{ T-AIA-902-TLS_10 }

Fiche de Synthèse

Corentin DEPRECQ,

Arnaud MOREAU,

Maxime DELMON,

Rémi PERRAY

I. Résumé

Dans le cadre de ce projet, nous avons exploré les fondamentaux de l'apprentissage par renforcement en développant des intelligences artificielles capables d'apprendre à jouer à des jeux simples fournis par la bibliothèque **Gymnasium**. L'objectif principal était de concevoir et d'implémenter des agents intelligents capables de prendre des décisions optimales dans un environnement donné, en se basant uniquement sur leurs interactions avec celui-ci.

Nous avons travaillé sur les environnements suivants :

- **FrozenLake** : un problème de navigation stochastique sur une grille glacée.
- **Taxi-v3** : un problème de logistique urbaine consistant à récupérer et déposer un passager.
- **CliffWalking** : un scénario de déplacement optimal avec risque de chute (pénalité élevée).
- Un jeu Atari au choix (Breakout, Pong, *Space Invaders*) pour appliquer une approche à grande échelle avec un réseau de neurones.

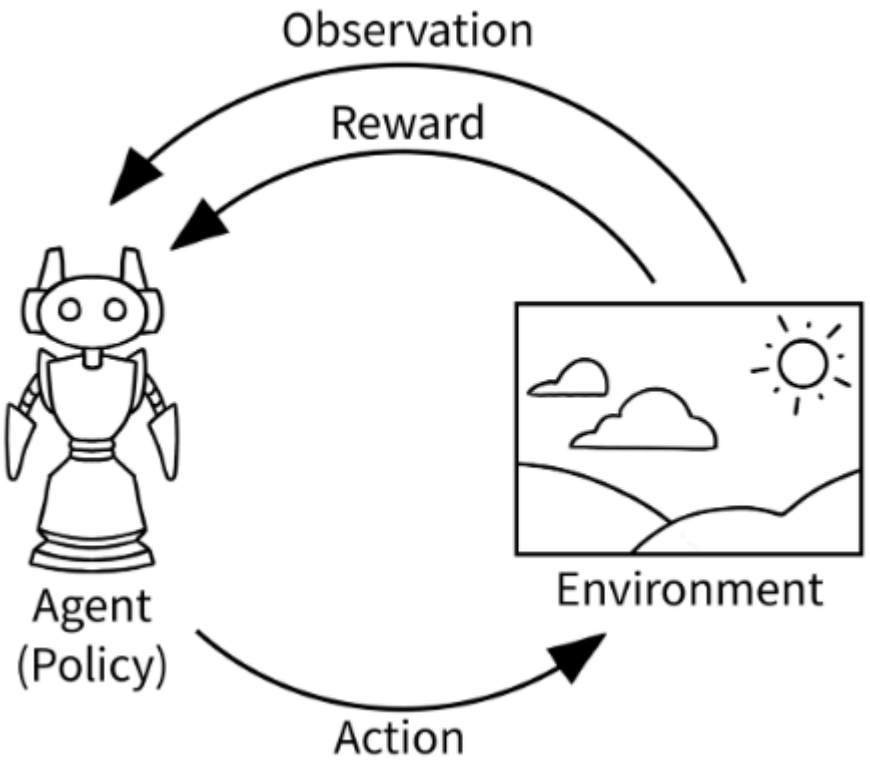
Nous avons testé différentes méthodes d'apprentissage, par exemple :

- Une méthode où l'IA essaie différentes actions et apprend lesquelles donnent les meilleurs résultats,
- Une autre où elle ajuste ses choix en fonction de ce qui a bien marché ou non dans le passé,
- Et même une méthode plus avancée où **un réseau de neurones**, une sorte de "cerveau artificiel", l'aide à prendre des décisions dans des jeux plus compliqués.

Petit à petit, notre IA est devenue meilleure, parfois en commettant beaucoup d'erreurs au début, mais en finissant par adopter les bons réflexes pour réussir le jeu.

II. Apprentissage par renforcement

L'apprentissage par renforcement est une méthode d'intelligence artificielle inspirée du comportement humain. L'idée est simple : une IA (appelée "agent") agit dans un environnement, reçoit des récompenses en fonction de ses actions, et apprend progressivement à prendre les meilleures décisions pour maximiser ces récompenses.



L'agent n'a pas de plan préétabli. Il doit expérimenter pour découvrir ce qui fonctionne, un peu comme un enfant qui apprend à faire du vélo en tombant plusieurs fois avant de trouver l'équilibre.

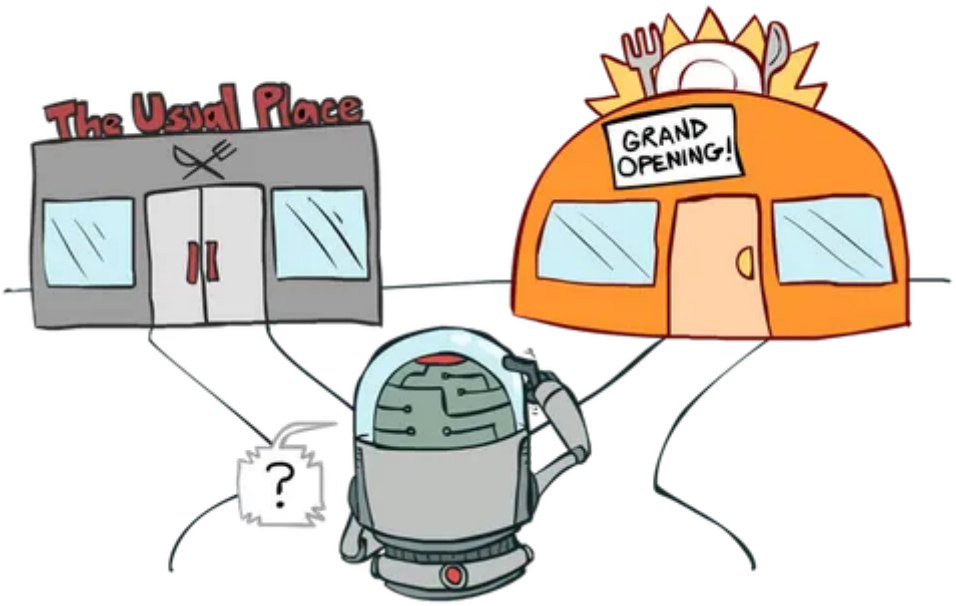
🎯 Exploration vs Exploitation – La stratégie Epsilon-Greedy

Un des grands défis de cet apprentissage est de savoir quand essayer de nouvelles choses (exploration) et quand réutiliser ce qu'on a appris (exploitation).

La stratégie epsilon-greedy sert à ça :

- Avec une petite probabilité (ε, ou "epsilon"), l'agent choisit une action au hasard pour découvrir de nouvelles possibilités.

- Le reste du temps, il choisit l'action qu'il pense être la meilleure, selon ce qu'il a appris jusque-là.



Ce mécanisme permet de trouver un bon équilibre entre apprentissage et performance.

⚙️ Les paramètres essentiels : Gamma (γ) et Alpha (α)

Deux paramètres jouent un rôle clé dans l'apprentissage de l'agent :

- **Gamma (γ)** : c'est le **facteur d'actualisation**. Il indique à quel point l'agent doit prendre en compte les **récompenses futures**.
  → Si gamma est proche de 1, l'agent pense **à long terme**.

→ S'il est proche de 0, il privilégie les **gains immédiats**.

- **Alpha (α)** : c'est le **taux d'apprentissage**. Il contrôle à quelle vitesse l'agent **met à jour ses connaissances** après chaque expérience.
  → Un alpha élevé = apprentissage rapide mais instable.
  → Un alpha bas = apprentissage plus lent, mais plus stable.

👉 **À insérer ici : un graphique illustrant l'effet de gamma et alpha sur l'évolution de l'apprentissage**

🤖 Les algorithmes classiques : Q-learning et SARSA

🔹 Q-learning

Q-learning est un algorithme hors-politique, c'est-à-dire qu'il apprend la meilleure action possible, même si cette action n'est pas celle choisie pendant l'apprentissage. À chaque étape, il met à jour ce qu'on appelle la valeur Q (qualité d'une action dans une situation donnée) à l'aide d'une formule mathématique basée sur :

- la récompense reçue,
- la meilleure action possible dans l'état suivant.

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

🔸 SARSA

SARSA est un algorithme en ligne (on-policy) : il met à jour ses valeurs en fonction des actions réellement choisies, pas celles théoriquement optimales.

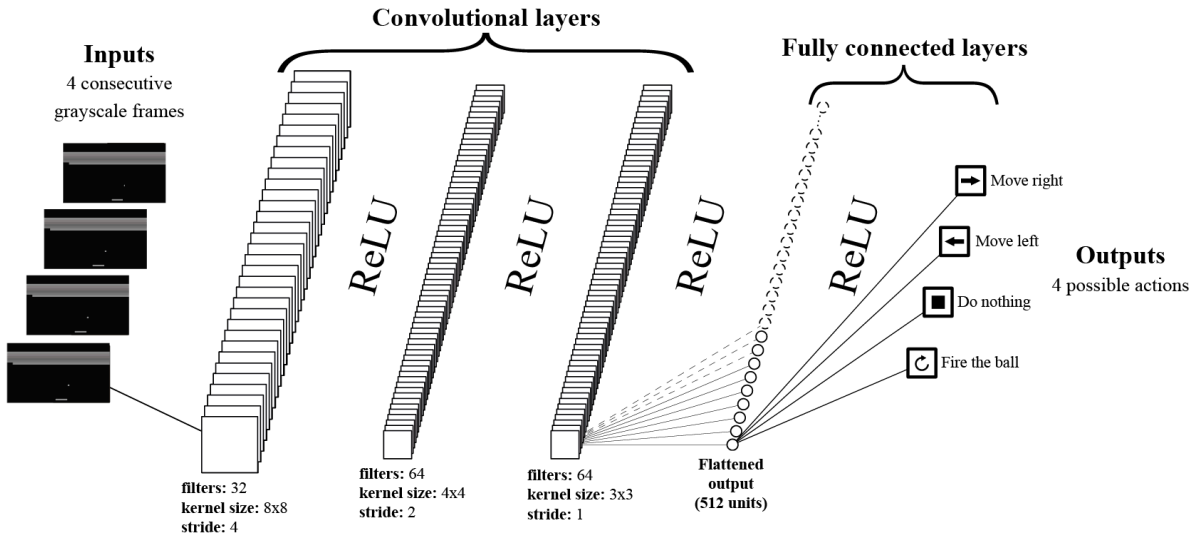
$$Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma\, Q(s',a') - Q(s,a)\big].$$

Ces deux algorithmes permettent à l'agent d'apprendre sans utiliser de réseaux de neurones, mais sont limités à des environnements simples.

🧠 Quand les environnements deviennent complexes : le DQN

Pour les jeux plus complexes, comme ceux d'Atari, les simples tableaux de valeurs ne suffisent plus. On utilise alors un réseau de neurones pour estimer les valeurs Q. C'est ce qu'on appelle un Deep Q-Network (DQN).

Le DQN remplace le tableau par un réseau de neurones profond, qui prend en entrée la situation actuelle (comme une image du jeu) et donne en sortie les valeurs estimées pour chaque action possible.



Composition de notre simple DQN

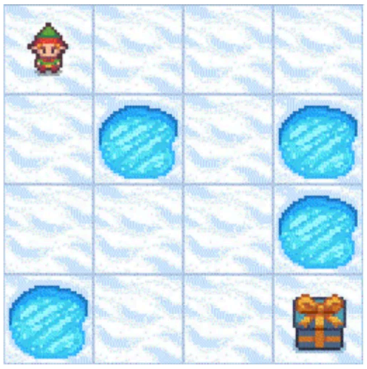
Le DQN permet de gérer des environnements beaucoup plus riches visuellement, mais demande aussi plus de données, de puissance de calcul et de temps d'entraînement.

III. Démarche

Pour cela, nous avons commencé par découvrir les bases de l'apprentissage par renforcement, une méthode où l'IA apprend en essayant différentes actions et en observant les conséquences. Nous avons aussi étudié la stratégie d'exploration et d'exploitation (appelée *epsilon-greedy*), qui aide l'IA à choisir entre tester de nouvelles options ou utiliser ce qu'elle sait déjà. Enfin, nous avons mis en pratique plusieurs algorithmes d'apprentissage, comme Q-learning et SARSA, qui permettent à l'IA d'améliorer ses décisions au fil du temps.

IV. Les Environnements de test

a. Frozenlake



Présentation du jeu FrozenLake

FrozenLake est un environnement classique d'apprentissage par renforcement fourni par la bibliothèque gymnasium. Le jeu se déroule sur une grille représentant un lac gelé, avec différents types de cases :

- S : point de départ,
- F : surface gelée sûre (solide),
- H : trou dans la glace (si l'agent tombe, il échoue),
- G : objectif à atteindre.

L'agent peut se déplacer dans les 4 directions (gauche, bas, droite, haut). L'objectif est d'atteindre G à partir de S, tout en évitant les H.

Objectif de l'apprentissage

L'objectif est d'apprendre **une politique optimale**, c'est-à-dire une stratégie permettant à l'agent de maximiser ses chances d'atteindre l'objectif tout en minimisant les risques de tomber dans un trou.

Q-learning : principe général

Le Q-learning est une méthode d'apprentissage par renforcement **hors-ligne (off-policy)**.
Il consiste à construire une **Q-table** (matrice Q(s, a)), qui estime la valeur (la qualité) d'effectuer une action a depuis un état s.

À chaque épisode de jeu, l'agent :

1. Explore l'environnement,
2. Met à jour sa Q-table selon la règle :

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

Avec :

- α (alpha) : taux d'apprentissage,
- γ (gamma) : facteur d'actualisation (importance du futur),
- r : récompense reçue,
- s' : état suivant,
- a' : action suivante optimale.

L'exploration est gérée via une stratégie **ε-greedy** : l'agent choisit parfois une action aléatoire (exploration), et le reste du temps, l'action qui maximise la Q-table (exploitation).

Ce que fait le code

Ce code met en œuvre un agent Q-learning entraîné sur FrozenLake, avec une interface de visualisation interactive. Il se compose de plusieurs éléments clés :

🔨 Entraînement (train_agent)

- L'agent apprend à partir de plusieurs épisodes de jeu.
- Il utilise une **récompense modifiée (shaping)** :
    - Récompense bonus si l'agent réussit,
    - Pénalité en cas d'échec,
    - Pénalité légère à chaque pas pour encourager des chemins courts.
- Le taux d'exploration ε diminue au fil du temps selon une **décroissance exponentielle**.

📊 Visualisation dynamique

Pendant l'apprentissage, plusieurs graphiques sont mis à jour régulièrement :

- Dernière frame du jeu (vue visuelle),
- Évolution du taux ε,
- Carte de la meilleure action pour chaque cellule (via des flèches),
- Moyenne glissante du nombre d'actions par épisode,
- Somme glissante des récompenses obtenues.

🤖 Simulation finale

Une fois l'entraînement terminé :

- L'agent joue un épisode en **mode exploitation pure** (aucune exploration),
- Un visuel final est enregistré, combinant :
    - Vue de la dernière situation de jeu,
    - Carte de la Q-table avec les meilleures actions affichées.

📁 Fonctionnalités avancées du code

- Prise en charge de **cartes de tailles personnalisées** (ex : 6x6, 10x10),
- Génération de **cartes aléatoires**,

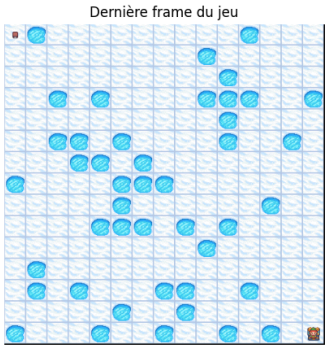
- Visualisation en temps réel avec matplotlib et seaborn,
- Code modulaire et clair, facile à adapter pour d'autres environnements.

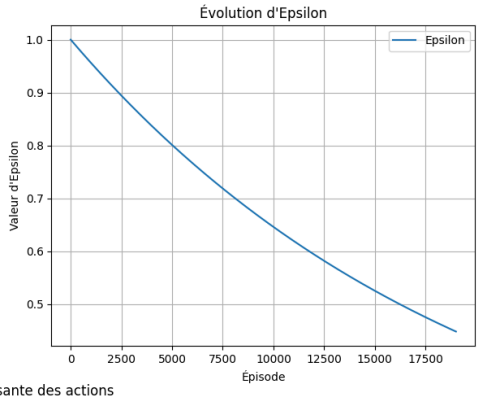
✅ Conclusion

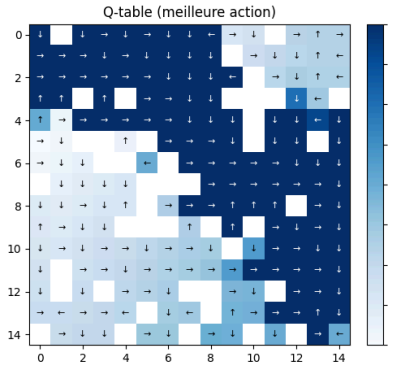
Ce projet constitue une application concrète et visuelle du Q-learning, adaptée à un environnement simple mais non trivial. Il met en lumière :

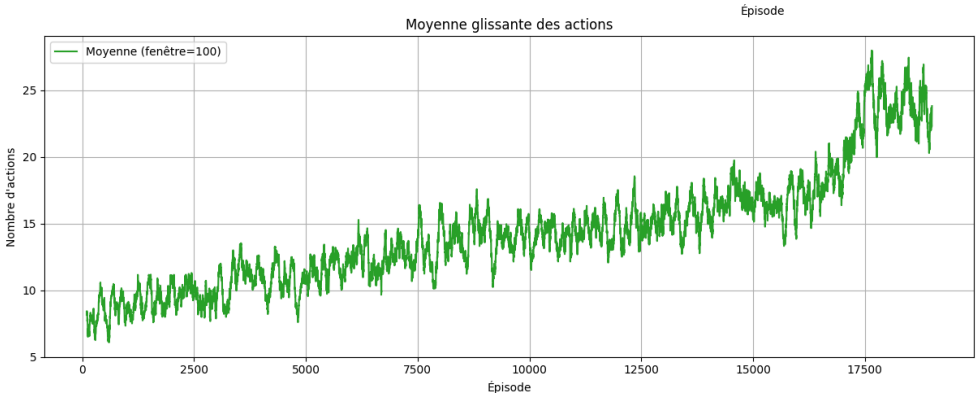
- Le rôle de la Q-table dans la prise de décision,
- L'intérêt du shaping des récompenses,
- L'importance de l'équilibre entre exploration et exploitation,
- L'utilité des visualisations pour analyser l'apprentissage.

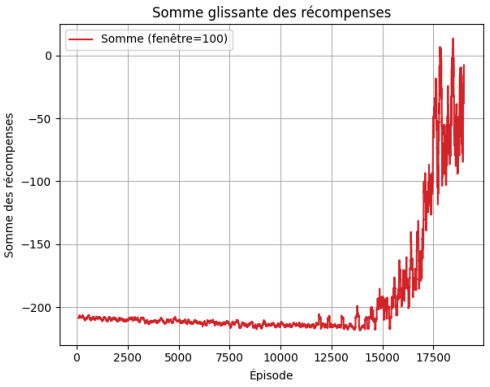
Ce type d'approche est un excellent point de départ pour expérimenter avec l'apprentissage par renforcement avant de passer à des environnements plus complexes.



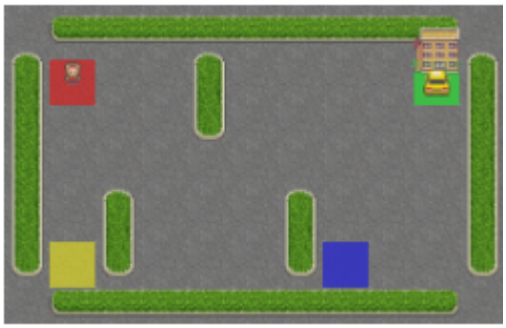








b. Taxi driver



Présentation du jeu Taxi

Taxi (alias Taxi-v3) est un environnement d'apprentissage par renforcement fourni par la bibliothèque Gymnasium. Le jeu se déroule sur une grille (5×5 par défaut) représentant une ville, avec différents éléments :

- 🚕 : position du taxi,
- 🚶 : position du passager,
- 📍 : destination du passager,
- 🚧 : obstacles (murs) délimitant les rues.

L'agent (le taxi) peut se déplacer dans les 4 directions (gauche, bas, droite, haut), ainsi que ramasser ("pick up") ou déposer ("dropoff") le passager.

Objectif de l'apprentissage

L'objectif est d'apprendre une politique optimale permettant au taxi :

1. De se rendre à la position du passager,
2. De prendre ("pickup") le passager,
3. De conduire jusqu'à la destination,
4. De déposer ("dropoff") le passager, tout en minimisant le nombre de pas et en évitant les collisions avec les obstacles.

Q-learning : principe général

Le Q-learning est une méthode d'apprentissage par renforcement hors-ligne (off-policy).
Il consiste à construire une Q-table Q(s, a) qui estime la « valeur » d'effectuer l'action a depuis l'état s.
À chaque épisode :

- L'agent explore l'environnement (états et récompenses),
- Met à jour sa Q-table selon :
  Q(s,a)←Q(s,a)+α[r+γmaxa′Q(s′,a′)−Q(s,a)]Q(s,a) \leftarrow Q(s,a) + \alpha \bigl[r + \gamma \max_{a'}Q(s',a') - Q(s,a)\bigr]Q(s,a)←Q(s,a)+α[r+γmaxa′Q(s′,a′)−Q(s,a)]
  Avec :
  - α : taux d'apprentissage,
  - γ : facteur d'actualisation (importance du futur),
  - r : récompense obtenue (bonus pour dropoff réussi, pénalité pour action invalide),
  - s′ : état suivant,
  - a′ : action optimale suivante.
    L'exploration est gérée via une stratégie ε-greedy : l'agent choisit aléatoirement une action avec probabilité ε (exploration), sinon l'action maximisant Q(s, a) (exploitation).

Ce que fait le code

Ce code met en œuvre un agent Q-learning entraîné sur Taxi-v3, avec visualisation interactive. Il comprend :

🔨 **Entraînement (train_agent)**

- L'agent joue sur plusieurs épisodes (ex : 10 000).
- Récompenses :
  - +20 pour un dropoff réussi,
  - −10 pour pickup/dropoff ratés (par ex. pickup hors de la case passager),
  - −1 à chaque pas pour encourager des trajets courts.
- ε diminue exponentiellement : $\varepsilon_t = \varepsilon_0 \times \exp(-\text{decay\_rate} \times t)$.

📊 Visualisation dynamique

Pendant l'apprentissage, plusieurs graphiques sont mis à jour :

- Dernière frame du jeu (vue du taxi et du passager),
- Évolution de ε,
- Carte des meilleures actions pour chaque état (flèches sur la grille),
- Moyenne glissante du nombre de pas par épisode,
- Somme glissante des récompenses cumulées.

🤖 Simulation finale

Après entraînement :

- L'agent roule en exploitation pure (ε = 0),
- On enregistre un visuel final combinant :
    - Vue de la partie (trajet complet),
    - Moyenne mobile des actions
    - Moyenne mobile des récompenses
    - l'évolution de la courbe d'epsilon-greedy

📁 Fonctionnalités avancées du code

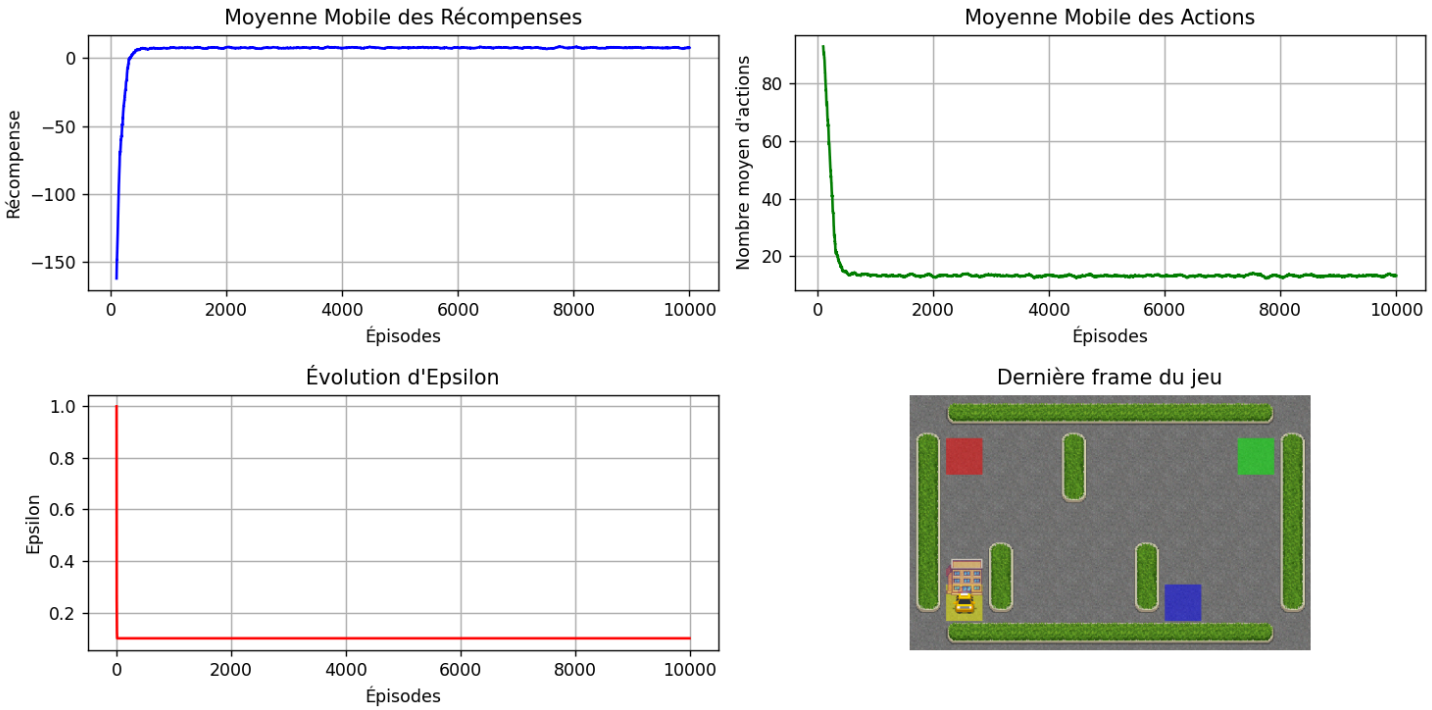
- Support de cartes personnalisées (modifier la taille),
- Génération aléatoire de positions passager/destination,
- Animation en temps réel avec Matplotlib,
- Code modulaire (séparation claire des fonctions d'entraînement, de visualisation et d'évaluation).

✅ Conclusion

Ce projet illustre de manière concrète le Q-learning sur un environnement simple mais riche en interactions. Il met en avant :

- Le rôle central de la Q-table dans la prise de décision,
- L'importance du shaping des récompenses pour guider l'agent,
- L'équilibre exploration/exploitation via ε-greedy,
- L'utilité des visualisations pour diagnostiquer et comprendre l'apprentissage.

Cette approche constitue une excellente base pour explorer des environnements plus complexes (Taxi-large maps, apprentissage par réseaux de neurones, politique d'exploration avancée, etc.).



c. CliffWalking

Présentation du jeu Cliff Walking

Cliff Walking est un environnement Toy Text de Gymnasium, inspiré de l'exemple 6.6 de Sutton & Barto:

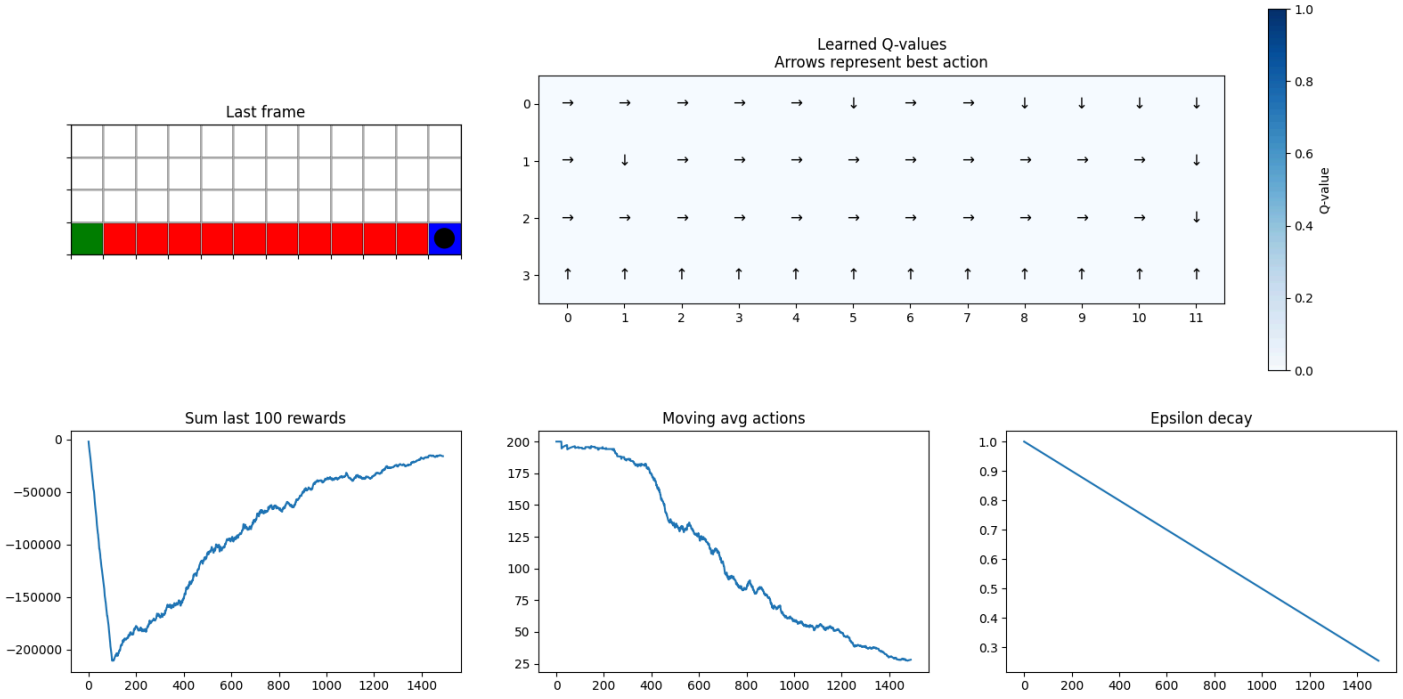
- Grille de **4×12** cases.
- L'agent commence en **[3, 0]** (coin inférieur gauche).
- L'objectif se trouve en **[3, 11]** (coin inférieur droit).
- Une falaise couvre **[3, 1..10]** : si l'agent y entre, il reçoit une grosse pénalité et est renvoyé au départ.
- Par défaut, l'environnement est déterministe ; une option *slippery* introduit de l'aléa dans les déplacements.
- **Reward** : −1 à chaque déplacement standard, −100 si l'agent tombe dans la falaise.
- L'espace d'action est Discrete(4) : haut, droite, bas, gauche.
- L'espace d'observation est Discrete(48), correspondant aux positions valides excluant falaise et bu.
- Un épisode se termine soit par l'atteinte du but, soit par une chute dans la falaise (suivant configuration) .

Comparatif Q-learning vs SARSA (dans Cliff Walking)

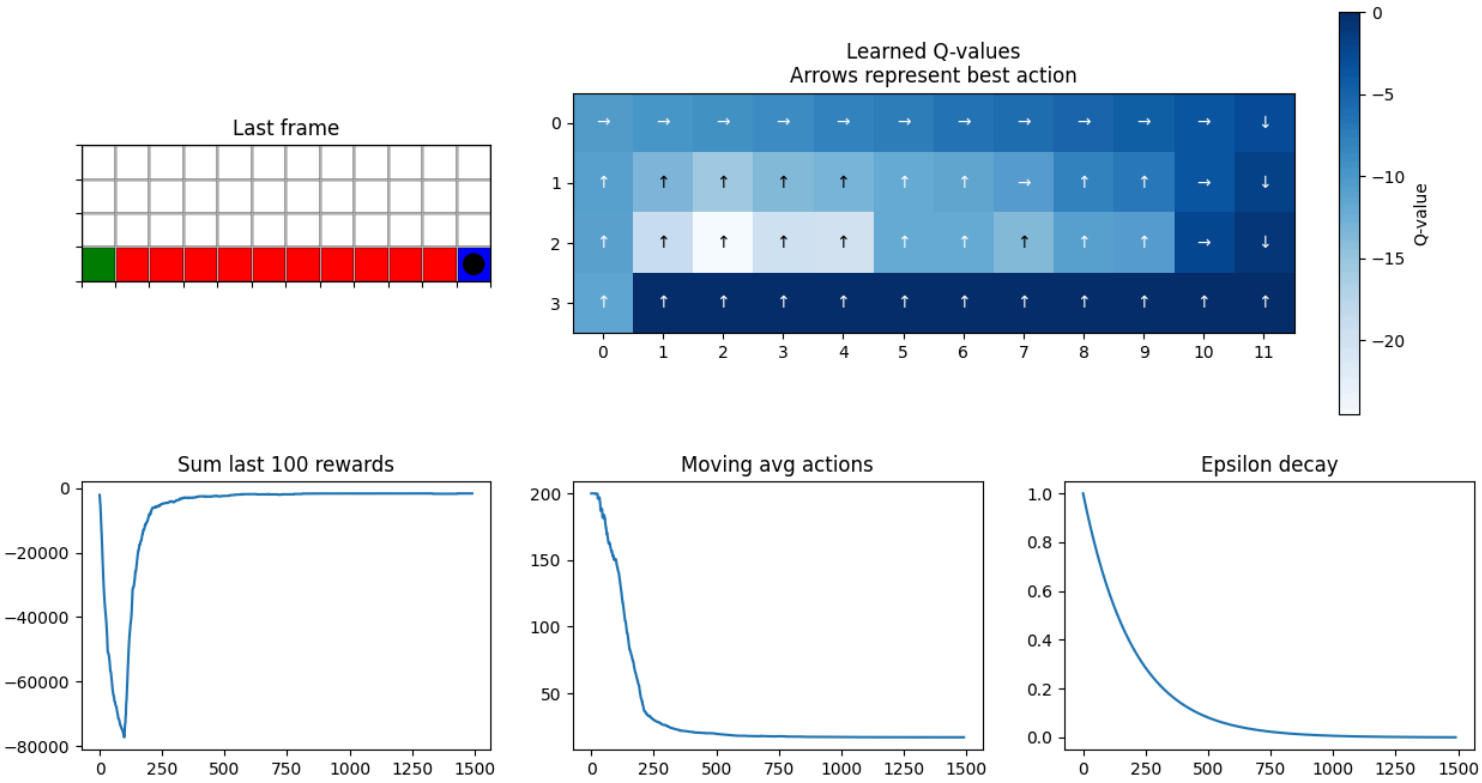
Un papier récent compare ces deux méthodes sur cet environnement classique :

- SARSA (on-policy) apprend une politique plus conservatrice : elle évite la falaise, préfère un chemin indirect et plus sûr.
- Q-learning (off-policy) apprend une politique plus optimiste : elle privilégie le chemin le plus direct au ras de la falaise, acceptant les risques pour maximiser la récompense cumulé.
- Les courbes de récompense montrent que SARSA est plus stable, mais ramène moins de gain par épisode.
- Q-learning atteint des récompenses plus élevées à terme, mais avec des fluctuations dues aux chutes occasionnelles.

Q-learning :



SARSA :



Ce que ton code peut inclure

🔨 Entraînement des agents

- SARSA et Q-learning s'entraînent sur plusieurs milliers d'épisodes avec stratégie ε-greedy identique (α, γ, ε initiaux mêmes valeurs, décroissance similaire).
- Récompenser/ pénaliser comme dans l'environnement : −1 pour un mouvement, −100 pour la falaise, etc.

📊 Visualisation comparative

- Deux courbes superposées des récompenses cumulées au fil des épisodes.
- Cartes politiques finales indiquant les chemins empruntés (flèches) pour chaque méthode.
- Histogramme ou graphique du nombre de chutes dans la falaise par méthode.

🤖 Simulation finale

- Exécuter les deux agents en exploitation pure (ε = 0).

- Enregistrer la trajectoire : nombre d'étapes, chutes s'il y en a (ce qui ne devrait pas arriver en exploitation), rapidité.

✅ **Résumé des différences pratiques**

Algorithme	Politique enseignée	Bénéfice principal	Risque
Q-learning	Politique optimale hors-politique	Chemin direct, récompenses maximales	Chutes fréquentes pendant l'apprentissage
SARSA	Politique on-policy liée à ε-greedy	Chemin sûr, apprentissage stable	Moins de gain moyen, chemin plus long

- SARSA est recommandé si tu privilégies la sécurité durant le trajet.
- Q-learning est préféré pour maximiser la performance finale, même si l'agent prend des risques.

d. Jeux Atari (Pong)

1. Contexte et objectifs

Depuis nos travaux initiaux sur le Q-learning classique (FrozenLake, CliffWalking, TaxiDriver), où l'environnement est simple et de petite taille, nous avons souhaité tester la robustesse du Deep Q-Network (DQN) dans des environnements visuels complexes. Les jeux Atari offrent un challenge majeur : des observations brutes en pixels et une dynamique de jeu rapide. L'objectif ici est donc de décrire la mise en œuvre, les difficultés rencontrées et les performances obtenues sur trois titres emblématiques : Space Invaders, Pong et Breakout.

2. Infrastructure et protocole expérimental

Pour assurer une reproductibilité et une stabilité des résultats, nous avons utilisé un serveur Linux accessible en SSH, équipé d'une carte NVIDIA RTX 3070 et de 32 Go de RAM. Le code est développé en Python, avec Gymnasium (ALE) pour les environnements Atari et PyTorch pour l'implémentation des réseaux de neurones.

Les principales étapes de prétraitement sont :

1. **Conversion en niveaux de gris** pour réduire la dimensionnalité et uniformiser les entrées.
2. **Redimensionnement à 84×84 pixels**, conformément à l'article originel de DeepMind, afin de maintenir une granularité visuelle suffisante.
3. **FrameStack de 4** images consécutives pour capturer la dynamique (vitesse, direction) des objets en mouvement.
4. **Frameskip de 4**, appliqué initialement deux fois (bug corrigé ensuite), pour accélérer l'entraînement tout en préservant la cohérence temporelle.
5. **Clipping des récompenses** à ±1, afin de stabiliser l'apprentissage et limiter l'overflow des gradients.

L'architecture de base du DQN suit la recette de Mnih et al. :

- Trois couches convolutionnelles (8×8 stride 4 → 4×4 stride 2 → 3×3 stride 1), chacune suivie d'une ReLU.
- Une couche fully-connected de 512 neurones, activée par ReLU.
- Une couche de sortie linéaire dont la dimension égale le nombre d'actions disponibles dans le jeu.

Les hyperparamètres sélectionnés sont les suivants :

- Optimiseur RMSprop avec un learning rate de 2.5×10^{-4}, $\alpha=0.95$, $\varepsilon=1\times10^{-2}$.
- Discount factor $\gamma=0.99$.
- Politique ε-greedy décroissante linéairement de 1.0 à 0.1 sur les 4 premiers millions de step.
- Utilisation d'un burn-in replay buffer de 50 000 transitions avant la première mise à jour du réseau.

3. Résultats et analyses

3.1 Space Invaders

Dans un premier temps, Space Invaders a servi de plate-forme de test pour évaluer la capacité du DQN à apprendre à partir d'observations visuelles complexes. Les premières expériences ont donné un score moyen très faible (<5) même après 58 000 épisodes (*Figure 1*).

Après investigation, deux problèmes majeurs ont été identifiés :

- Le **frameskip appliqué deux fois** (dans ALE + wrapper manuel), entraînant une répétition de la même action sur 16 images consécutives et masquant les projectiles.
- Un **signal de récompense très bruité**, malgré le clipping à ±1, provoquant de fortes oscillations du loss MSE et du Q_mean.

Ces difficultés ont conduit à arrêter les expériences sur Space Invaders et à passer immédiatement à l'exploration de Pong pour itérer plus rapidement sur les paramètres.

3.2 Pong Pong (phase initiale)

Nous avons ensuite testé Pong pour comprendre la généralisation du DQN. Chaque match se joue en 21 points gagnants, ce qui se traduit par des épisodes très longs et une montée en score lente. Après plusieurs milliers d'épisodes, la progression restait trop faible pour itérer rapidement sur les paramètres. Nous avons donc décidé de basculer sur Breakout, plus court et offrant un feedback d'entraînement plus intense.

3.3 Breakout

3.3.1 Expérimentation initiale (bug frameskip)

Dans cette phase, le DQN vanilla tournait avec un frameskip appliqué deux fois, ce qui répétait chaque action sur 16 frames consécutives. Après 32 000 épisodes, le score moyen plafonnait autour de **20 points** (*Figure 2*). Plusieurs variantes ont été testées **avant** la correction du frameskip :

- **Raw DQN (vanilla *Figure 2*)**
- **Double DQN** (*Figure 3*)
- **Dueling DQN** (*Figure 4*)
- **Double + Dueling** (*Figure 5*)
- **PER lite** (*Figure 6*)

Aucune de ces extensions n'a apporté de gain significatif : toutes les courbes Score, Loss et Q_mean restaient presque superposées.

3.3.2 Correction du frameskip et combinaison optimisée

Après avoir identifié et corrigé le double frameskip (désactivation du skip manuel et maintien du skip=4 de l'API ALE), nous avons relancé l'entraînement sur Breakout. Plutôt que de re-tester toutes les variantes, nous avons focalisé sur la combinaison **Double DQN + Dueling DQN**, qui a démontré un bond spectaculaire des performances :

- **Score moyen de ~350 points** atteint après 44 000 épisodes (*Figure 7*).
- Convergence plus rapide et stabilité accrue du loss.

Cette configuration (bug corrigé + Double + Dueling) constitue notre meilleur compromis entre complexité et efficacité pour Breakout.

3.4 Validation finale sur Pong

Avec le pipeline complet et le bug résolu, nous sommes retournés sur Pong pour valider les améliorations. Après **12 000 épisodes**, l'agent a atteint un **score stable de 17–18 points** par match (victoires fréquentes 21–3 ou 21–4), démontrant la robustesse de la méthodologie sur un autre titre que Breakout (*Figure 8*).

4. Conclusions et perspectives

En synthèse :

- **Space Invaders** présente des défis uniques liés au frameskip et aux lasers « clignotants ».
- **Pong**, d'abord trop lent, a validé en fin de pipeline l'efficacité du DQN corrigé.
- **Breakout** s'est révélé le plus favorable, atteignant un score >350 avec Double DQN + Dueling une fois le double frameskip corrigé.

Perspectives futures :

1. Affiner le PER avec ajustement des coefficients α/β et introduction de multi-step targets.
2. Intégrer des algorithmes distributionnels (C51, Rainbow) pour améliorer la précision de l'estimation de la valeur.
3. Étudier le transfert de connaissances entre jeux (pré-entraînement sur Breakout, fine-tuning sur Pong).
4. Tester des méthodes d'exploration avancées (Noisy Nets, Bootstrapped DQN) pour renforcer la capacité d'exploration.

III. Annexe

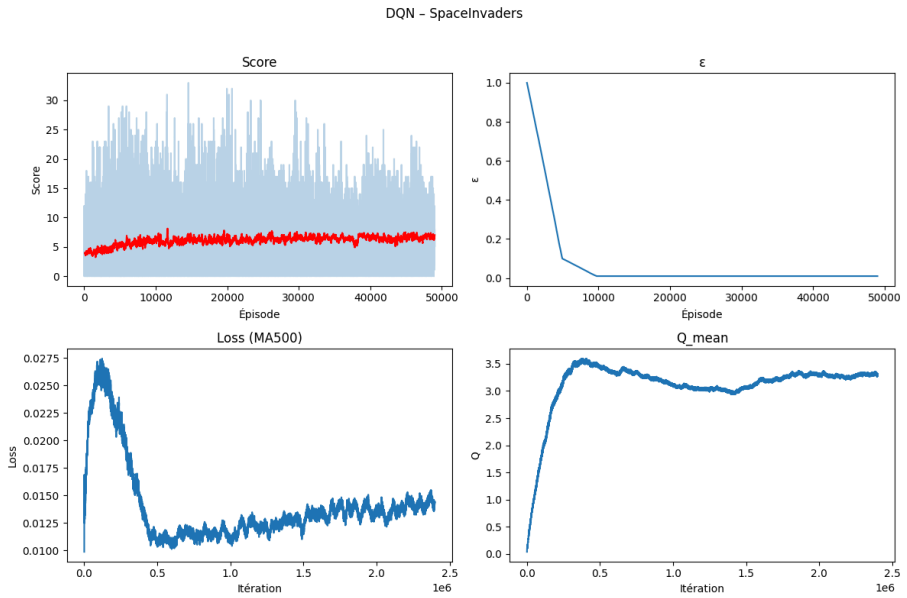


Figure 1 : Metrics Score, Perte, Epsilon, Q_mean (estimation du modèle) sur Space Invaders

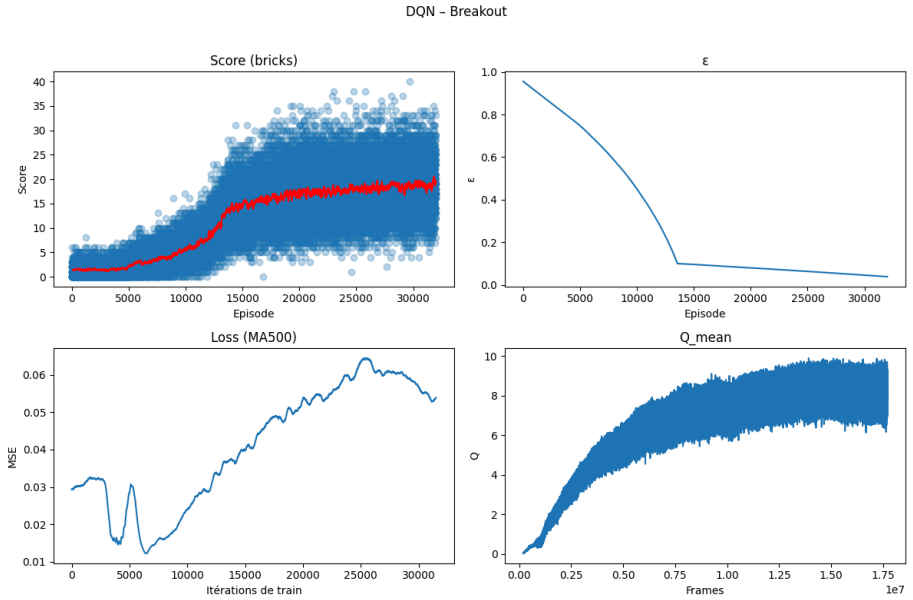


Figure 2 : Metrics Score, Perte, Epsilon, Q_mean (estimation du modèle) sur Breakout avec un DQN vanilla

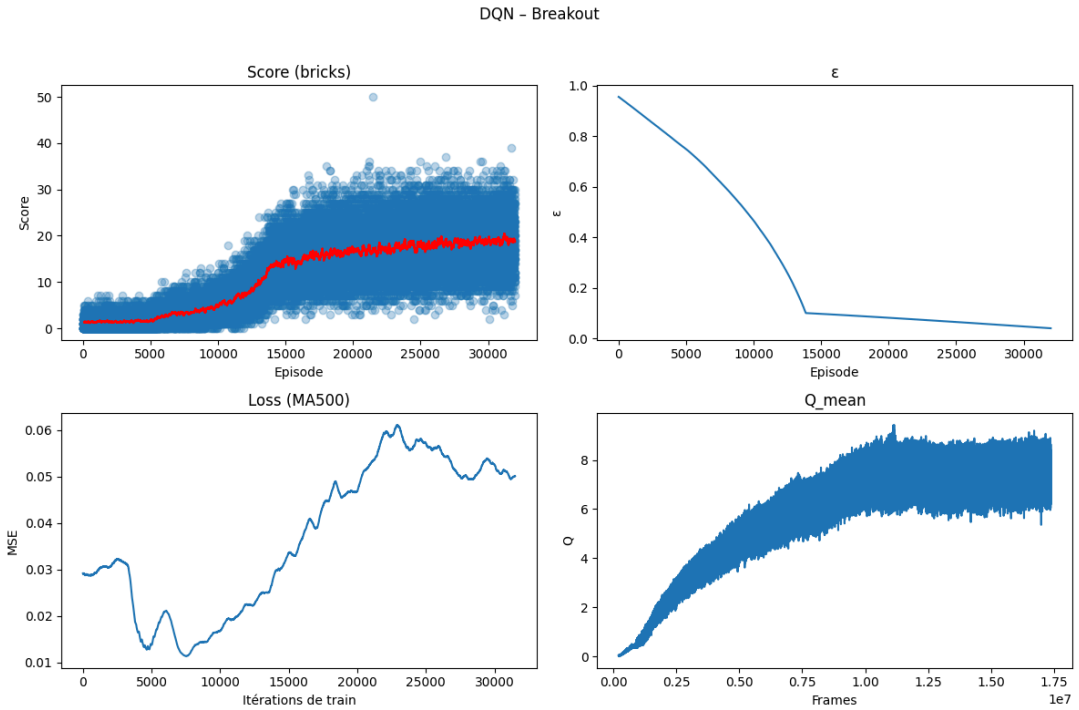


Figure 3 : Metrics Score, Perte, Epsilon, Q_mean (estimation du modèle) sur Breakout avec un double DQN

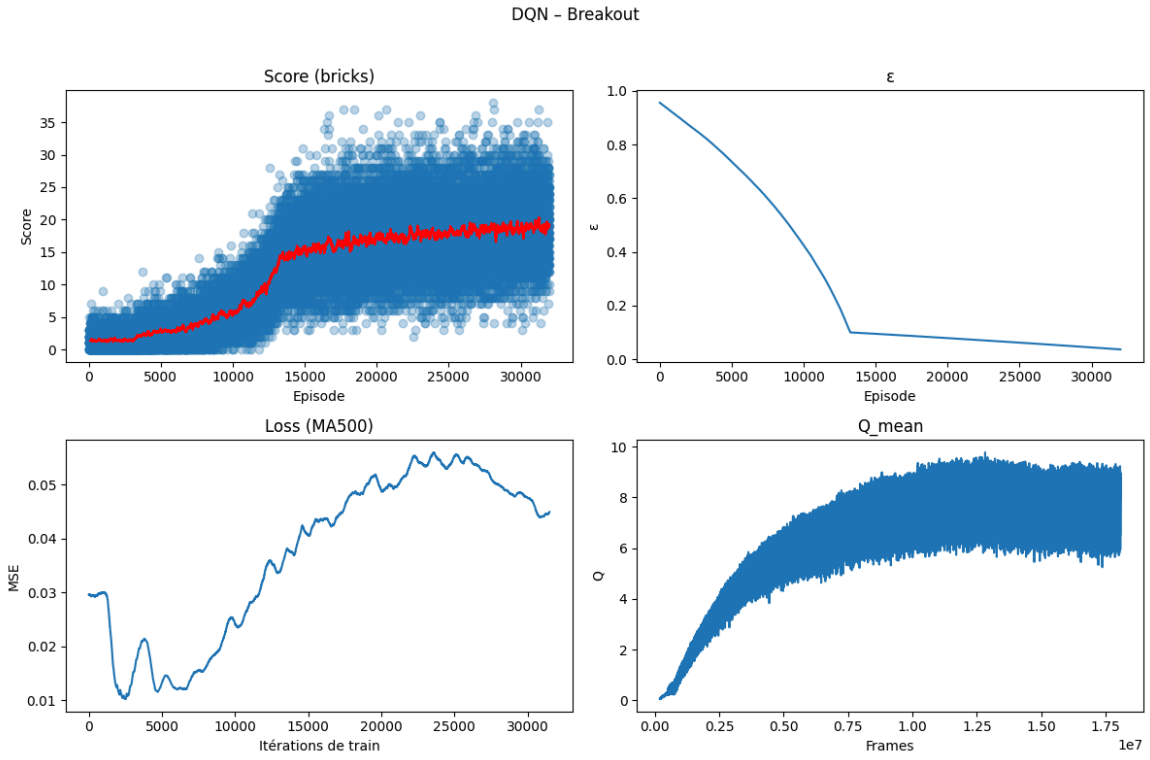


Figure 4 : Metrics Score, Perte, Epsilon, Q_mean (estimation du modèle) sur Breakout avec deux têtes dueling

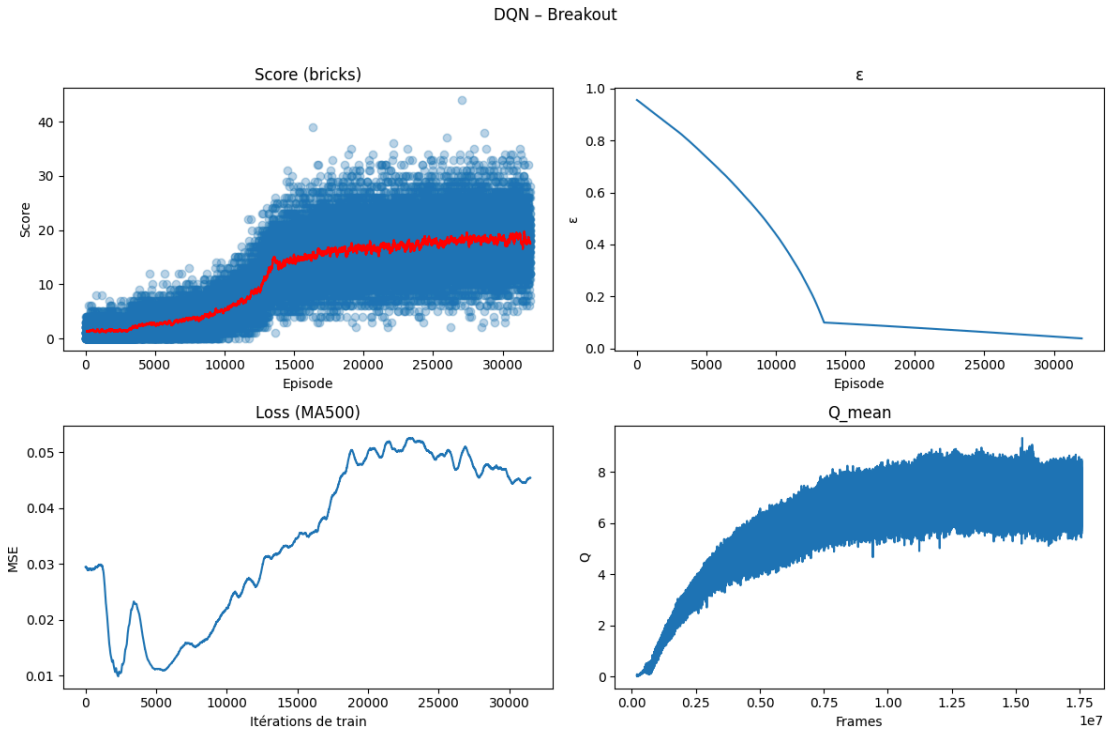


Figure 5 : Metrics Score, Perte, Epsilon, Q_mean (estimation du modèle) sur Breakout avec deux têtes dueling + double DQN

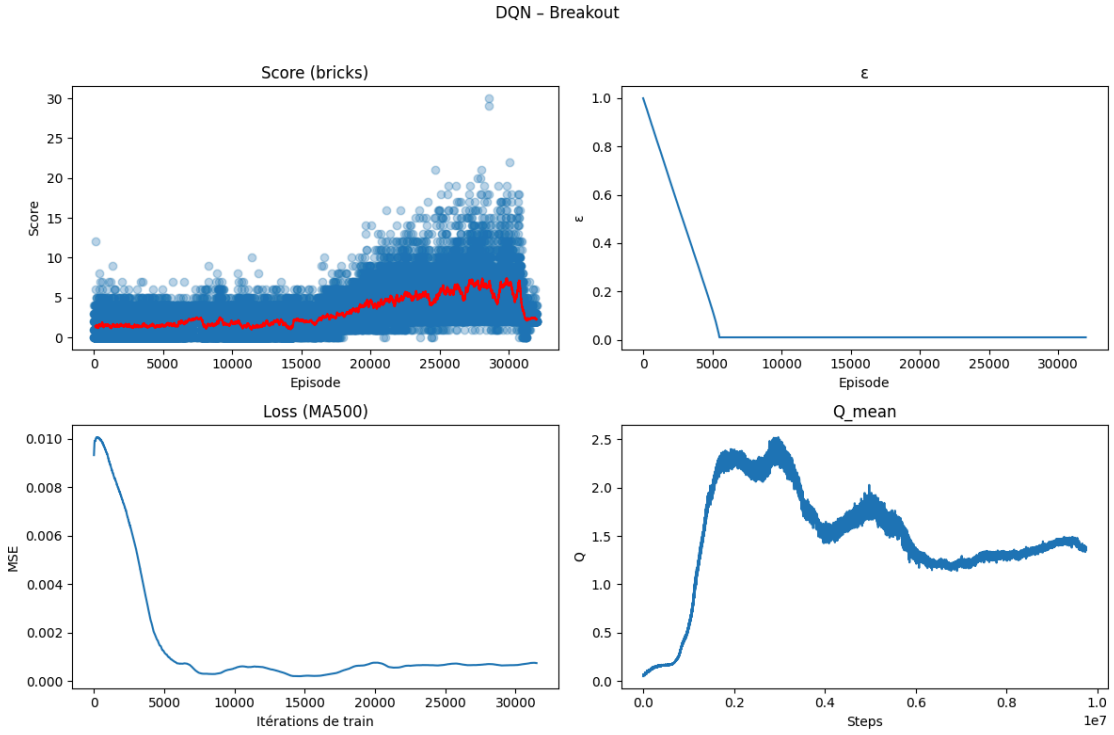


Figure 6 : Metrics Score, Perte, Epsilon, Q_mean (estimation du modèle) sur Breakout avec DQN vanilla et PER

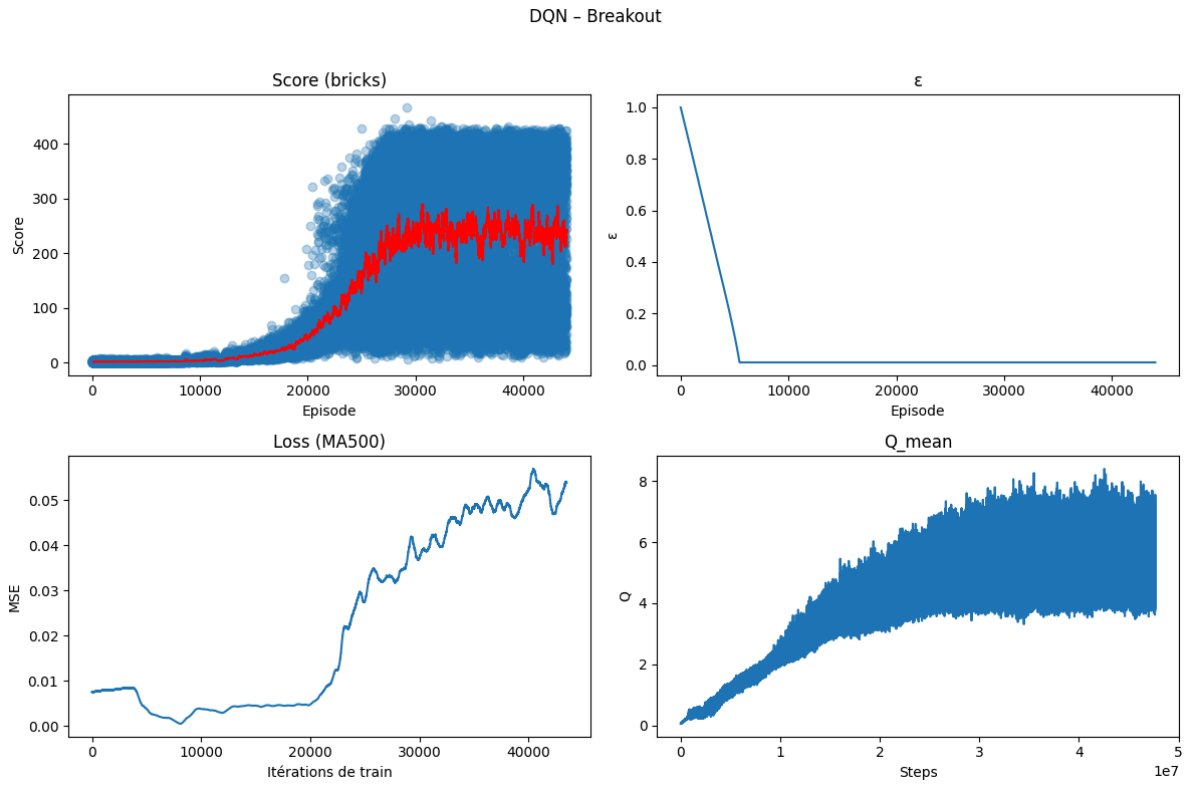


Figure 7 : Metrics Score, Perte, Epsilon, Q_mean (estimation du modèle) sur Breakout avec dueling et double DQN (après frameskip fixé)

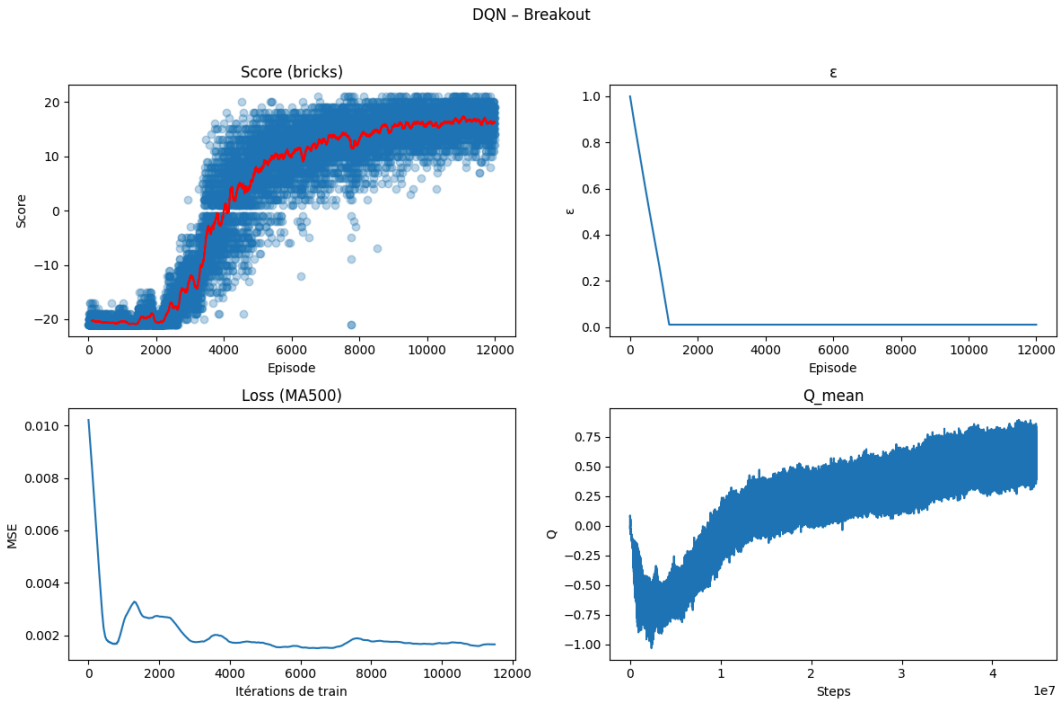


Figure 8 : Metrics Score, Perte, Epsilon, Q_mean (estimation du modèle) sur Pong (malgré le titre) avec double DQN + dueling (même configuration que figure 7)

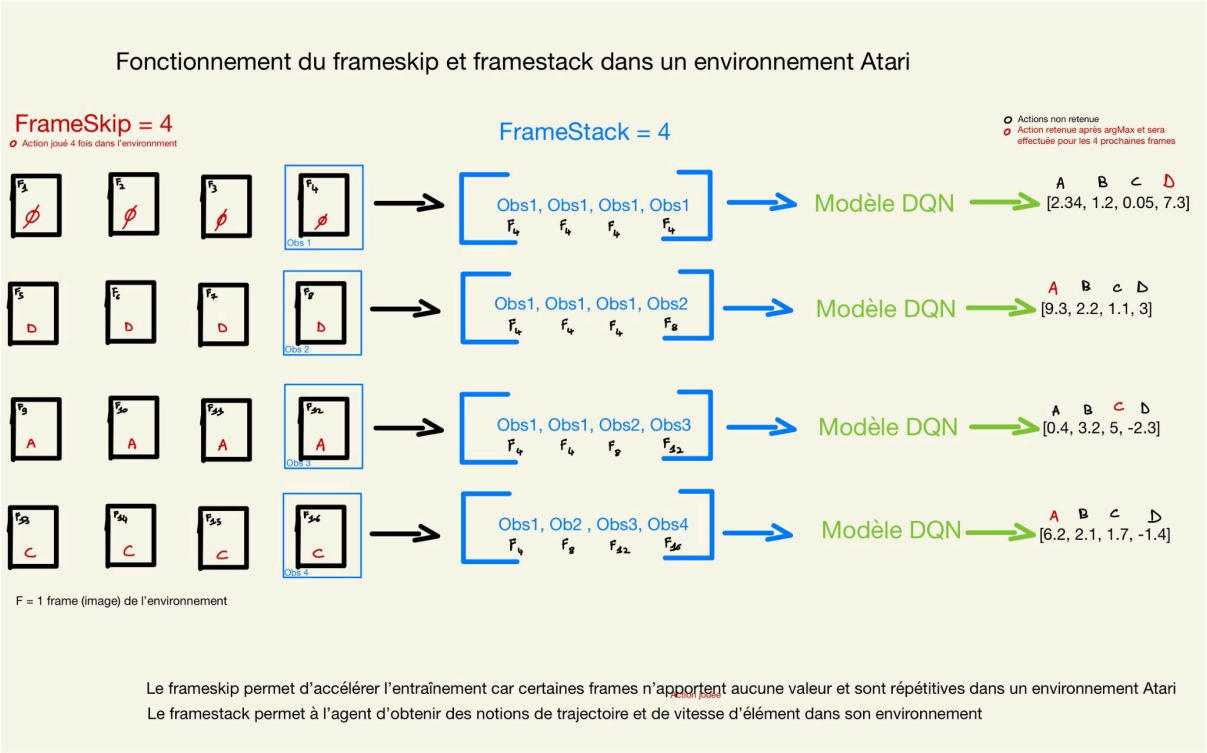


Schéma explicatif du fonctionnement de frameskip et framestack dans un environnement atari

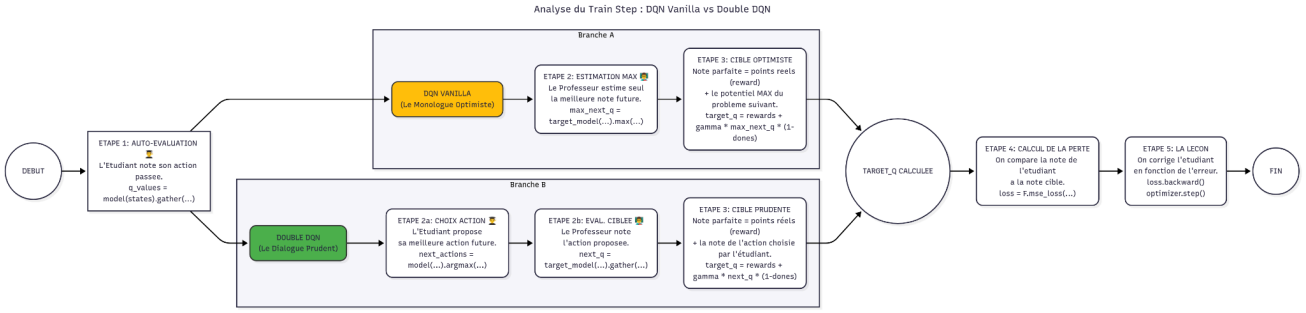


Schéma explicatif de la comparaison entre double DQN et vanilla DQN lors d'une étape d'entraînement.

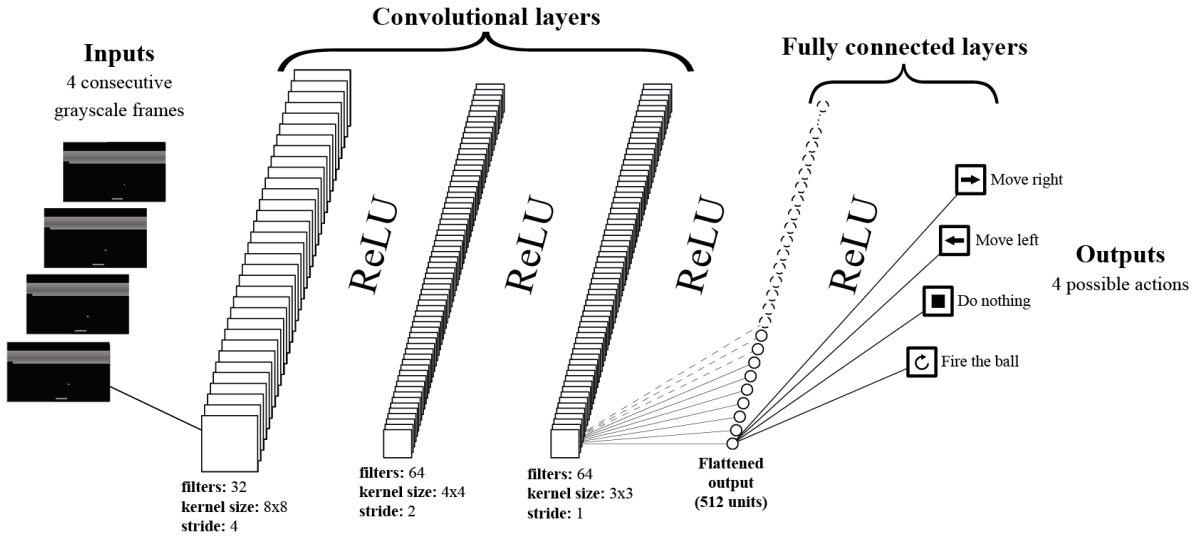


Schéma explicatif de l'architecture de raw DQN

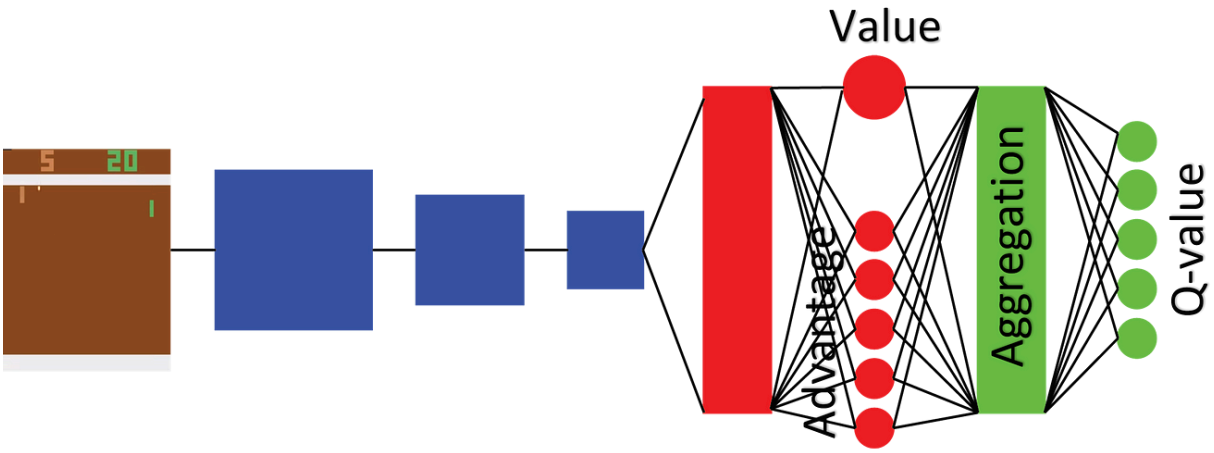


Schéma explicatif de l'architecture dueling DQN