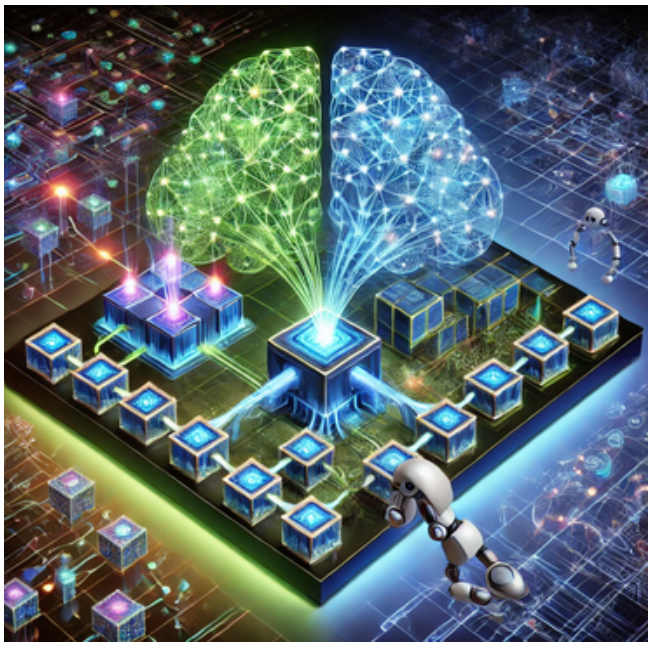
EUPI : École Universitaire de Physique et d'Ingénierie
4, avenue Blaise Pascal - CS 60026
63178 Aubière Cedex

Master PAR - Apprentissage

Analyse de l'article
Deep Reinforcement Learning with Double Q-learning



Élèves :

Lucas Pichon
Loubna Taleb

Professeur :

Céline Teuliere

Rendu le 17 Février 2025

Table des matières

1 Résumé et contribution de l'article

Dans cet article, les auteurs mettent en évidence l'algorithme **Double DQN**, en s'appuyant à la fois sur les travaux de **van Hasselt (2010)** [1], qui ont introduit le **Double Q-learning** dans un contexte tabulaire, et sur les travaux de **Mnih et al. (2015)** [2] relatifs à l'apprentissage des valeurs d'actions (***Q-values***) au moyen de réseaux de neurones profonds.

L'objectif principal est de montrer pourquoi la **surestimation** des Q-values apparaît lorsqu'on traite un **grand espace d'états** : à cause des **erreurs d'estimation** qui s'accumulent à chaque fois qu'on sélectionne l'action maximisant la Q-value. Les auteurs illustrent ce phénomène de surestimation à travers plusieurs jeux **Atari**, puis proposent l'algorithme de **Double Q-learning** (appliqué au cadre **Deep Learning**) pour atténuer ce biais. Ils démontrent expérimentalement que cette approche rend l'apprentissage plus **stable** et plus **performant** (notamment sur **Atari 2600**), en permettant de trouver des **politiques optimales** plus efficacement que la méthode **DQN** classique.

2 Analyse et explication de la méthode Double DQN

2.1 Contexte général : la fonction $Q(s, a)$ et l'équation de Bellman

En cours, nous avons vu que la fonction valeur-action $Q(s, a)$ représente la qualité d'exécuter l'action a dans l'état s, en tenant compte de la récompense immédiate et de la somme des récompenses futures actualisées. On peut l'écrire sous forme de l'équation de Bellman (*p. 50 du cours*) :

$$Q(s, a) \;=\; r(s, a) \;+\; \gamma \max_{a'} Q\big(s', a'\big),$$

où :
— s est l'état courant,
— a est l'action choisie,
— $r(s, a)$ est la récompense obtenue,
— $\gamma \in [0, 1]$ (ou parfois jusqu'à 1) est le facteur de discount qui pondère l'importance des récompenses futures.
Cette fonction $Q(s, a)$ peut être apprise de façon itérative par l'algorithme de *Q-learning*.

2.1.1 Exploration vs. Exploitation

Lors de l'apprentissage, il faut gérer le compromis entre *exploration* et *exploitation*. Par exemple, on peut choisir une stratégie ϵ-greedy :
— avec probabilité ϵ, on *explore* en choisissant une action au hasard,
— sinon, on *exploite* la connaissance actuelle et on choisit l'action qui maximise $Q(s, a)$.
Au fur et à mesure de l'apprentissage, on diminue ϵ pour favoriser progressivement l'exploitation.

2.2 Cas tabulaire vs. réseaux de neurones

Quand l'espace d'états (et d'actions) est de taille modeste, on peut mémoriser les valeurs $Q(s, a)$ dans un simple tableau. Cette approche, dite « tabulaire », permet d'obtenir une mise à jour directe de Q pour chaque paire (s, a). Cependant, dès que le nombre d'états devient considérable, cette stratégie devient inapplicable : on ne peut plus stocker ni explorer efficacement toutes les configurations possibles.

Pour les environnements plus complexes (en particulier lorsqu'on traite des images ou un grand nombre d'états), on recourt alors à un *réseau de neurones* afin d'approximer la fonction $Q(s, a; \theta)$. C'est la démarche proposée par **Mnih et al. (2013, 2015)**, connue sous le nom de *DQN*, où θ désigne les paramètres du réseau. L'idée centrale est de partir de l'équation de Bellman pour définir la fonction de perte (*loss*) : on compare la valeur estimée $Q(s_t, a_t; \theta)$ à une *cible*

$$y_t^{\mathrm{DQN}} = r_t + \gamma \max_{a'} Q\big(s_{t+1}, a'; \theta^-\big),$$

où θ^- est un « réseau cible » mis à jour moins fréquemment pour limiter l'instabilité de l'apprentissage. À chaque itération, on procède ensuite à une descente de gradient pour ajuster θ. Cette stratégie permet de gérer des espaces d'états gigantesques de manière plus efficace qu'une approche purement tabulaire.

2.3 Problème : la surestimation (*overestimation*)

Dans Q-learning tabulaire comme dans DQN, l'opérateur max est utilisé pour **sélectionner** et **évaluer** l'action. Si les valeurs Q sont entachées d'un *bruit* ou d'une erreur stochastique, ce max aura tendance à **gonfler artificiellement** la valeur maximale.

Exemple de l'effet du bruit [1]
On suppose un bruit ε dans l'estimation de Q, tel que $\mathbb{E}[\varepsilon] = 0$. Lorsque l'on prend $\max_{a'} Q\big(s, a'; \theta\big)$ sur plusieurs actions, la sélection tombe souvent sur la valeur la plus « chanceusement positive ». Au fil du temps, ce biais positif peut :

— **fausser** l'évaluation des actions (surestimation),
— **rendre** l'apprentissage moins stable,
— **mener** à une politique sous-optimale.

L'article de **van Hasselt et al.** vise à **corriger** cette surestimation en **dissociant** l'évaluation de Q-learning et la sélection de l'action maximisant Q. C'est le principe de *Double Q-learning*, qu'ils ont étendu au cadre profond (Double DQN). L'idée est de **séparer** l'opération $\arg\max_a$ (sélection) et la fonction max (évaluation) via deux réseaux de neurones ou deux estimateurs différents. Cette modification réduit significativement la surestimation et stabilise l'apprentissage comme montré dans l'article (figure 2 dans l'article de Van hasselt et al (2016))

2.4 Double Q-learning (2010)

Afin de mieux comprendre l'article et la notion de Double Q-learning, nous avons étudié les travaux de Hasselt (2010). Hasselt a constaté que l'apprentissage de la fonction de valeur d'action $Q(s, a)$ est affecté par le bruit lorsque l'évaluation et la sélection d'actions s'effectuent à partir de la même fonction, ce qui entraîne une surestimation des valeurs.

Pour remédier à ce problème dans le cadre de l'estimation tabulaire, Hasselt a proposé de travailler avec deux estimateurs, Q^A et Q^B, au lieu d'un seul. Ainsi, à chaque mise à jour de la fonction d'action Q, on utilise l'un des estimateurs pour sélectionner l'action maximale et l'autre pour évaluer cette action :

$$Q^A(S_t, A_t) \leftarrow Q^A(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma\, Q^B\big(S_{t+1}, \arg\max_a Q^A(S_{t+1}, a)\big) - Q^A(S_t, A_t)\Big],$$

$$Q^B(S_t, A_t) \leftarrow Q^B(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma\, Q^A\big(S_{t+1}, \arg\max_a Q^B(S_{t+1}, a)\big) - Q^B(S_t, A_t)\Big].$$

(voir l'Algorithme 2).

Exemple : la roulette
Dans l'article original, Hasselt illustre l'efficacité de Double Q-learning en comparant Q-learning et Double Q-learning dans un *jeu de roulette*. Cet environnement comporte environ 170 actions de pari (parier sur un nombre, sur une couleur, etc.), dont la plupart ont une espérance de gain négative. Q-learning, du fait de la surestimation, attribue progressivement des valeurs positives voire élevées à ces actions, dépassant largement ce qu'elles valent réellement (l'espérance se situe autour de -0.053 par mise). En revanche, Double Q-learning *limite* ce biais d'optimisme et converge bien plus rapidement vers des valeurs estimées proches de leur véritable rendement, nettement inférieures à celles que Q-learning continue de surestimer. Cet exemple illustre concrètement l'avantage de la **décorrélation** entre sélection et évaluation d'action, au coeur de la méthode Double Q-learning.

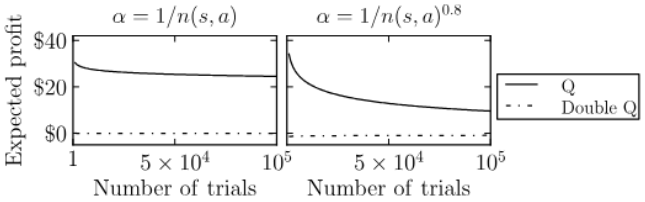


Figure 1: The average action values according to Q-learning and Double Q-learning when playing roulette. The 'walk-away' action is worth $0. Averaged over 10 experiments.

FIGURE 2.1 – Figure extraite de l'article [1].

2.5 Double DQN (2016)

En 2016, Van Hasselt, Guez et Silver proposent de transposer le principe de Double Q-learning au cadre *Deep Q-Network*, donnant naissance au **Double DQN** (DDQN). Cette extension a pour but de réduire la surestimation des valeurs d'action, déjà constatée dans Q-learning, mais qui se retrouve également amplifiée dans le contexte *deep*.

2.5.1 Rappels sur DQN

Le *Deep Q-Network* (DQN) repose sur deux réseaux de neurones distincts :
— **Réseau en ligne** (*online network*), dont les paramètres sont notés θ ;
— **Réseau cible** (*target network*), dont les paramètres sont notés θ^- et qui sont mis à jour moins fréquemment (tous les C pas ou via une mise à jour *soft*).

L'objectif est de stabiliser l'apprentissage en évitant que la même approximation de la fonction Q ne serve simultanément à la *sélection* et à l'*évaluation* des actions. Dans DQN, la *cible* de mise à jour pour une transition (s_t, a_t, r_t, s_{t+1}) est habituellement :

$$y_t^{\text{DQN}} \;=\; r_t + \gamma \max_{a'} Q\big(s_{t+1}, a'; \theta^-\big).$$

2.5.2 Modification de la cible dans Double DQN

La principale nouveauté de **Double DQN** (DDQN) consiste à **décorréler** la sélection de l'action arg max de l'estimation de sa valeur, en utilisant deux ensembles de paramètres :
— θ (du *online network*) pour *choisir* l'action $\arg\max_{a'} Q\big(s_{t+1}, a'; \theta\big)$;
— θ^- (du *target network*) pour *évaluer* la valeur de l'action choisie : $Q\big(s_{t+1}, a'; \theta^-\big)$.
On obtient donc la cible :

$$y_t^{\text{DoubleDQN}} \;=\; r_t \;+\; \gamma\, Q\Big(s_{t+1}, \arg\max_{a'} Q\big(s_{t+1}, a'; \theta\big); \theta^-\Big).$$

Cette séparation **limite le biais de surestimation** car le même réseau ne sert plus à déterminer à la fois arg max et la valeur associée à cette action.

2.5.3 Procédure algorithmique

La procédure de Double DQN suit les étapes globales du DQN classique, avec une **différence dans la cible** telle que définie en (2.5.2) :

1. **Collecte** des transitions (s_t, a_t, r_t, s_{t+1}) dans un *replay buffer*.
2. **Échantillonnage** aléatoire d'un *mini-batch* dans le *replay buffer*.
3. **Calcul de la cible Double DQN** pour chaque transition :

$$y_t^{\text{DoubleDQN}} \;=\; r_t \;+\; \gamma\, Q\Big(s_{t+1}, \arg\max_{a'} Q\big(s_{t+1}, a'; \theta\big); \theta^-\Big).$$

4. **Calcul de la perte** :

$$\mathcal{L}(\theta) \;=\; \mathbb{E}\Big[\big(y_t^{\text{DoubleDQN}} - Q(s_t, a_t; \theta)\big)^2\Big].$$

5. **Descente de gradient** pour mettre à jour θ.
6. **Mise à jour** périodique de θ^- par copie (ou partiellement, via un facteur τ).

2.5.4 Analyse et résultats expérimentaux des auteurs

Dans leurs travaux, Van Hasselt *et al.* comparent **Double DQN** et **DQN** sur plusieurs jeux Atari 2600, illustrés notamment par la Figure 3 (voir l'exemple ci-dessous). Les courbes **DQN** (en orange) tendent à surestimer la valeur de plusieurs actions, tandis que **Double DQN** (en bleu) conserve des estimations beaucoup plus proches de la valeur réelle. Cette réduction de la surestimation se traduit concrètement par :
— **Des valeurs d'action plus fiables :** Sur des jeux comme *Zaxxon*, *Time Pilot* ou *Asterix*, DQN montre un biais d'optimisme important (valeurs orange très au-dessus de la courbe réelle), alors que Double DQN (bleu) reste beaucoup plus aligné sur les retours attendus.
— **Une convergence plus rapide et de meilleurs scores :** Par exemple, pour *Wizard of Wor* et *Asterix*, les résultats indiquent que Double DQN atteint des scores supérieurs à ceux de DQN, et de manière plus stable.

— **Une plus grande stabilité :** Les courbes de Double DQN présentent moins de variance et s'avèrent globalement plus régulières au fil de l'apprentissage, montrant que l'agent est moins sujet aux fluctuations liées au biais de surestimation.

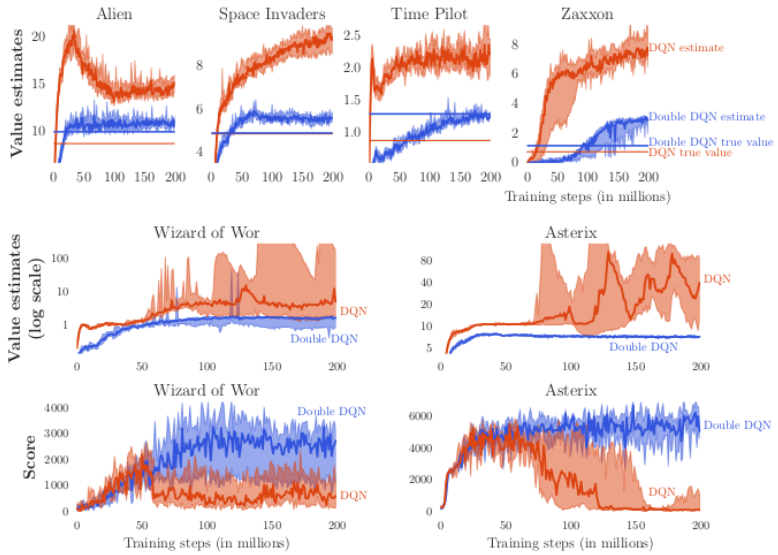


Figure 3: The **top** and **middle** rows show value estimates by DQN (orange) and Double DQN (blue) on six Atari games. The results are obtained by running DQN and Double DQN with 6 different random seeds with the hyper-parameters employed by Mnih et al. (2015). The darker line shows the median over seeds and we average the two extreme values to obtain the shaded area (i.e., 10% and 90% quantiles with linear interpolation). The straight horizontal orange (for DQN) and blue (for Double DQN) lines in the top row are computed by running the corresponding agents after learning concluded, and averaging the actual discounted return obtained from each visited state. These straight lines would match the learning curves at the right side of the plots if there is no bias. The **middle** row shows the value estimates (in log scale) for two games in which DQN's overoptimism is quite extreme. The **bottom** row shows the detrimental effect of this on the score achieved by the agent as it is evaluated during training: the scores drop when the overestimations begin. Learning with Double DQN is much more stable.

FIGURE 2.2 – Figure extraite de l'article [3].

3 Implémentation

3.1 Implémentation des algorithmes DQN et DDQN avec un réseau de neurones (NN)

Afin d'évaluer les performances des algorithmes de **Deep Q-Learning (DQN)** et **Double DQN (DDQN)**, nous avons commencé par les implémenter en utilisant un réseau de neurones simple (*fully connected neural network*) sur des environnements simples tels que `CartPole-v1`.

3.1.1 Architecture du réseau de neurones

Nous avons défini un réseau de neurones entièrement connecté pour approximer la fonction d'action-état $Q(s, a)$. Il prend en entrée un vecteur représentant l'état de l'environnement et produit un vecteur de dimension égale au nombre d'actions possibles, chaque élément correspondant à la valeur estimée $Q(s, a)$.

Le réseau est composé des couches suivantes :

1. Une couche d'entrée prenant le nombre d'observations (`n_observations`) de l'environnement.
2. Une couche cachée de 128 neurones avec une activation `ReLU`.
3. Une couche de sortie de taille égale au nombre d'actions possibles (`n_actions`).

3.1.2 Mémoire et Optimisation

Nous avons mis en place une mémoire tampon `ReplayMemory` qui stocke les transitions (s, a, r, s') rencontrées par l'agent. Cela permet de découpler la corrélation temporelle entre les expériences et d'améliorer la stabilité de l'apprentissage.

L'optimisation du réseau est effectuée via la fonction de perte *Smooth L1 Loss* et l'algorithme AdamW. Le réseau cible (`target_net`) est mis à jour en interpolant les poids du réseau principal (`policy_net`) avec un facteur τ.

3.1.3 Différence entre DQN et DDQN

La distinction entre les deux algorithmes repose sur la méthode de calcul des Q-values cibles :

1. Dans **DQN**, le même réseau est utilisé pour sélectionner et évaluer l'action optimale, ce qui entraîne une surestimation des valeurs Q.
2. Dans **DDQN**, l'action optimale est déterminée par le réseau principal, mais son évaluation est effectuée par le réseau cible, ce qui réduit la surestimation et améliore la stabilité.

Pour comparer les performances des deux méthodes, nous avons observé l'évolution des scores et des Q-values sur plusieurs épisodes d'entraînement.

3.2 Implémentation avec un réseau convolutif (CNN) pour les jeux Atari

Après avoir testé DQN et DDQN avec un réseau de neurones entièrement connecté sur des environnements simples, nous avons adapté notre implémentation pour les jeux Atari, qui impliquent un traitement d'images. Pour cela, nous avons remplacé les couches entièrement connectées du DQN par des couches convolutives permettant d'extraire des caractéristiques pertinentes à partir des observations visuelles.

3.2.1 Architecture du CNN

Le réseau convolutif utilisé est constitué des couches suivantes :

1. Trois couches de convolution avec des filtres de taille 5×5 et des activations `ReLU`.
2. Une couche entièrement connectée prenant en entrée les caractéristiques extraites par les convolutions.
3. Une couche de sortie de dimension égale au nombre d'actions possibles.

Cette architecture permet au réseau d'apprendre directement des représentations pertinentes à partir des images des jeux Atari.

3.2.2 Optimisation et sélection d'actions

La méthode d'optimisation reste identique à celle utilisée pour le réseau de neurones simple. Cependant, en raison de la complexité plus élevée des jeux Atari, nous avons ajusté certains hyperparamètres :

— Augmentation de la taille du batch à 64.
— Réduction du taux de mise à jour du réseau cible avec $\tau = 0.005$.
— Application d'une exploration *epsilon-greedy* ajustée avec une décroissance plus lente de ε.

3.2.3 Visualisation des performances

Pour évaluer les performances de notre implémentation sur des jeux simples comme `CartPole-v1` et `Acrobot-v1`, ainsi que sur des jeux Atari, nous avons analysé les métriques suivantes :

— L'évolution des **scores cumulés** au fil des pas d'entraînement, moyennée sur l'ensemble des épisodes joués.
— La progression des **Q-values moyennes** sur l'ensemble des états visités au cours des épisodes.
— L'évolution de la **récompense totale** obtenue à la fin de chaque épisode, moyennée sur un maximum de 5 exécutions (*runs*) lancées aléatoirement.
— L'évolution des **Q-values moyennes** sur l'ensemble du jeu pour chaque *run*, permettant d'évaluer la stabilité des estimations de Q.

Les résultats ont été représentés sous forme de courbes en fonction des **Training Steps** afin d'assurer une comparaison plus standardisée avec les études précédentes. De plus, nous avons également tracé ces métriques en fonction du nombre total **d'épisodes** joués pour mieux visualiser la progression de l'apprentissage.

4 Tests et Analyse des Résultats

4.1 Analyse des performances du réseau de neurones (NN) sur les jeux `CartPole-v1` et `Acrobot-v1`

Nous avons utilisé les jeux disponibles dans la bibliothèque `Gymnasium`, qui propose plusieurs environnements standards pour tester les algorithmes d'apprentissage par renforcement. Dans un premier temps, nous avons testé deux environnements classiques : `CartPole-v1` et `Acrobot-v1`. Ces environnements sont fréquemment utilisés dans la littérature pour évaluer les performances des algorithmes de contrôle par renforcement.

Environnement	Type d'actions	Espace d'observation	Objectif
`CartPole-v1`	Discret (2 actions)	4 dimensions	Équilibrer un poteau sur un chariot en mouvement
`Acrobot-v1`	Discret (3 actions)	6 dimensions	Faire basculer un pendule double jusqu'à une hauteur cible

TABLE 4.1 – Résumé des environnements testés avec la bibliothèque Gymnasium

FIGURE 4.1 – `Acrobot-v1` : un pendule double

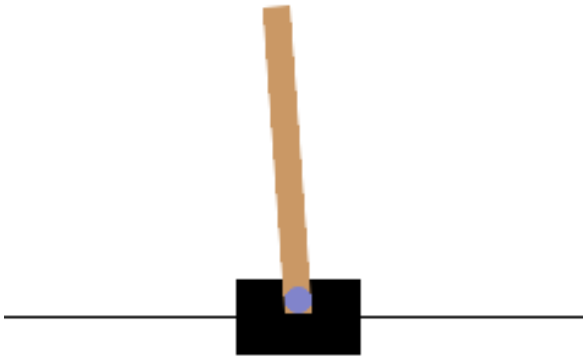


FIGURE 4.2 – `CartPole-v1`

FIGURE 4.3 – Illustration des environnements testés : `Acrobot-v1` et `CartPole-v1`.

4.1.1 Le choix des hyperparamètres

Nous avons ajusté les paramètres en fonction des spécificités de chaque environnement testé. Les hyperparamètres principaux incluent le nombre d'épisodes d'entraînement (`num_episodes`), le nombre de runs (`num_runs`), ainsi que les paramètres de la stratégie d'exploration-exploitation (`EPS_START`, `EPS_END`, `EPS_DECAY`) et les paramètres d'optimisation (`BATCH_SIZE` et `GAMMA`).

Pour l'environnement `CartPole-v1`, nous avons choisi un nombre d'épisodes élevé (`num_episodes=1000`) et un nombre de runs modéré (`num_runs=10`) afin d'assurer une convergence stable et d'obtenir des résultats fiables. La stratégie d'exploration-exploitation a été paramétrée avec un epsilon initial de `EPS_START=0.9`, une valeur minimale d'epsilon de `EPS_END=0.05`, et une décroissance progressive de `EPS_DECAY=1000`. Ces valeurs permettent à l'agent de privilégier l'exploration dans les premiers épisodes, tout en se concentrant davantage sur l'exploitation à mesure que l'entraînement progresse. Le mini-batch utilisé pour la mise à jour des Q-values est de taille `BATCH_SIZE=32`, avec un facteur de discount `GAMMA=0.99`, afin de favoriser les récompenses futures.

Pour l'environnement `Acrobot-v1`, qui est plus complexe, nous avons ajusté les hyperparamètres en conséquence. Étant donné que cet environnement nécessite plus d'exploration initiale pour accumuler de l'élan, nous avons choisi un epsilon initial plus élevé (`EPS_START=1.0`), avec une valeur minimale d'epsilon de `EPS_END=0.01` et une décroissance plus rapide (`EPS_DECAY=500`). Le nombre d'épisodes a été réduit à `num_episodes=100` et le nombre de runs à `num_runs=3` pour limiter la complexité computationnelle. Le mini-batch est de taille `BATCH_SIZE=64`, avec un facteur de discount `GAMMA=0.99` similaire à celui utilisé pour `CartPole-v1`.

4.1.2 Résultats avec le NN

4.1.2.1 CartPole-v1

Après avoir implémenté les algorithmes **DQN** (Deep Q-Network) et **DDQN** (Double Deep Q-Network), nous avons cherché à évaluer leurs performances en exécutant l'entraînement **10 fois** avec des graines aléatoires différentes.

Dans un premier temps, l'apprentissage s'est révélé instable, avec des scores oscillant majoritairement entre **20** et **40** points, et quelques pics atteignant jusqu'à **160** points. Cette forte variabilité suggère une sensibilité aux hyperparamètres et aux conditions d'exploration. En comparant nos résultats avec ceux d'autres implémentations disponibles sur GitHub, nous avons identifié des améliorations potentielles, notamment en ajustant la stratégie d'exploration/exploitation et en évitant une mise à jour classique du *epsilon threshold*. Après ces ajustements, les performances se sont nettement améliorées.

L'analyse des scores montre une tendance à la convergence en fin d'entraînement, bien que des fluctuations persistent. Cette variabilité, mise en évidence par l'**IQR** relativement large, indique que l'apprentissage reste influencé par les conditions d'entraînement spécifiques.

Concernant l'évolution des **Q-values**, nous observons que les valeurs estimées augmentent rapidement pour les deux algorithmes. Toutefois, la différence entre DQN et DDQN reste *faiblement marquée* sur cet environnement. Vers la fin de l'entraînement, la courbe des Q-values du **DDQN** se situe légèrement en dessous de celle du **DQN**, ce qui suggère une correction partielle de la surestimation, mais sans impact significatif sur les performances globales.

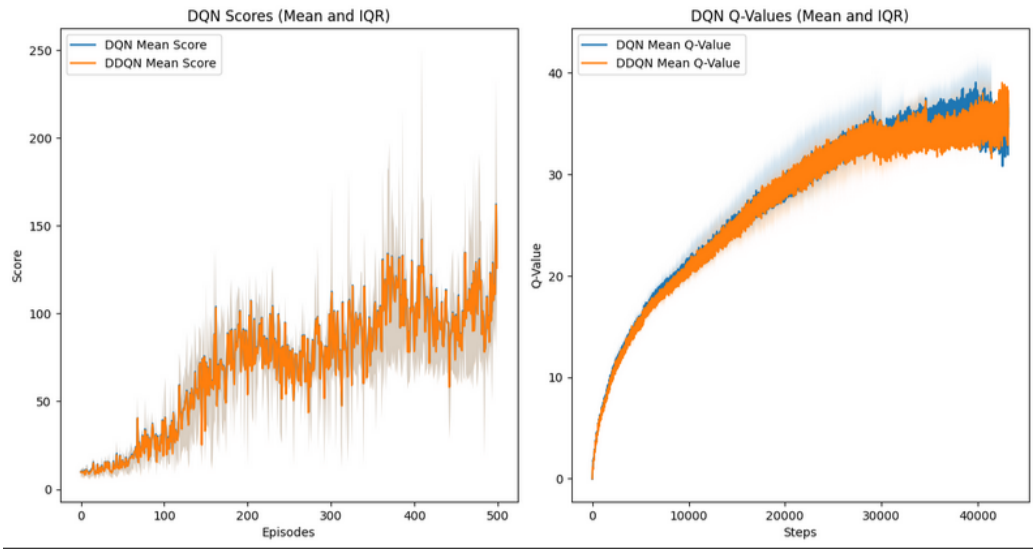


FIGURE 4.4 – Comparaison des performances de DQN et DDQN sur l'environnement `CartPole-v1`. Le graphique de gauche montre les scores obtenus au fil des épisodes pour **10 lancements aléatoires**. Les courbes représentent les **médianes**, tandis que les zones ombrées indiquent les **plages interquartiles** comprises entre le 25^{e} et le 75^{e} percentile. Le graphique de droite présente les *Q-values* estimées par les deux algorithmes au cours de l'apprentissage.

4.1.2.2 Acrobot-v1

L'entraînement des algorithmes **DQN** et **DDQN** sur l'environnement `Acrobot-v1` a été effectué sur **6 lancements aléatoires**. Contrairement à `CartPole-v1`, où l'agent cherche à maintenir un équilibre, `Acrobot-v1` est un problème plus complexe, nécessitant une séquence d'actions coordonnées pour atteindre un état cible.

L'évolution des scores montre une amélioration progressive au fil des épisodes, avec une forte variabilité en début d'apprentissage, comme le suggèrent les **plages interquartiles** relativement larges. Cette variabilité se réduit progressivement, indiquant une convergence plus stable en fin d'entraînement.

Concernant les **Q-values**, nous observons une diminution progressive au fil des étapes d'apprentissage, ce qui est cohérent avec l'objectif de l'environnement, où les récompenses sont négatives à chaque étape jusqu'à atteindre l'état terminal. Cependant, la différence entre DQN et DDQN est **très peu marquée**, ce qui suggère que la surestimation des Q-values n'a pas un impact significatif sur cet environnement. Contrairement à d'autres environnements où DDQN apporte une correction notable, ici, les courbes restent très proches, indiquant que l'effet de la double mise à jour de Q n'apporte pas d'amélioration évidente.

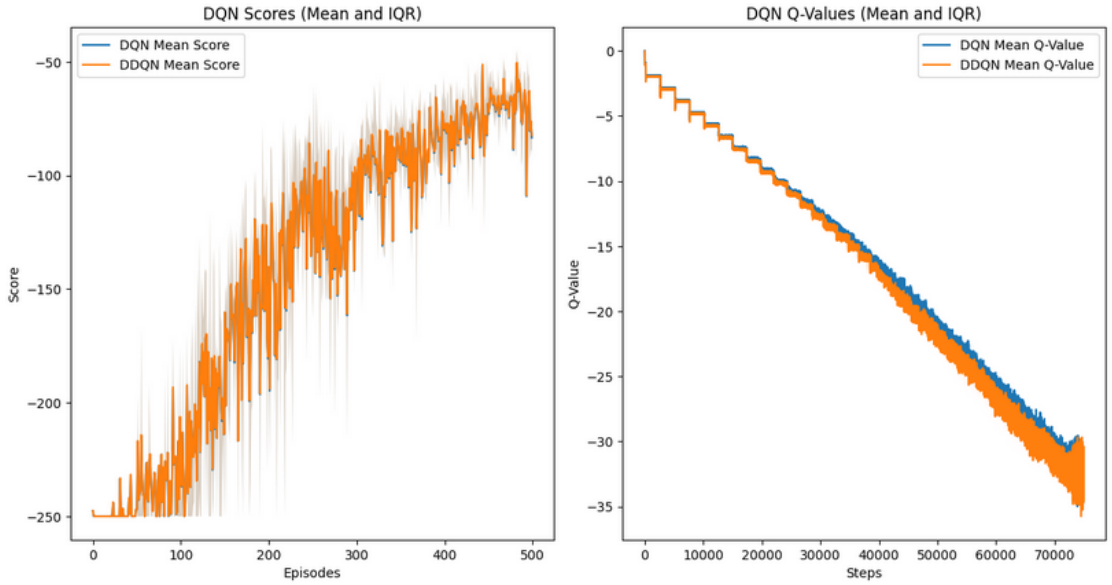


FIGURE 4.5 – Comparaison des performances de DQN et DDQN sur l'environnement `Acrobot-v1`. Le graphique de gauche montre les scores obtenus au fil des épisodes pour **6 lancements aléatoires**, tandis que le graphique de droite présente les *Q-values* estimées. Les zones ombrées indiquent les **plages interquartiles** comprises entre le 25^{e} et le 75^{e} percentile.

Enfin, ces résultats confirment que l'architecture *fully connected NN* ne semble pas idéale pour ce type d'environnement nécessitant des décisions plus stratégiques. Comme pour `CartPole-v1`, nous avons observé une forte sensibilité aux hyperparamètres et aux stratégies d'exploration. Cette observation nous a conduits à tester des architectures plus adaptées, notamment avec des réseaux convolutifs (**CNN**) sur les jeux Atari, afin d'obtenir des résultats plus significatifs sur des environnements à représentations visuelles plus riches.

4.2 Analyse des performances du CNN sur les jeux `Zaxxon-v5`, `Alien` et `SpaceInvaders-v5`

Afin d'évaluer les performances des algorithmes **DQN** (Deep Q-Network) et **DDQN** (Double Deep Q-Network) sur des environnements plus complexes implémentés avec un **CNN**, nous avons testé trois jeux issus de la bibliothèque `Atari` : **Space Invaders**, **Zaxxon** et **Alien**. Pour chaque environnement, nous avons exécuté les algorithmes sur **150 parties différentes**, en utilisant les paramètres suivants : un `EPS_START` de 1, un `EPS_END` de 0.05 et un `EPS_DECAY` d'environ 10 000, avec une taille de batch (`BATCH_SIZE`) fixée à 64. Les valeurs de `TAU` et `GAMMA` ont été conservées identiques aux expérimentations précédentes.

Nous avons ensuite analysé l'évolution des **Q-values** en échelle logarithmique afin d'observer les différences à chaque étape d'apprentissage et d'identifier la correction des surestimations par **DDQN**. Par ailleurs, les **scores** obtenus au fil des épisodes ont été tracés en échelle normale pour visualiser la progression des performances des agents au cours du temps.

Concernant l'analyse des résultats, nous avons remarqué que **DQN surestime massivement les valeurs Q**, comme en témoigne la courbe bleue qui croît de manière exagérée en fin d'apprentissage, que ce soit pour

les jeux `Zaxxon`, `Space Invaders` ou `Alien`. Cette tendance illustre bien le fait que **DDQN réduit les fluctuations et stabilise les scores en corrigeant la surestimation des valeurs Q**.

En ce qui concerne les scores et les récompenses obtenues, nous avons observé des dynamiques distinctes selon les jeux :

— **Alien** : Les scores augmentent progressivement au fil des training steps, avec une amélioration marquée vers **1200-1400 steps**. L'apprentissage est relativement linéaire, ce qui suggère une dynamique plus stable où la correction des Q-values par DDQNa un effet modéré.

— **Zaxxon** : Contrairement à `Alien`, les scores restent faibles sur une longue période avant une amélioration brutale autour de **1200-1400 steps**. Cette phase de stagnation suivie d'une progression soudaine montre que l'apprentissage a besoin d'un temps plus long pour converger vers de meilleures stratégies.

— **Space Invaders** : Nous observons une progression constante des performances des deux algorithmes en fonction des étapes d'entraînement. Les scores de DDQN et DQN suivent des trajectoires similaires au début, avec une légère supériorité de DQN dans certaines phases d'entraînement. Cependant, en fin d'apprentissage, DDQN montre une progression plus stable, tandis que DQN connaît des augmentations soudaines suivies de fluctuations importantes.

Ces résultats confirment que **DDQN corrige la surestimation des Q-values et offre une meilleure stabilité d'apprentissage**, notamment dans les environnements où DQN souffre d'une divergence progressive. Cependant, la correction apportée par DDQN est plus perceptible dans les jeux où les variations de récompenses sont plus marquées, comme `Zaxxon` et `Space Invaders`, plutôt que dans des environnements plus linéaires comme `Alien`.

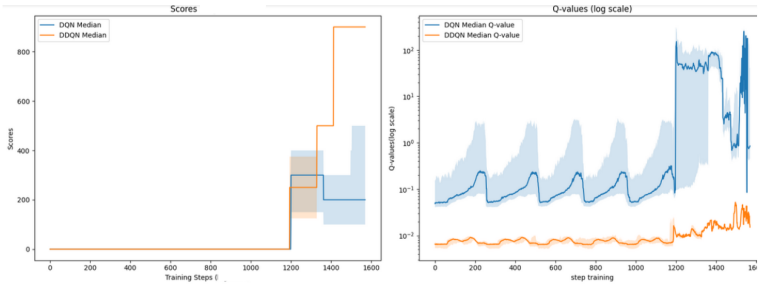


FIGURE 4.6 – `Zaxxon`

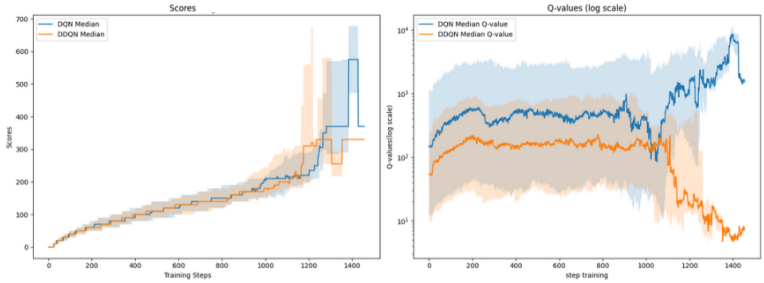


FIGURE 4.7 – `Alien`

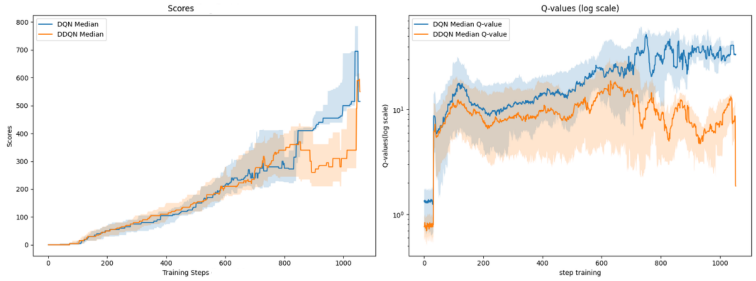


FIGURE 4.8 – `spaceInvade`

FIGURE 4.9 – Illustration des environnements testés : `Zaxxon` , `Alien` et `spaceInvade`.

5 Conclusion

Dans le cadre de l'analyse de l'article *Deep Reinforcement Learning with Double Q-Learning* de Hasselt *et al.*, nous avons cherché à comprendre sa contribution majeure dans le domaine de l'apprentissage par renforcement. Pour ce faire, nous avons d'abord étudié les travaux préalables de Hasselt publiés en 2010, afin de mieux appréhender le problème des surestimations des valeurs d'action observé dans les algorithmes de Q-Learning. Ces surestimations surviennent lorsque la même fonction Q est utilisée à la fois pour la sélection et l'évaluation des actions, conduisant à un biais positif dans l'estimation des récompenses futures.

Quatre ans plus tard, Mnih *et al.* ont proposé une avancée significative en utilisant un réseau de neurones simple pour approximer la fonction Q dans l'algorithme **DQN** (Deep Q-Network), qu'ils ont testé sur plusieurs jeux Atari. Cependant, ils ont également constaté que cet algorithme souffrait de surestimations similaires, entraînant des performances sous-optimales dans certains environnements.

Pour remédier à ce problème, Hasselt *et al.* ont proposé une amélioration majeure en introduisant l'algorithme **DDQN** (Double Deep Q-Network), qui sépare le processus de sélection et d'évaluation des actions en utilisant deux réseaux de neurones distincts. Cette méthode a été testée sur six jeux Atari 2600, montrant des résultats significativement meilleurs en termes de scores et de stabilité des valeurs Q.

Afin de tester ces deux algorithmes, nous avons implémenté l'algorithme **DQN** avec un réseau de neurone simple , que nous avons entraîné sur les jeux `CartPole-v1` et `Acrobot-v1`. Nous avons également implémenté l'algorithme **DDQN**, qui divise les réseaux pour la sélection des actions et l'évaluation des valeurs Q associées.

Nous avons ensuite implémenté un CNN pour l'algorithme DDQN et DQN et l'avons appliqué sur trois jeux Atari : *Zaxxon*, *Space Invaders* et *Alien*. Avec ces trois jeux, nous avons observé que l'algorithme DDQN corrige efficacement les surestimations observées avec DQN, comme en témoigne la présence d'un décalage significatif entre les courbes des deux algorithmes. Concernant les scores, les deux algorithmes montrent une tendance à maximiser leurs performances vers la fin de l'entraînement, illustrant leur bon fonctionnement.

En conclusion, notre analyse confirme que l'approche DDQN constitue une avancée notable en apprentissage par renforcement, notamment dans les environnements riches et dynamiques. Les perspectives futures incluent l'optimisation des architectures CNN, l'ajustement des hyperparamètres pour des environnements plus variés, et l'exploration d'algorithmes hybrides combinant DDQN avec d'autres techniques de RL avancées.

A Annexes

A.1 Algorithme de Q-learning

Algorithm 1 Q-learning [4]

Initialisation : pour chaque état s et action a, initialiser $Q(s, a)$ arbitrairement
for épisode = 1 à M **do**
  Choisir un état initial s
  while s n'est pas terminal **do**
    Choisir une action a dans s selon une politique ϵ-greedy tirée de Q
    Exécuter l'action a, observer la récompense r et le nouvel état s'
    Mettre à jour :

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\Big]$$

    Assigner $s \leftarrow s'$
  end while
end for

A.2 Algorithme Double Q-learning

Algorithm 2 Double Q-learning [1]

Initialiser Q^A, Q^B, s
repeat
  Choisir une action a en se basant sur $Q^A(s,\cdot)$ et $Q^B(s,\cdot)$ (par exemple, de façon ϵ-gloutonne), puis observer r et s'
  Choisir (par exemple aléatoirement) de mettre à jour soit UpdateA, soit UpdateB
  if UpdateA **then**
    $a^* \leftarrow \arg\max_a Q^A(s', a)$
    $Q^A(s,a) \leftarrow Q^A(s,a) \; + \; \alpha\Big[r \; + \; \gamma\, Q^B\big(s', a^*\big) \; - \; Q^A(s,a)\Big]$
  else
    $b^* \leftarrow \arg\max_a Q^B(s', a)$
    $Q^B(s,a) \leftarrow Q^B(s,a) \; + \; \alpha\Big[r \; + \; \gamma\, Q^A\big(s', b^*\big) \; - \; Q^B(s,a)\Big]$
  end if
  $s \leftarrow s'$
until *fin de l'épisode*

B Annexes

B.1 Autres résultats pour CartPole

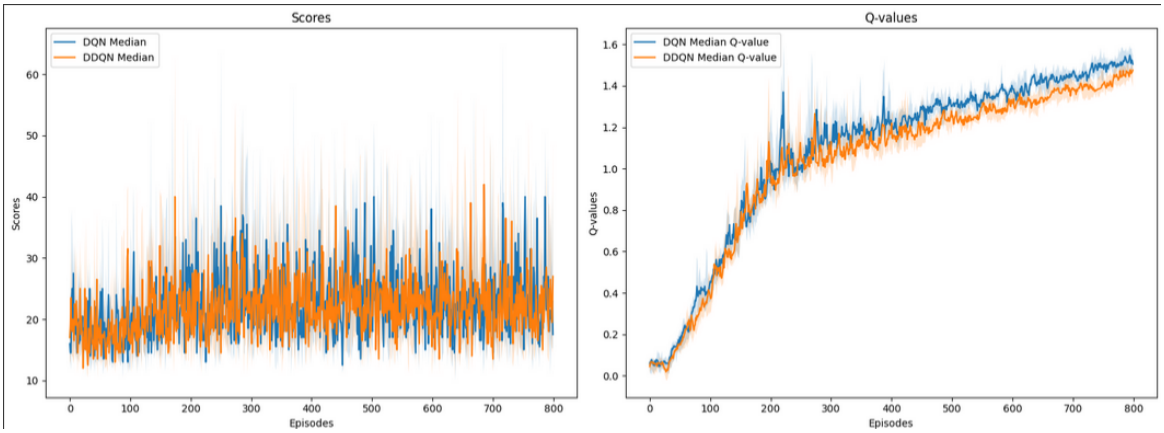


FIGURE B.1 – Résultats pour `num_episodes=2000`, `num_runs=10`, `BATCH_SIZE=32`

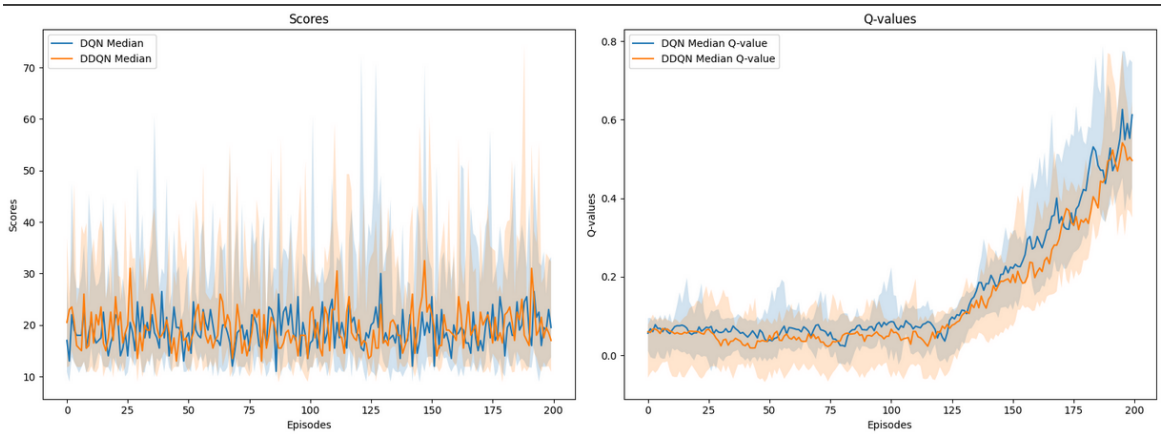


FIGURE B.2 – Résultats pour `num_episodes=2000`, `num_runs=10`, `BATCH_SIZE=128`

FIGURE B.3 – Comparaison des performances de DQN et DDQN sur l'environnement CartPole. Le graphique de gauche montre les scores obtenus au fil des épisodes pour 10 lancements aléatoires sur 800 épisodes (graphes en haut) et 200 épisodes (graphes en bas). Le graphique de droite présente les Q-values estimées. Les zones ombrées indiquent les plages interquartiles comprises entre le 10e et le 90e percentile.

B.2 Autres résultats pour Acrobot

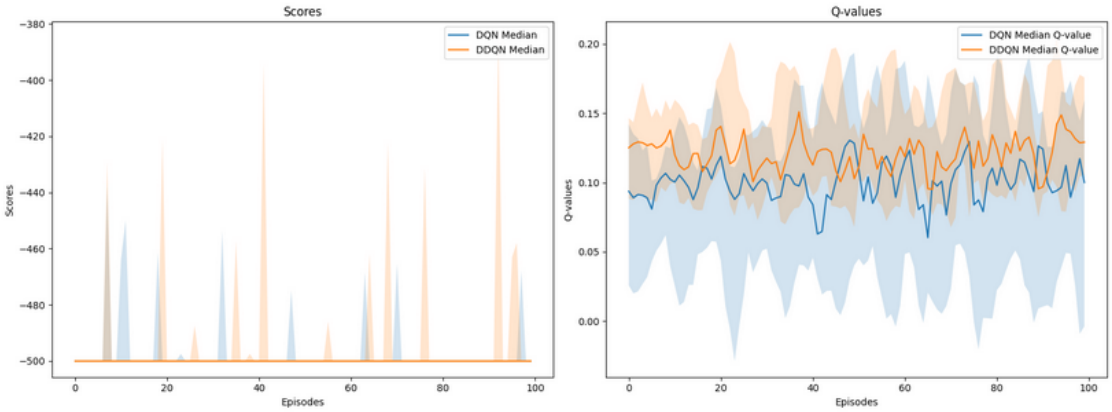


FIGURE B.4 – Résultats pour `num_runs=6`, `EPS_DECAY=500`, `GAMMA=0.99`, `TAU=0.005`

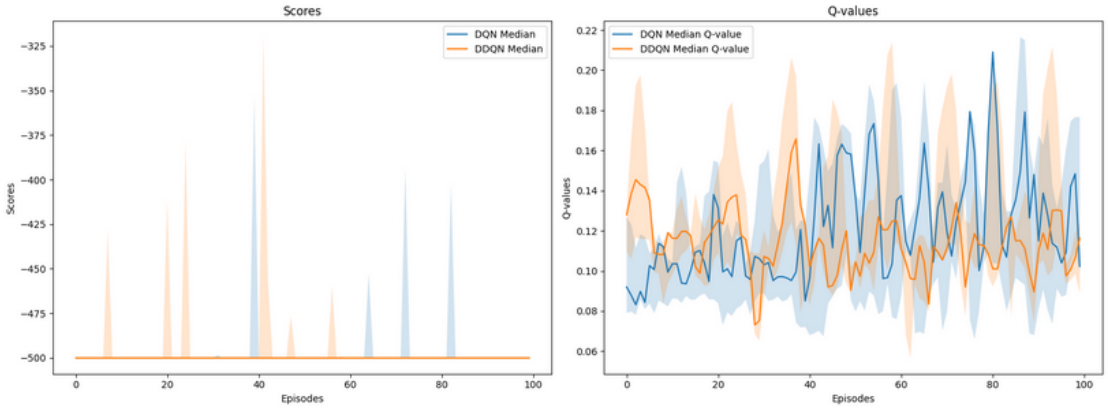


FIGURE B.5 – Résultats pour `num_runs=3`, `EPS_DECAY=10000`, `GAMMA=0.98`, `TAU=0.001`

FIGURE B.6 – Comparaison des performances de DQN et DDQN sur l'environnement Acrobot-v1. Le graphique de gauche montre les scores obtenus au fil des épisodes pour 3 lancements aléatoires sur 100 épisodes, tandis que le graphique de droite présente les Q-values estimées. Les zones ombrées indiquent les plages interquartiles comprises entre le 10e et le 90e percentile.

Bibliographie

[1] Hado van Hasselt. "Double Q-Learning". In : *Advances in Neural Information Processing Systems (NeurIPS)*. 2010, p. 2613-2621.

[2] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In : *Nature* 518.7540 (2015), p. 529-533. doi : 10.1038/nature14236.

[3] Hado van Hasselt, Arthur Guez et David Silver. "Deep reinforcement learning with double Q-learning". In : *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI'16)*. 2016, p. 2094-2100.

[4] Christopher John Cornish Hellaby Watkins et Peter Dayan. "Q-learning". In : *Machine Learning* 8.3-4 (1992), p. 279-292.