intéressant d'observer qu'ajouter une récompense aléatoire (entre -0.01 et 0.01) pour chacune des actions améliorera sensiblement le résultat du Q-learning, passant de 849 touches (Q-Learning Discret) à 1035 touches (Q-learning Rdm).

Contrairement à nos attentes, nous pouvons voir que l'ajout d'un terme de décroissance (ϵ_{decay}) à la stratégie d'exploration n'aura pas eu d'effet positif mais au contraire, aura renfermé le modèle dans un optimum local. En effet, nous pouvons voir graphiquement au nombre de touches que le Q-Learning Discret *(cf. Annexes figure 5)* arrive à légèrement surpasser le Q-Learning Discret mais voit rapidement ses performances diminuer en se faisant rattraper par le Q-Learning continu en ayant à peine plus d'une chance sur deux de touche la balle (541 touches sur 1000 épisodes).

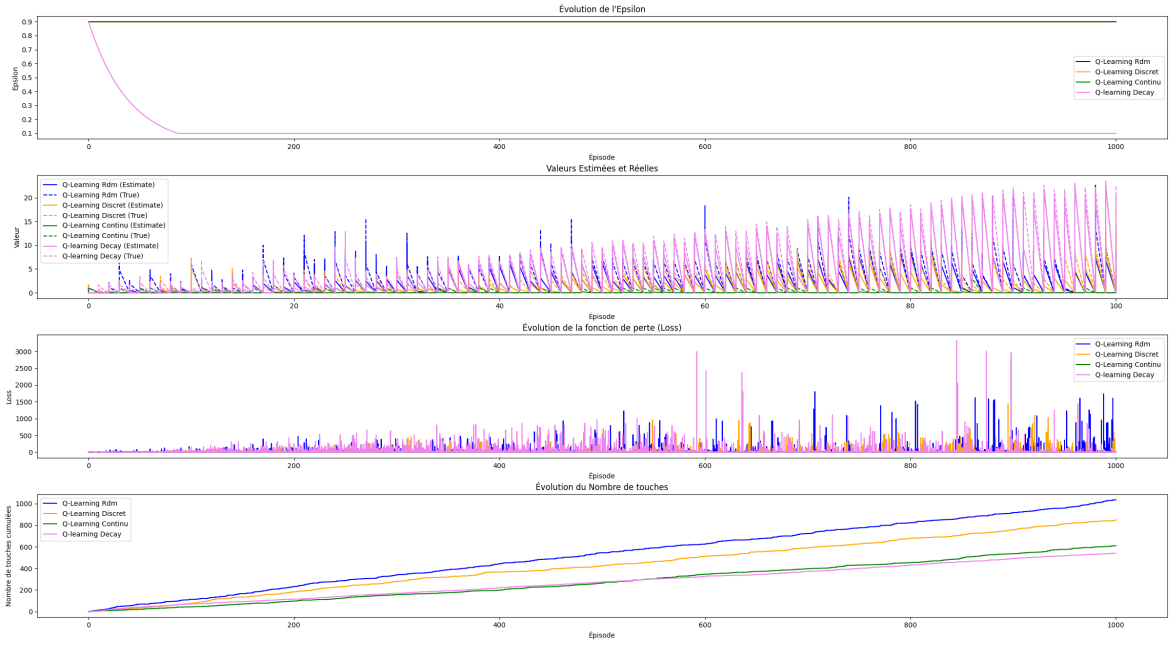


Figure 1. Présentation des Principales Caractéristiques des modèles Q-learning en Environnements Discrets et Continus

Lecture Modèles Deep Q-Network

Dans un premier temps, nous pouvons remarquer que dans un environnement discret, les modèles affichent de moins bons résultats que dans un environnement continu. Ce résultat s'explique par la faible taille de l'espace d'action qui limite l'intérêt d'un réseau par rapport à une table car, le modèle ne sera plus en capacité de généraliser efficacement. En effet, le fait de discrétiser l'environnement va mener à des transitions plus brutes entre les états plutôt que des petits changements qui seront similaires, entrainants donc de la surestimation pour les modèles. Dans le cadre du DQN discret, nous observons empiriquement sur une période prolongée (jusqu'à l'épisode 45 environ) que la TD Error (cf. Valeurs Estimées et Réelles) est anormalement élevé et supérieur à 0. Ce qui indique une surestimation du modèle. Par ailleurs, il est intéressant d'observer que le DDQN souffre moins de la surestimation en raison de sa capacité à se rapprocher rapidement d'un TD error raisonnable. Notre cas permet de montrer le lien entre une bonne estimation et la capacité du modèle à converger plus rapidement si l'on compare nos observations du DQN *(convergence en $\approx$ 58 épisodes)* et du DDQN *(convergence en $\approx$ 50 épisodes)*.

Figure 2. Présentation des Principales Caractéristiques des Modèles Deep Q-Network et Double Deep Q-Network en Environnements Discrets et Continus

(Ces résultats graphiques sont disponibles en plus grand format dans la section Annexe *(cf. figures 7, 8)*).

Nos observations peuvent s'expliquer par le fait qu'un modèle qui surestime les valeurs Q va commettre des erreurs plus importantes lors des épisodes suivants car il ne sera pas en mesure de s'adapter aux

nouvelles données d'apprentissage. Par conséquent, il convergera moins rapidement qu'un modèle qui s'ajuste correctement. Conformément à nos attentes, nous pouvons lire sur le graphique que d'après leur fonction de perte respective, le modèle DDQN affichent une performance sensiblement supérieur à celle du DQN. Nous pouvons remarquer que les modèles DQN et DDQN continus convergent tout deux au niveau de l'épisode 40 bien que la courbe de l'évolution de la fonction de perte du DDQN continu semble plus aplatie et plus stable que celle du DQN continu. Cela témoigne de l'efficacité du DDQN pour améliorer les performances durant l'entraînement en réduisant le biais de surestimation.

5 Conclusion et Discussion

Dans cette étude, nous avons exploré et comparé différentes approches d'apprentissage par renforcement appliquées au jeu Pong, en mettant en œuvre le Q-Learning, DQN et le DDQN. Chacune de ces méthodes a été évaluée sur sa capacité à apprendre et à généraliser dans des environnements discrets et continus, avec une attention particulière portée aux performances, à la stabilité et aux limites inhérentes à chaque modèle.

Les résultats confirment les limites du Q-Learning tabulaire dans des environnements complexes et dynamiques comme Pong. Cependant, ce modèle reste peu adapté aux espaces d'états continus en raison de l'impossibilité d'exploiter efficacement la structure de ces environnements. L'ajout de stratégies comme la discrétisation et les récompenses aléatoires ont permis d'améliorer sensiblement les performances du modèle, bien que ces dernières n'aient pas été suffisantes pour combler les lacunes fondamentales du Q-Learning dans ce contexte.

D'une autre mesure, les algorithmes basés sur les réseaux neuronaux, tels que le DQN et le DDQN, ont montré de meilleurs performance pour généraliser et apprendre dans des espaces d'états de grande dimension. Grâce à l'utilisation d'un replay buffer et de réseaux distincts, le DQN, a permis un apprentissage plus rapide et plus stable que le Q-Learning. Nous avons mis en évidence le biais de surestimation des Q-values qui affecte le DQN, entraînant davantage d'erreurs et un ralentissement de la convergence, particulièrement en milieu discret. Ce biais a été efficacement corrigé par le DDQN, qui utilise une séparation entre la sélection et l'évaluation des actions, améliorant ainsi la robustesse et la stabilité de l'apprentissage. Nos observations montrent que les agents entraînés sur des environnements continus affichent de meilleures performances que ceux sur des environnements discrets, grâce à leur capacité à généraliser des expériences passées. Le DDQN continu s'est distingué par sa convergence rapide et stable, confirmant son efficacité dans des environnements complexes dans lesquels les modèles peuvent être sujet au biais de surestimation.

Annexes

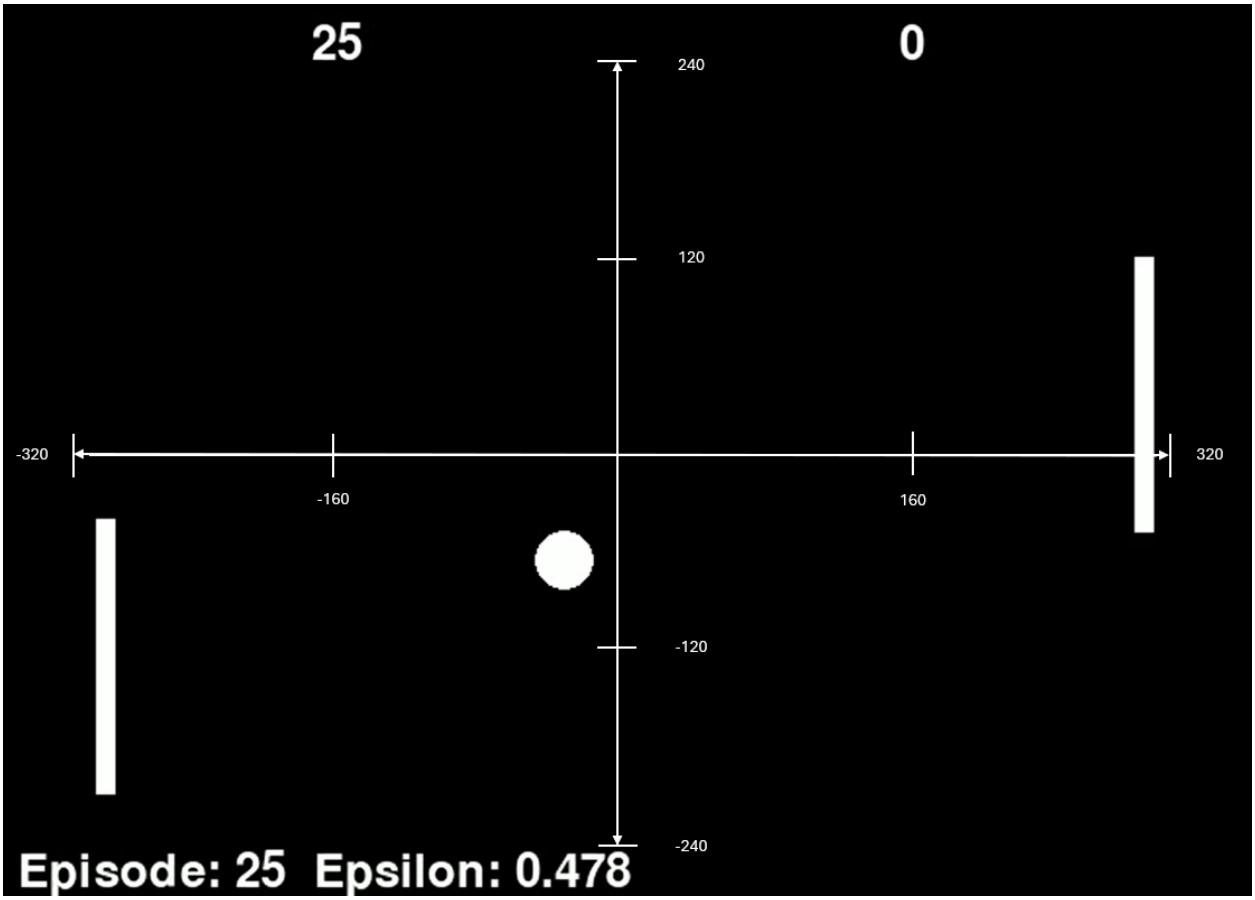


Figure 3. Notre Environnement Pong

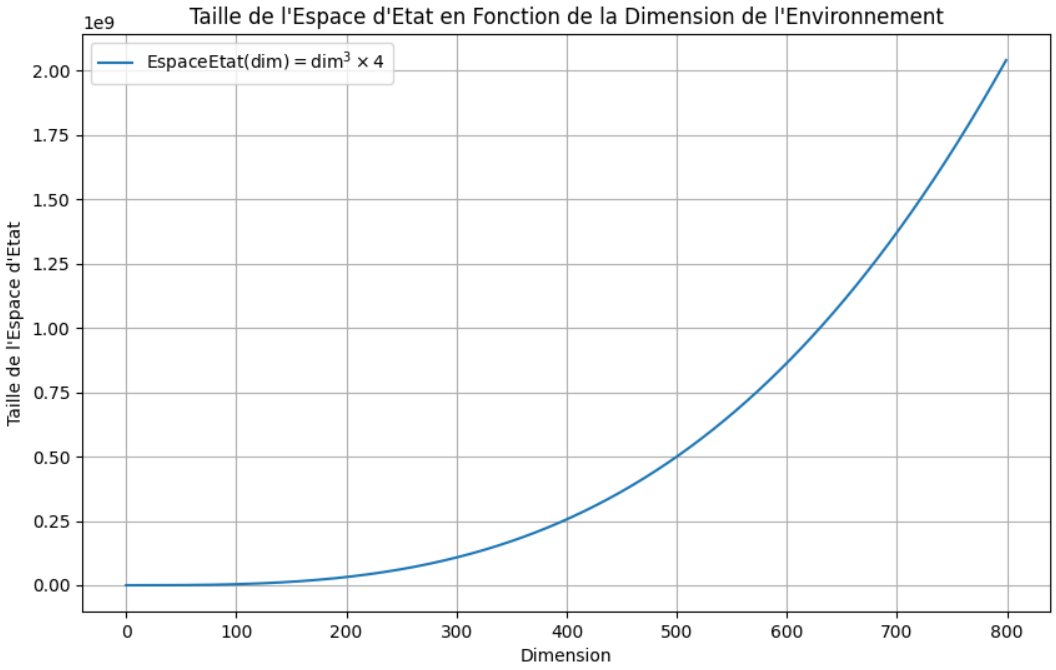


Figure 4. Représentation du Nombre d'États en Fonction de la Dimension de l'Environnement Pong

Par exemple, pour un environnement de 500×500, le nombre d'états sera donné par $|\text{Espace d'état}| = |x_{\text{position}}| \times |y_{\text{position}}| \times |x_{\text{vitesse}}| \times |y_{\text{vitesse}}| \times |r_{\text{agent}}|$, soit $500 \times 500 \times 2 \times 2 \times 500 \approx 0{,}5 \times 10^9 \approx 500\,000\,000$.

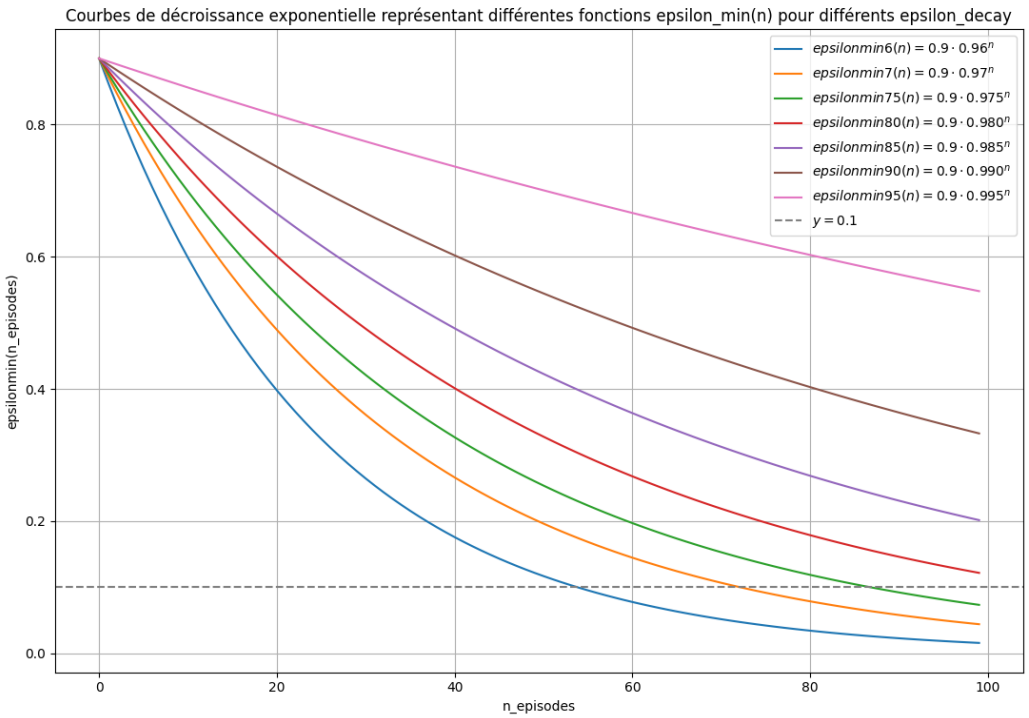

Courbes de décroissance exponentielle représentant différentes fonctions epsilon_min(n) pour différents epsilon_decay
epsilonmin6(n) = 0.9 · 0.96^n
epsilonmin7(n) = 0.9 · 0.97^n
epsilonmin75(n) = 0.9 · 0.975^n
epsilonmin80(n) = 0.9 · 0.980^n
epsilonmin85(n) = 0.9 · 0.985^n
epsilonmin90(n) = 0.9 · 0.990^n
epsilonmin95(n) = 0.9 · 0.995^n
y = 0.1
epsilonmin(n_episodes)
n_episodes
0.0
0.2
0.4
0.6
0.8
0
20
40
60
80
100

Figure 5

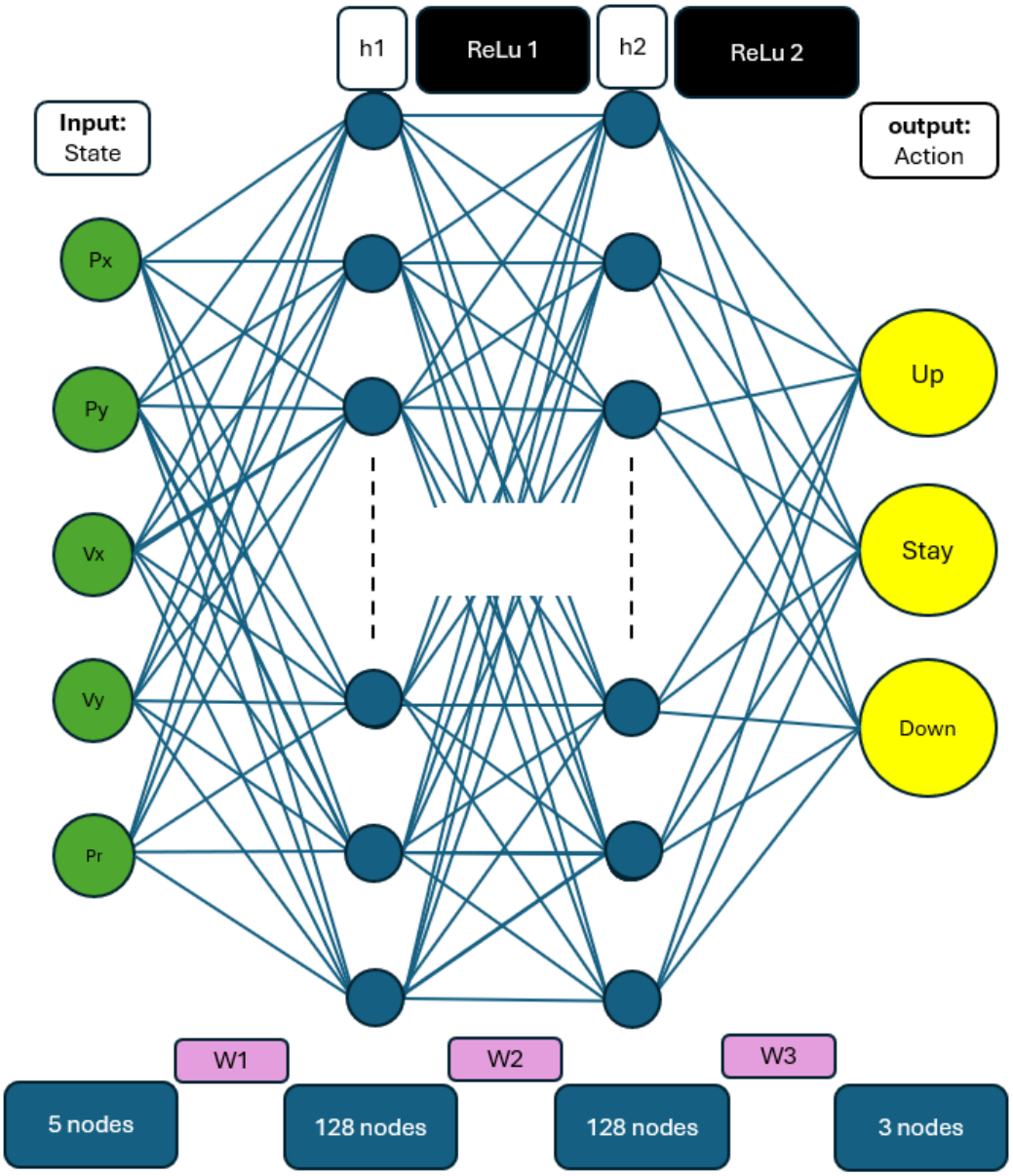


Figure 6. Schéma de Nos Réseaux de Neurones DQN et Double DQN

Figure 7. Evolution des Estimate Values et True Values des Deep Q-Network en Environnement Discret et Continu

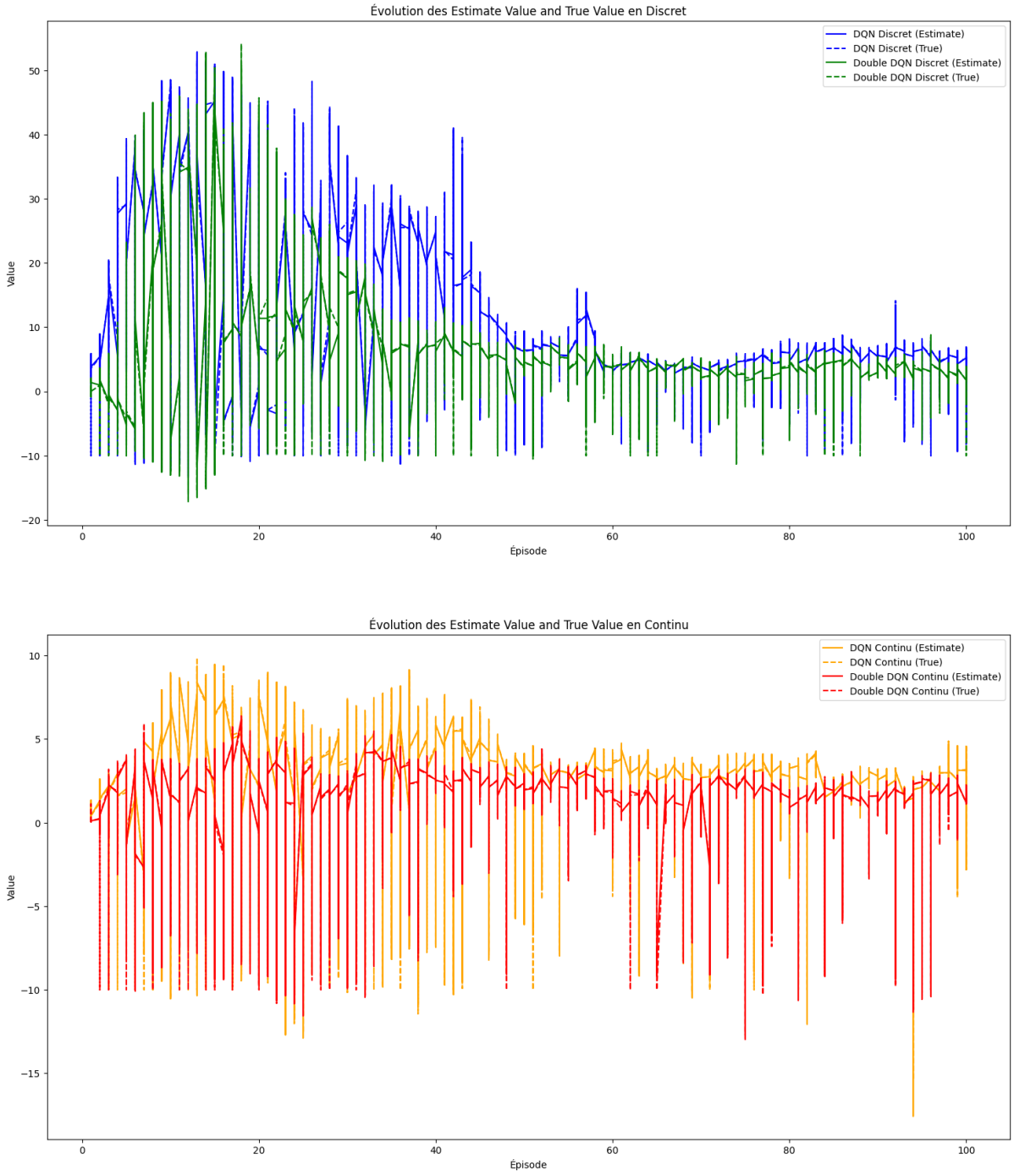
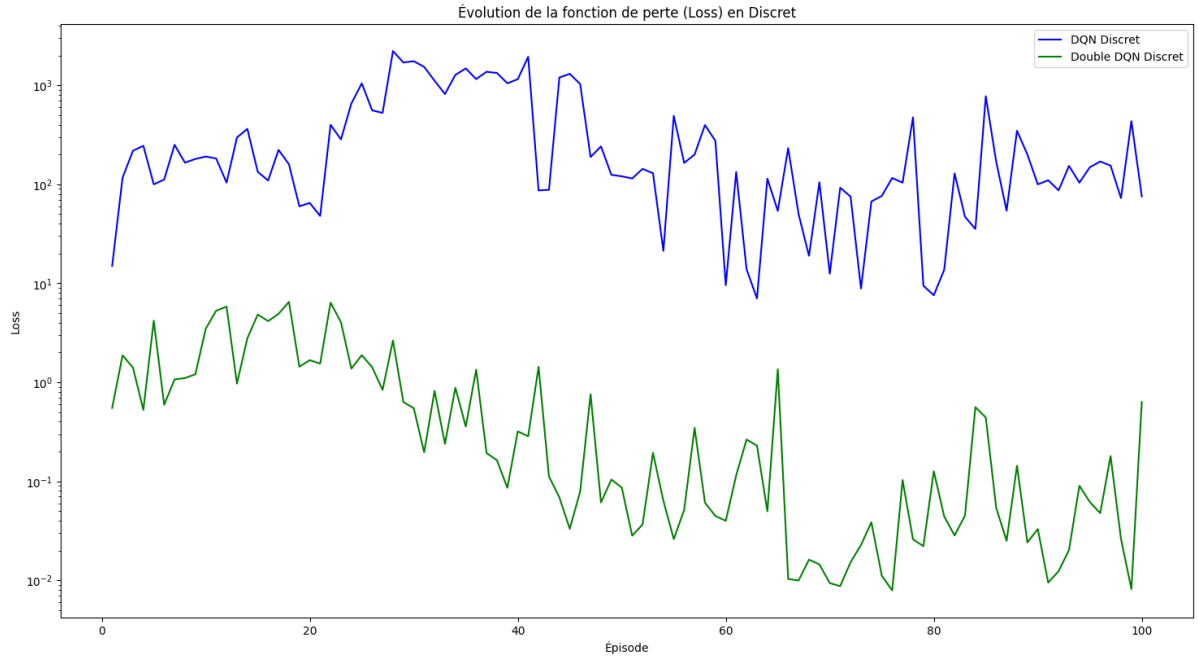
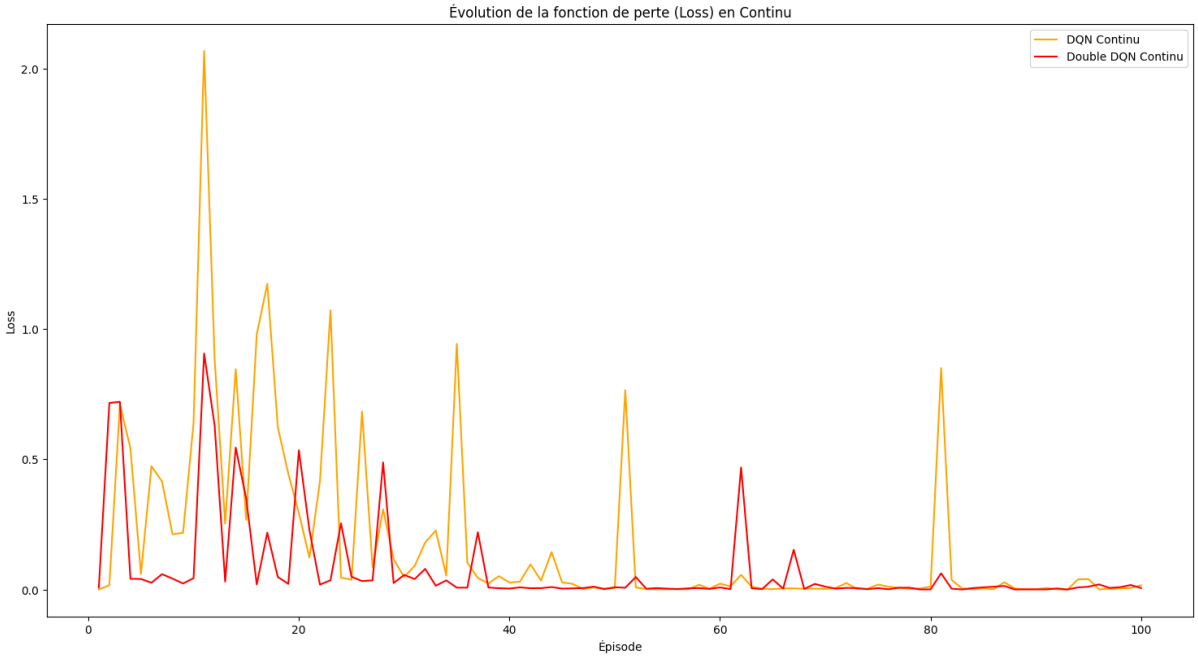


Figure 8. Evolution des Fonctions de perte des Deep Q-Network en Environnement Discret et Continu





Références

Fukushima, Kunihiko, "Visual feature extraction by a multilayered network of analog threshold elements," *IEEE Transactions on Systems Science and Cybernetics*, 1969, *5* (4), 322–333.

Hasselt, Hado, "Double Q-learning," *Advances in neural information processing systems*, 2010, *23.*

Hasselt, Hado Van, Arthur Guez, and David Silver, "Deep reinforcement learning with double q-learning," in "Proceedings of the AAAI conference on artificial intelligence," Vol. 30 2016.

Kingma, Diederik P, "Adam : A method for stochastic optimization," *arXiv preprint arXiv :1412.6980*, 2014.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, 2012, *25.*

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton, "Deep learning," *nature*, 2015, *521* (7553), 436–444.

Mnih, Volodymyr, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv :1312.5602*, 2013.

Sutton, Richard S, "Reinforcement learning : An introduction," *A Bradford Book*, 2018.

Watkins, Christopher JCH and Peter Dayan, "Q-learning," *Machine learning*, 1992, *8*, 279–292.

René Dudfield, *Pygame* - Récupéré de Documentation Pygame.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, *Pytorch* - Récupéré de Documentation Pytorch, PyTorch Foundation.

Ronan Collobert, Samy Bengio, Johnny Mariéthoz, *Torch* - Récupéré de Torch website, Idiap Research Institute & EPFL.