RAPPORT DE PROJET D'ÉTUDES ET DE RECHERCHE

Apprentissage par renforcement dans les jeux : une étude de cas sur 2048

Nathan Edery, Aubin Courtial, Alexis Blond

Encadrant : L. Fillatre
Période du projet : septembre 2024 - mars 2025
Département : Mathématiques appliquées et modélisation - Science des données
Établissement : Polytech Nice Sophia

Contents

1 Résumé du projet **3**

2 Organisation du projet **4**

3 Environnement technique **5**

4 Description du travail réalisé depuis le premier rapport **6**
4.1 Rappel des réalisations initiales 6
4.1.1 Motivation de Value Iteration 6
4.2 Validation théorique de Value Iteration 6
4.2.1 Définition du processus de décision markovien 7
4.2.2 Value Iteration : Fondements théoriques 8
4.2.3 Convergence de Value Iteration 9
4.2.4 Résultats empiriques et analyse des échecs 9
4.2.5 Généralisation à des grilles de dimensions supérieures 10
4.2.6 Complexité de VI pour une grille 3×3 11
4.3 Optimisation du code 12
4.3.1 Regroupement des opérations 12
4.3.2 Utilisation de Numba 12
4.3.3 Techniques de Bitboard 13
4.3.4 Techniques Python modernes 13
4.4 Amélioration du système de récompense 13
4.5 Résultats obtenus et analyse des performances 13
4.6 Synthèse et perspectives 14
4.7 Temporal Differencing for N-Tuple Networks 15
4.7.1 Motivation et Contexte 15
4.7.2 Modélisation par N-Tuple Networks 15
4.7.3 Introduction au Temporal Difference Learning 16
4.7.4 Méthodologie du Temporal Difference Learning 17
4.7.5 Résultat 17
4.7.6 Grille 3x3 17
4.7.7 Grille 4x4 18
4.8 Démonstrateur 19
4.8.1 Introduction 19
4.8.2 Écran d'accueil 20
4.8.3 Interface de jeu 2048 sur grille 3×3 ou 4×4 assistée par un agent . . 20

5 Difficultés rencontrées **22**
5.1 Deep Q-Learning 22
5.2 Défis avec les méthodes Monte Carlo et MCTS 23
5.3 Contraintes de temps et de ressources 23

6 Conclusion et perspectives **25**

Chapter 1

Résumé du projet

Ce projet explore l'application des techniques d'apprentissage par renforcement au jeu 2048 dans une approche à la fois théorique et expérimentale. L'objectif principal est de modéliser le jeu d'en extraire une politique optimale, tout en surmontant les défis liés à l'explosion combinatoire de l'espace des états et à la nature stochastique du jeu.

- **Modélisation et validation théorique :** Le jeu est formalisé par un processus de décision markovien, en définissant précisément les espaces d'états et d'actions, ainsi que les probabilités de transition et la fonction de récompense. La méthode de Value Iteration a été d'abord appliquée sur des grilles réduites 2x2 pour valider théoriquement sa convergence et l'efficacité de la politique obtenue.

- **Optimisation et performances :** Pour faire face à la complexité computationnelle, plusieurs techniques d'optimisation ont été mises en œuvre. L'utilisation de Numba pour la compilation JIT, l'adoption de représentations en bitboard et l'emploi de constructions idiomatiques de Python ont permis d'accélérer considérablement les simulations, multipliant par cinq le nombre de parties jouées et améliorant le taux de victoire de manière significative.

- **Méthodes avancées et exploration complémentaire :** En complément de la Value Iteration, le projet a étudié l'approche du Temporal Difference Learning associée aux N-Tuple Networks. Cette méthode offre une alternative plus flexible et scalable pour appréhender des grilles de dimensions supérieures, en décomposant l'état du jeu en configurations locales pertinentes.

- **Développement d'un démonstrateur interactif :** Un interface graphique a été réalisé pour permettre de jouer au 2048 avec différents agents (Random, Value Iteration, Expectimax). Ce démonstrateur offre une validation pratique des modèles développés et facilite l'analyse comparative des stratégies adoptées.

Les résultats expérimentaux montrent une nette amélioration du taux de victoire, passant d'environ 20% à 85% sur certaines configurations, et permettant même d'obtenir la fameuse tuile 2048 sur un jeu de dimension 4x4. Ce projet met en lumière le potentiel des méthodes d'apprentissage par renforcement dans des environnements complexes.

Chapter 2

Organisation du projet

Depuis la date de rendu du premier rapport, l'équipe a continué à s'organiser de manière structurée pour faire avancer le Projet d'Etude et de Recherche (PER). Voici comment nous avons procédé :

Répartition des rôles et responsabilités

Les rôles des membres de l'équipe ont quelque peu évolué pour mieux répondre aux exigences du projet :

- **Nathan Edery** : Continue d'explorer les aspects théoriques et d'implémenter les algorithmes d'apprentissage par renforcement (RL). Il s'est notamment concentré sur l'exploration de nouvelles techniques de RL pour améliorer les performances des algorithmes existants.
- **Aubin Courtial** : S'est chargé de la réalisation du démonstrateur et de la création du Docker final, assurant ainsi la validation pratique des solutions proposées.
- **Alexis Blond** : S'est focalisé sur l'optimisation des méthodes déjà en place, notamment l'itération sur les valeurs (Value Iteration), afin d'améliorer leur efficacité et leur robustesse. Il s'est également concentré sur la partie Deep Q-Learning pour finalement comprendre son inefficacité dans notre contexte.

Respect du planning prévisionnel

L'organisation prévisionnelle a été globalement respectée, bien que certains ajustements aient été nécessaires. Nous nous attendions à consacrer plus de temps au Deep Q-Learning, mais nous avons rapidement rencontré des obstacles. Cela nous a conduits à explorer davantage de pistes que prévu, en nous concentrant sur une dimensionnalité réduite (3x3) pour mieux comprendre les défis rencontrés.

Ces ajustements ont permis de renforcer notre approche et d'optimiser notre méthodologie pour les phases ultérieures du projet.

Chapter 3

Environnement technique

Ce chapitre résume l'environnement technique tel que décrit dans le premier rapport, tout en intégrant les ajustements apportés au second rapport.

Rappel du premier rapport

Comme précisé dans le premier rapport, le projet s'appuie sur un environnement technique diversifié intégrant des outils et technologies modernes. Le développement repose principalement sur le langage Python, choisi pour sa flexibilité et son vaste écosystème scientifique. Des bibliothèques telles que NumPy, essentielles pour la manipulation de données complexes et les calculs numériques, et PyTorch, utilisée pour construire et entraîner des modèles complexes dans le cadre du Deep Q-Learning, ont été employées. L'environnement de développement a été constitué autour de Visual Studio Code et Google Colab, facilitant ainsi la collaboration et l'exécution de notebooks. Pour la gestion du code source, Git et GitHub ont permis un suivi rigoureux des versions, tandis que Docker a assuré la reproductibilité de l'environnement de développement. Enfin, le projet prévoyait l'utilisation du SuperGPU PolytechA100 pour les phases intensives d'entraînement des réseaux de neurones et la simulation du jeu 2048.

Adaptations et ajustements depuis le premier rapport

Au cours du projet, il est apparu que le Deep Q-Learning n'était pas suffisamment efficace pour justifier l'utilisation du SuperGPU A100. Par conséquent, l'usage de ce GPU n'a pas été retenu et nous n'avons pas parallélisé nos autres calculs sur GPU. Afin de contourner ces limitations, nous avons largement utilisé Kaggle, ce qui nous a permis de faire tourner des notebooks sur un serveur distant pendant des périodes pouvant aller jusqu'à 12 heures, évitant ainsi la sollicitation de nos machines locales. De plus, l'utilisation de Numba a permis d'accélérer la compilation, et GitHub a continué d'être l'outil privilégié pour partager et collaborer sur le code.

Cette révision de l'environnement technique reflète notre capacité à adapter nos choix technologiques en fonction des résultats obtenus et des contraintes rencontrées au cours du développement.

Chapter 4

Description du travail réalisé depuis le premier rapport

4.1 Rappel des réalisations initiales

Dans le premier rapport, nous avions posé les bases du projet en développant une version simplifiée du jeu 2048, jouable sur une grille 3×3 avec un objectif fixé à la tuile 256. La méthode utilisée reposait sur l'équation de Bellman, qui permettait d'estimer la valeur de chaque état du jeu en combinant la récompense immédiate issue des fusions de tuiles et la valeur des états futurs. Pour cela, nous avions mis en place un algorithme de Value Iteration (VI) intégrant l'aspect stochastique de l'apparition aléatoire des nouvelles tuiles. Ce travail initial, réalisé sur environ 80 000 parties simulées en cinq heures, avait conduit à un taux de victoire d'environ 20 %, fournissant ainsi un premier indicateur de la capacité de l'agent à progresser dans l'apprentissage par renforcement.

4.1.1 Motivation de Value Iteration

Dès le début de ce projet, nous avons choisi d'explorer VI, une méthode reposant sur l'équation de Bellman, que nous avions étudiée en cours et mise en œuvre lors de travaux pratiques. Ce choix s'est imposé naturellement, d'autant que notre encadrant nous avait recommandé de partir sur cette approche, reconnue pour sa rigueur théorique et sa simplicité d'implémentation.

Nous nous sommes penchés directement sur la théorie, sans avoir réalisé une revue exhaustive de la littérature existante ni vérifié l'étendue des travaux antérieurs dans ce domaine, nous avons considéré VI comme un point de départ pertinent pour aborder le problème de la prise de décision dans le jeu 2048. Cette méthode nous a ainsi permis de transposer des concepts fondamentaux de l'apprentissage par renforcement à un contexte ludique, tout en offrant une première évaluation de l'efficacité de notre agent à travers des simulations sur un nombre conséquent de parties.

4.2 Validation théorique de Value Iteration

Dans cette section, nous procédons à une analyse mathématique rigoureuse de VI appliquée au jeu 2048 dans une grille de dimension 2×2, avec pour condition de victoire l'apparition d'une tuile de valeur 16. L'objectif est de valider théoriquement la convergence et l'efficacité de cette méthode dans le cadre d'un processus de décision markovien

(MDP) adapté à un jeu réduit, en prenant en compte la nature stochastique des transitions.

4.2.1 Définition du processus de décision markovien

Le jeu 2048 est modélisé comme un MDP défini par le quintuplet

$$\langle S, A, P, R, \gamma \rangle,$$

où :

- S **(espace d'états)** : L'ensemble des configurations possibles de la grille. Chaque cellule peut contenir une tuile allant de 0 (case vide ; 2^0) à 32768 (2^{15}). Dans la pratique, pour une grille 2×2 avec une tuile maximale de 16, les exposants effectifs sont limités de 0 à 4. L'espace total des états est donc de

  $$|S| = 16^{2\times 2} = 16^4 = 65\,536,$$

  représentant toutes les combinaisons possibles des exposants dans les quatre cellules (même si certains états sont inatteignables depuis les conditions initiales sur une grille 2x2).

- A **(espace d'actions)** : L'ensemble des actions possibles, correspondant aux quatre mouvements directionnels : haut, bas, gauche, droite. Ainsi,

  $$|A| = 4.$$

- P **(probabilités de transition)** : Les transitions sont déterministes pour l'exécution d'un mouvement (fusion et déplacement des tuiles), et stochastiques pour l'insertion d'une nouvelle tuile après chaque mouvement valide. Après un mouvement, une tuile de valeur 2 (exposant 1) apparaît avec une probabilité de 0,9 et une tuile de valeur 4 (exposant 2) avec une probabilité de 0,1, dans une case vide choisie uniformément. Formellement, pour un état $s \in S$, une action $a \in A$ et un état suivant $s' \in S$, la probabilité de transition est donnée par :

  $$p(s', r \mid s, a) = \begin{cases} P(s', r \mid s, a), & \text{si } s' \text{ est obtenu après le mouvement } a \text{ et l'insertion aléatoire,} \\ 1, & \text{si } s \text{ est terminal et } s' = s,\ r = 0, \\ 1, & \text{si le mouvement } a \text{ est invalide, } s' = s,\ r = -10000, \\ 0, & \text{sinon.} \end{cases}$$

  Les détails des transitions incluent :

  - Si l'état s est terminal (victoire ou défaite), la transition est une boucle sur lui-même avec une récompense $r = 0$.
  - Si le mouvement a est invalide (aucun changement dans la grille), la transition est une boucle sur lui-même avec une récompense $r = -10000$.
  - Si le mouvement a crée immédiatement un état gagnant (tuile maximale ≥ 16), la récompense est $r = 1000$ et l'état devient terminal.
  - Sinon, après le mouvement, une nouvelle tuile est insérée et la récompense de base est le changement de score dû au mouvement. Si, après insertion, aucun mouvement valide n'est possible, la récompense devient $r = -1000$ et l'état est terminal.

- R **(fonction de récompense)** : La récompense r est définie comme suit :
  - $r = -10000$ pour un mouvement invalide,
  - $r = 1000$ si le mouvement crée une tuile ≥ 16 (victoire),
  - $r = -1000$ si, après insertion, aucun mouvement valide n'est possible (défaite),
  - Sinon, r est égal au changement de score dû au mouvement (calculé comme la différence entre les scores de la grille avant et après le mouvement, où le score est déterminé par les fusions effectuées).
- γ **(facteur de réduction)** : Dans notre implémentation, $\gamma = 1.0$. Bien que $\gamma = 1.0$ puisse poser problème dans des MDPs non épisodiques (avec des valeurs potentiellement infinies), le jeu étant épisodique (se terminant toujours par une victoire ou une défaite), les valeurs restent finies.

4.2.2 Value Iteration : Fondements théoriques

L'algorithme de Value Iteration (VI) vise à déterminer la fonction de valeur optimale V^*, qui satisfait l'équation de Bellman suivante pour tout état non terminal $s \in S$:

$$V^*(s) = \max_{a \in A} \left[r(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) V^*(s') \right].$$

où :

- $V^*(s)$ est la valeur optimale de l'état s,
- A est l'ensemble des actions possibles,
- $r(s,a)$ est la récompense immédiate obtenue en exécutant l'action a depuis l'état s,
- $p(s'|s,a)$ est la probabilité de transition vers l'état s' après avoir pris l'action a dans s,
- $\gamma \in [0,1]$ est le facteur d'actualisation, qui pondère l'importance des récompenses futures.

L'algorithme de VI procède comme suit :

- **Initialisation** :
$$V_0(s) = 0 \quad \forall\, s \in S.$$
- **Mise à jour itérative** : À chaque itération $k \geq 0$, on met à jour la valeur de chaque état selon :
$$V_{k+1}(s) = \max_{a \in A} \left[r(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) V_k(s') \right].$$
L'itération continue jusqu'à convergence, c'est-à-dire lorsque :
$$\max_{s \in S} |V_{k+1}(s) - V_k(s)| < \theta,$$
où $\theta > 0$ est un seuil de convergence (par exemple, $\theta = 0.0001$).

- **Extraction de la politique optimale** : Une fois V^* approximée, la politique optimale π^* est déterminée par :

$$\pi^*(s) = \arg\max_{a \in A} \left[r(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) V^*(s') \right].$$

Cette politique π^* spécifie pour chaque état s l'action optimale a^* qui maximise la valeur espérée des récompenses cumulées.

4.2.3 Convergence de Value Iteration

La convergence de VI est garantie dans notre contexte pour les raisons suivantes :

- **Finitude des espaces d'états et d'actions** : $|S| = 65\,536$ et $|A| = 4$.
- **Récompenses bornées** : Les récompenses possibles sont :

$$\begin{aligned} &r = -10000 \quad \text{(mouvement invalide)}, \\ &r = -1000 \quad \text{(défaite)}, \\ &r = 1000 \quad \text{(victoire)}, \\ &r \geq 0 \quad \text{(changement de score)}. \end{aligned}$$

- **Nature épisodique du jeu** : Même si $\gamma = 1.0$, le fait que le jeu se termine en un nombre fini d'étapes garantit que les valeurs cumulées restent finies.

Ainsi, VI converge vers la fonction de valeur optimale V^*, et la politique optimale π^* extraite permet de maximiser la récompense cumulée attendue.

4.2.4 Résultats empiriques et analyse des échecs

Lors des simulations, nous avons comparé les performances de la politique optimale dérivée par VI à une politique aléatoire sur 1000 parties :

- **Politique optimale** : Taux de victoire de 96,7%.
- **Politique aléatoire** : Taux de victoire de 15%.

La figure ci–dessous illustre la répartition des valeurs V pour différents états atteignables, en fonction de l'action optimale sélectionnée (U, D, L ou R). Chaque point correspond à un état, et sa position verticale indique la valeur de cet état. On observe que certains états possèdent une valeur élevée (proche de 1000), ce qui correspond à des configurations favorables menant rapidement à la victoire. À l'inverse, les points bas traduisent des états où la probabilité d'échec est plus forte ou bien où aucune fusion avantageuse n'est possible ou c'est un état qui est terminé (ayant une tuile 16). Les différentes couleurs montrent que l'action optimale peut varier selon l'état, reflétant la politique dérivée par Policy Iteration pour maximiser la récompense attendue.
Ces résultats confirment l'efficacité de VI pour apprendre une politique performante. Les quelques échecs (33 parties perdues avec la politique optimale) s'expliquent par la nature stochastique des transitions, en particulier par l'insertion aléatoire des nouvelles tuiles. La politique optimale est calculée en considérant une espérance des résultats futurs basée sur une probabilité de 0,9 pour une tuile 2 et 0,1 pour une tuile 4. Cependant, l'apparition d'une tuile 4 dans certaines situations peut mener à une configuration défavorable, rendant la défaite inévitable.

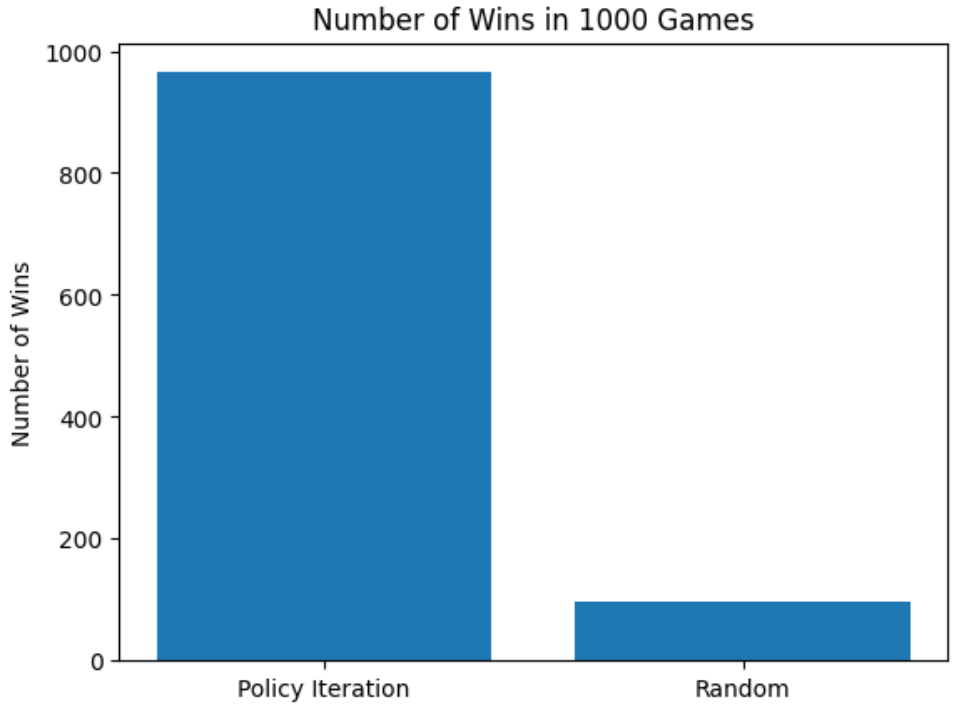

Figure 4.1: Comparaison des performances entre la politique optimale et une politique aléatoire sur 1000 parties.

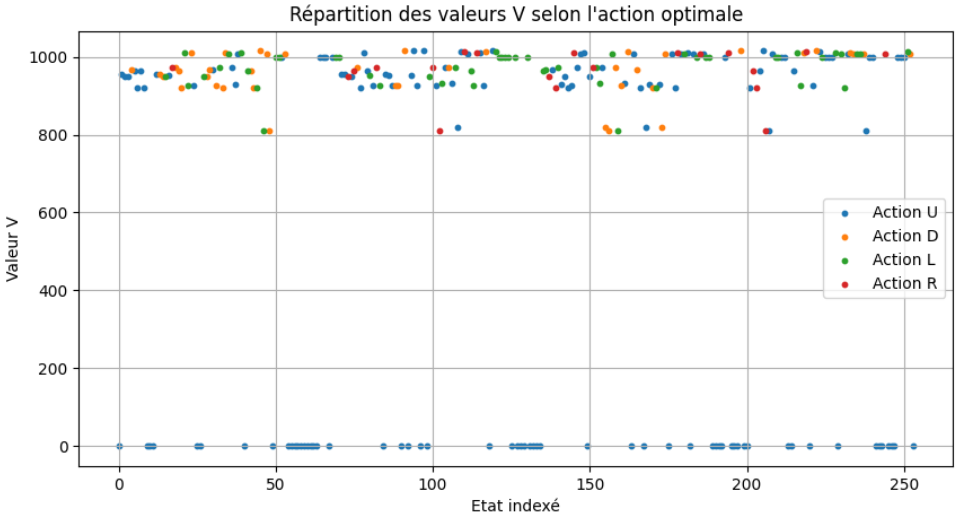

Figure 4.2: Valeurs V pour différents états en fonction de l'action optimale.

4.2.5 Généralisation à des grilles de dimensions supérieures

Les résultats obtenus sur une grille 2×2 confirment l'efficacité de VI pour optimiser les décisions dans un cadre restreint. Toutefois, pour évaluer la scalabilité de cette approche, il est essentiel d'examiner son application à des configurations plus grandes, notamment une grille 3×3. L'augmentation du nombre de cellules entraîne une explosion combinatoire de l'espace des états et des transitions, posant des défis computationnels significatifs. Dans cette section, nous analysons mathématiquement cette complexité et discutons des implications pour l'utilisation de Value Iteration dans des environnements de plus grande dimension.

4.2.6 Complexité de VI pour une grille 3×3

Nous considérons ici le jeu 2048 dans une version restreinte à une grille de dimension 3×3, avec une condition de victoire imposée par l'apparition d'une tuile de valeur 32. Pour ce faire, chaque cellule de la grille peut prendre une valeur dans l'ensemble discret

$$\mathcal{V} = \{0, 2, 4, 8, 16, 32\},$$

où 0 représente l'absence de tuile (cellule vide). L'ensemble des états possibles S du jeu s'identifie alors à l'ensemble des fonctions

$$s : \{1, 2, \ldots, 9\} \to \mathcal{V},$$

ce qui conduit immédiatement à une cardinalité

$$|S| = |\mathcal{V}|^9 = 6^9 = 10\,077\,696.$$

Le processus décisionnel est modélisé par un MDP défini par le quintuplet $\langle S, A, P, R, \gamma \rangle$, où l'ensemble des actions est

$$A = \{\text{haut},\ \text{bas},\ \text{gauche},\ \text{droite}\},$$

et donc $|A| = 4$. La dynamique du jeu comporte deux phases distinctes : d'une part, l'action choisie $a \in A$ est appliquée de manière déterministe pour déplacer et fusionner les tuiles, générant ainsi un état intermédiaire s^* ; d'autre part, une nouvelle tuile est insérée de façon stochastique dans l'une des cellules vides de la configuration obtenue.

Soit $n(s^*)$ le nombre de cellules vides dans l'état intermédiaire s^*, défini par

$$n(s^*) = \#\{i \in \{1, \ldots, 9\} \mid s^*(i) = 0\}.$$

La phase d'insertion consiste à sélectionner, uniformément au hasard, une cellule parmi ces $n(s^*)$ et à y insérer une tuile de valeur 2 avec probabilité $p_2 = 0.9$, ou de valeur 4 avec probabilité $p_4 = 0.1$. Pour chaque cellule vide, il existe ainsi deux transitions possibles. Par conséquent, pour un état intermédiaire donné s^*, le nombre total de transitions possibles induites par l'insertion est

$$N(s^*) = 2\, n(s^*).$$

Pour établir une estimation moyenne, nous introduisons l'hypothèse que, dans un état "typique", la probabilité qu'une cellule soit vide est constante et égale à $\frac{1}{6}$, c'est-à-dire

$$\mathbb{P}\big(s(i) = 0\big) = \frac{1}{6} \quad \forall\, i \in \{1, \ldots, 9\}.$$

En appliquant la linéarité de l'espérance, le nombre moyen de cellules vides dans un état s est alors donné par

$$\mathbb{E}[n(s)] = \sum_{i=1}^{9} \mathbb{P}\big(s(i) = 0\big) = 9 \times \frac{1}{6} = 1.5.$$

Il s'ensuit que, pour un état intermédiaire s^*, le nombre moyen de transitions dues à l'insertion est

$$\mathbb{E}[N(s^*)] = 2\, \mathbb{E}[n(s)] = 2 \times 1.5 = 3.$$

Compte tenu du fait que pour chaque état $s \in S$, l'agent dispose de 4 actions, la structure de transition P doit, pour chaque couple (s, a), prendre en compte en moyenne 3 transitions différentes issues de l'insertion aléatoire. Ainsi, le nombre moyen total de transitions sortantes par état est

$$T_{\text{moyen}} = |A| \times \mathbb{E}[N(s^*)] = 4 \times 3 = 12.$$

En considérant l'ensemble de l'espace d'états, le nombre total de transitions que le MDP doit modéliser est donc de l'ordre de

$$|S| \times T_{\text{moyen}} = 6^9 \times 12 \approx 10\,077\,696 \times 12 \approx 120\,932\,352.$$

Cette explosion combinatoire, où le nombre d'états et, par extension, le nombre total de transitions augmentent de manière exponentielle, illustre rigoureusement pourquoi l'algorithme de Value Iteration devient rapidement inapplicable pour des grilles plus grandes. En effet, la complexité temporelle et spatiale de l'algorithme est directement proportionnelle à la taille de l'espace des états ainsi qu'au nombre de transitions à traiter. Dans notre cas, même une estimation moyenne aboutit à la nécessité de gérer plus de 120 millions d'entrées dans la fonction de transition P pour atteindre seulement la victoire avec une tuile à 32. Cette réalité mathématique justifie le recours, dans des contextes de dimension plus élevée, à des méthodes d'approximation ou à des techniques de réduction de dimension, afin de pallier l'intractabilité computationnelle induite par la croissance exponentielle de l'espace d'états.

4.3 Optimisation du code

Pour accélérer le processus d'entraînement et augmenter le nombre de simulations, nous avons adopté une stratégie d'optimisation multi-niveaux qui combine plusieurs techniques complémentaires. Ces améliorations nous ont permis de multiplier par cinq la vitesse d'exécution de notre code, un gain essentiel pour l'entraînement des agents en apprentissage par renforcement [4]. Voici les principaux axes d'optimisation :

4.3.1 Regroupement des opérations

En fusionnant les opérations de déplacement et de fusion des tuiles en une seule passe, nous avons réduit significativement le nombre d'itérations nécessaires pour mettre à jour l'état du plateau. Cette approche simplifie la logique de gestion du jeu en éliminant des étapes redondantes et accélère les mises à jour [4].

4.3.2 Utilisation de Numba

Grâce à Numba, nous compilons à la volée (JIT) les fonctions critiques, notamment celles qui traitent les boucles et les calculs intensifs. En pratique, Numba permet de convertir automatiquement des fonctions Python annotées en code machine très optimisé, éliminant ainsi l'overhead d'interprétation inhérent à Python [12].

4.3.3 Techniques de Bitboard

Au départ, nous utilisions une représentation matricielle du jeu, où l'état du plateau était stocké sous forme de grille 2D. Pour améliorer cela, nous avons adopté la représentation bitboard, qui consiste à encoder chaque tuile sur 4 bits, permettant ainsi de stocker l'état complet d'une grille dans un seul entier. Dans notre code, nous utilisons cette technique pour manipuler efficacement les états du jeu. Les opérations de glissement et de fusion sont pré-calculées et stockées dans des tables de lookup, ce qui permet d'obtenir instantanément le résultat de chaque mouvement. Par ailleurs, les opérations bit à bit offrent une gestion rapide des symétries (transpositions et inversions) du plateau [17].

4.3.4 Techniques Python modernes

L'utilisation de compréhensions de liste, du slicing et d'autres constructions idiomatiques de Python remplace avantageusement les boucles classiques. Ces techniques rendent le code non seulement plus concis, mais également plus performant en optimisant la manipulation des données [4].

4.4 Amélioration du système de récompense

Parallèlement aux optimisations de performance, nous avons introduit un nouveau système de récompense destiné à guider l'agent vers des configurations de plateau plus stables et rentables. Un bonus est désormais attribué lorsque la tuile la plus élevée se trouve dans l'un des coins du plateau. Calculé comme le carré de la valeur de la tuile en coin, ce bonus encourage l'agent à maintenir sa tuile la plus forte dans une position stratégique, facilitant ainsi l'organisation des autres tuiles pour optimiser les fusions futures.

4.5 Résultats obtenus et analyse des performances

Grâce aux améliorations apportées, nous avons pu simuler 350 000 parties sur un plateau 3×3 avec la tuile de victoire fixée à 512 en seulement cinq heures, multipliant ainsi par plus de quatre le nombre de parties jouées. Cette augmentation nous a permis d'explorer plus largement l'espace des configurations et d'accélérer la convergence de l'algorithme. Par ailleurs, le taux de victoire est passé de 20 % à 85 % pour la tuile 256 et surtout 27 % pour la tuile 512, la tuile maximale pour la grille 3x3, comme l'illustre la courbe de performance présentée dans la figure 4.3.

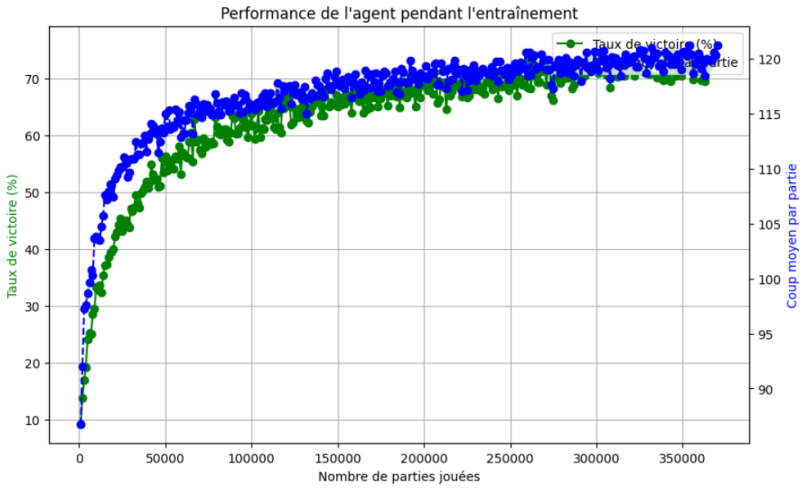

Figure 4.3: Courbe illustrant l'augmentation du taux de victoire après optimisation du code et amélioration du système de récompense.

Une métrique complémentaire, le nombre de coups par partie, a également été suivie pour affiner l'évaluation du modèle. Toutefois, cette métrique se comporte de manière similaire au taux de victoire, apportant ainsi peu d'informations additionnelles sur la performance globale de l'agent.

Le graphique de gauche montre la distribution des scores finaux. On peut voir que cette distribution présente une concentration marquée dans la plage élevée, indiquant que l'agent atteint souvent un niveau de jeu avancé avant la fin de la partie. En parallèle, le graphique de droite montre la répartition des tuiles maximales. On remarque que les rangs 7 et 8 (respectivement 128 et 256) sont les plus fréquents, traduisant une performance solide, tandis que l'obtention de rang 9 (512) reste plus rare. Globalement, ces graphiques montrent que l'agent atteint régulièrement des scores élevés, associés à des tuiles maximales de rang 7, 8 et même 9.

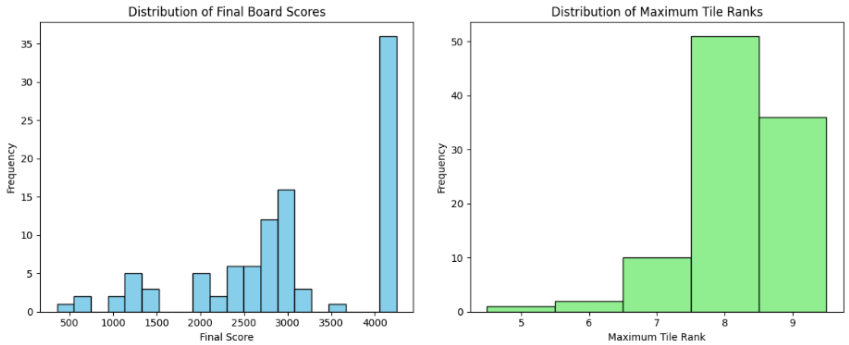

Figure 4.4: Répartition des scores finaux et des rangs de tuiles maximales atteints

4.6 Synthèse et perspectives

Les optimisations du code et l'amélioration du système de récompense ont permis de réaliser des avancées significatives dans notre projet de reinforcement learning appliqué au jeu 2048. En multipliant par cinq le nombre de simulations possibles et en améliorant notablement le taux de victoire, ces ajustements ont favorisé une convergence plus rapide

de l'algorithme et une stratégie d'agent plus stable. Ces résultats ouvrent la voie à des explorations futures, notamment l'adaptation de ces techniques à des configurations de plateau plus complexes ou l'intégration de méthodes d'apprentissage encore plus avancées.

4.7 Temporal Differencing for N-Tuple Networks

4.7.1 Motivation et Contexte

Après avoir exploité la méthode de Value Iteration (VI) pour établir une première politique dans un environnement à espace d'états relativement restreint, nous nous sommes penchés sur d'autres techniques d'apprentissage par renforcement à partir des ressources disponibles sur le web. Il est apparu que l'approche par *Temporal Difference Learning* (TD learning) combinée aux réseaux de N-tuple offrait une alternative particulièrement attractive pour le jeu 2048 [8, 6, 11].

Dans le cadre initial de notre projet, VI nous a permis de modéliser le problème de décision en utilisant l'équation de Bellman et d'obtenir une politique fonctionnelle sur des grilles de faible dimension. Cependant, l'explosion combinatoire de l'espace d'états lors de l'extension vers des grilles plus grandes et la nécessité de traiter des transitions stochastiques complexes rendent cette approche rapidement inapplicable à grande échelle. Nous avons décidé de creuser le fonctionnement de cette méthode afin de l'intégrer à notre jeu sur la grille 3x3, afin de comparer les résultats avec le VI [5].

4.7.2 Modélisation par N-Tuple Networks

Les réseaux de N-Tuple ont été initialement introduits pour fournir une méthode efficace d'évaluation des positions dans les jeux de plateau. Ils ont démontré leur efficacité dans des jeux tels que l'Othello [7, 10], où la complexité des positions nécessite une analyse approfondie des configurations locales. Dans ce contexte, les réseaux de N-Tuple permettent de capturer des motifs récurrents en segmentant le plateau en sous-ensembles de cellules, facilitant ainsi l'apprentissage et l'évaluation des positions.

Les réseaux de N-Tuple constituent une représentation fonctionnelle dans laquelle le plateau de jeu (ici, une grille 3×3) est segmenté en sous-ensembles de cellules, appelés tuples, dont les configurations locales sont directement associées à des poids appris. Pour chaque tuple de taille N, un tableau de 16^N poids (ou moins, compte tenu des symétries et du partage de paramètres) est utilisé pour stocker l'évaluation de chaque configuration possible des cellules couvertes. Ainsi, la fonction d'évaluation d'un état s s'exprime comme la somme des poids associés aux configurations observées dans chacun des tuples :

$$V(s) = \sum_{j=1}^{m} \sum_{k=1}^{K_j} w_{j,k}\, I_{\{s \in \text{feature } k \text{ du tuple } j\}},$$

où m est le nombre total de tuples et $\mathbb{I}$ la fonction indicatrice. La redondance induite par les symétries du plateau (rotations et réflexions) permet de réduire le nombre effectif de paramètres à apprendre tout en conservant une richesse d'expressions pour la représentation de la valeur de l'état.

Le N-Tuple Network se révèle particulièrement adapté au jeu 2048, notamment sur une grille 3×3, en raison de sa capacité à décomposer l'état global en configurations locales

pertinentes [13, 14]. Cette décomposition permet de capturer les régularités et les motifs locaux caractéristiques du jeu, tels que les combinaisons de tuiles susceptibles de fusionner. Par ailleurs, en tirant parti des symétries du plateau – rotations et réflexions – le modèle réduit efficacement le nombre de paramètres à apprendre tout en assurant une bonne généralisation sur des configurations équivalentes. L'agrégation des évaluations de ces motifs locaux offre ainsi une estimation fine de la qualité d'un état, ce qui s'avère particulièrement utile dans le cadre de l'apprentissage par différence temporelle (TD Learning).

Nous avons choisi de définir des motifs en sélectionnant des ensembles d'indices représentant des régions stratégiques du plateau, telles que les deux premières lignes ou les deux dernières colonnes. Cette approche nous permet de cibler des zones particulièrement pertinentes pour l'évaluation des états du jeu. Pour maximiser l'efficacité de notre modèle, nous avons exploité les isométries inhérentes au plateau. Ainsi, pour chaque motif, nous générons huit transformations (rotations et réflexions) afin d'assurer que des configurations identiques, bien que présentées sous différentes orientations, soient évaluées de manière cohérente. Ce choix réduit la redondance des paramètres tout en renforçant l'expressivité du modèle.

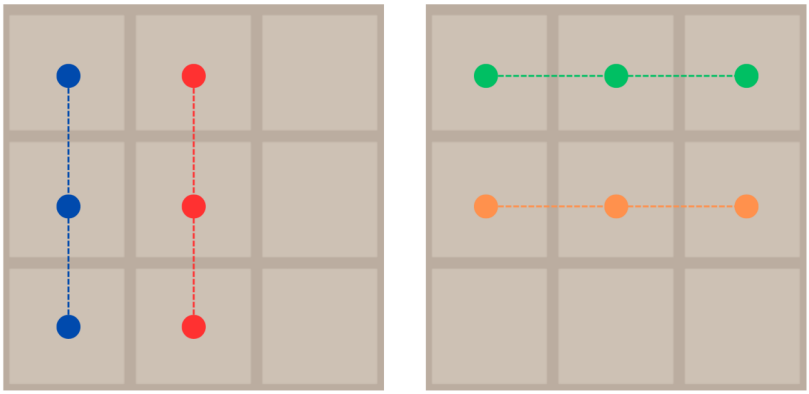

Figure 4.5: Tuples utilisés dans notre code

4.7.3 Introduction au Temporal Difference Learning

Le Temporal Difference Learning (TD Learning) est une méthode d'apprentissage par renforcement qui combine les idées de la programmation dynamique et de l'apprentissage supervisé [3, 1]. Il permet d'estimer la valeur des états sans nécessiter une connaissance préalable du modèle de l'environnement, ce qui en fait une approche particulièrement efficace pour des problèmes où la dynamique du système est inconnue ou difficile à modéliser.

Contrairement aux méthodes purement basées sur la récompense finale, comme Monte Carlo, qui mettent à jour les estimations uniquement à la fin d'un épisode, le TD Learning ajuste les valeurs des états de manière incrémentale, après chaque transition. Il repose sur le principe du bootstrapping, où une estimation est mise à jour à l'aide d'une autre estimation, réduisant ainsi la dépendance à des expériences complètes et permettant une convergence plus rapide.

4.7.4 Méthodologie du Temporal Difference Learning

La force du TD learning réside dans sa capacité à ajuster de manière itérative la fonction d'évaluation à partir de l'erreur entre deux évaluations consécutives [6, 11]. Dans le contexte des afterstates, le procédé consiste à calculer l'erreur de prédiction (ou TD error) pour un mouvement donné selon la formule :

$$\delta = r + V(s') - V(s),$$

où r représente la récompense immédiate obtenue lors du mouvement, s est l'afterstate courant (obtenu après avoir déplacé et fusionné les tuiles) et s' est l'afterstate suivant résultant de la prochaine action. Les poids associés aux features actives dans l'afterstate s sont ensuite ajustés selon la règle :

$$w_i \leftarrow w_i + \alpha\delta,$$

avec un taux d'apprentissage α choisi de manière appropriée (nous avons choisi $\alpha = 0.1$). Ce mécanisme de mise à jour « bootstrapping » permet à l'agent de corriger progressivement sa fonction d'évaluation en minimisant la différence entre les valeurs prédites et les récompenses réellement observées, assurant ainsi la convergence vers une estimation précise de la qualité des états.

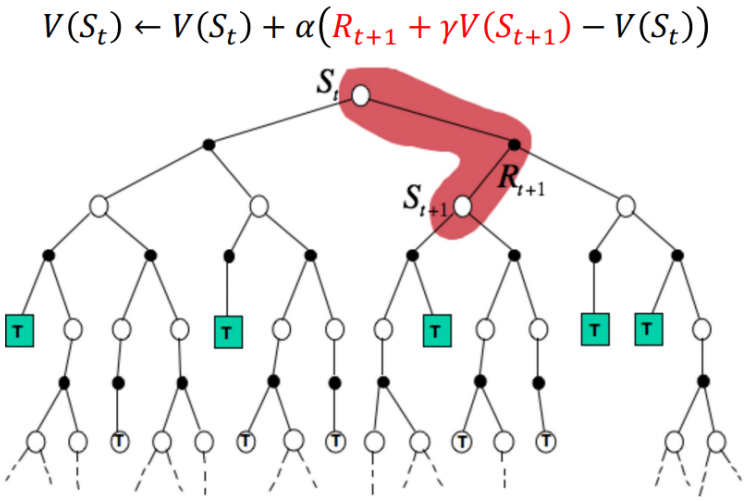

Figure 4.6: Méthode TD(0)

4.7.5 Résultat

4.7.6 Grille 3x3

Nous avons évalué l'agent TDL sur un total de 200000 parties, afin d'apprécier l'efficacité de la méthode TD appliquée aux N-tuple networks dans un environnement de jeu 2048 adapté à une grille 3×3. L'agent a démontré une capacité remarquable à maîtriser les configurations de base, atteignant systématiquement un taux de réussite de 100 % pour les tuiles de faible valeur, à savoir 2, 4, 8, 16 et 32. Cette constance traduit la solidité de la stratégie adoptée dans les situations initiales du jeu.

Au fur et à mesure que la complexité du jeu augmente, les résultats indiquent une légère décroissance de la performance. La tuile 64 est obtenue dans 99,9 % des cas, tandis que

la tuile 128 apparaît avec un taux de réussite avoisinant 98,8 %. Ces chiffres démontrent une robustesse encore notable dans la gestion des transitions intermédiaires. Cependant, pour la tuile 256, le taux de succès descend à 87,3 %, illustrant la montée en difficulté pour l'agent lorsque les configurations se compliquent. La performance se dégrade plus significativement pour la tuile 512, atteignant un taux de 33,0 %, ce qui met en lumière les défis associés aux niveaux de jeu les plus élevés.

Ci dessous, trois plots, permettant de comprendre les résultats. La distribution des scores finaux, illustrée par le box plot, montre un étalement marqué, avec quelques parties atteignant des valeurs nettement plus élevées. Par ailleurs, le violin plot des tuiles maximales indique que l'agent converge souvent vers un certain rang de tuile, tout en présentant une dispersion notable. Enfin, le nuage de points entre score et nombre de coups révèle une tendance : plus une partie dure, plus le score augmente. De plus, la coloration selon la tuile maximale confirme que les meilleurs scores coïncident avec des tuiles de rang supérieur.

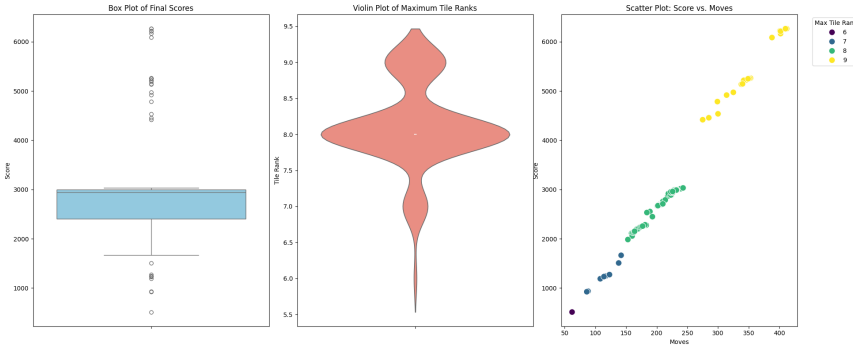

Figure 4.7: Analyse statistique des performances de l'agent

4.7.7 Grille 4x4

Enfin, nous avons réussi à créer un agent qui joue sur une grille 4x4 avec le temps qu'il nous reste. Pour évaluer l'efficacité de la méthode TD sur une grille 4x4 dans un environnement 2048, l'agent TDL a été testé sur 100000 parties. Comme pour la grille 3x3, les tuiles de faible valeur (2, 4, 8, ..., 512) sont systématiquement obtenues avec un taux de réussite de 100 %.

Pour des configurations plus complexes, la tuile 1024 apparaît dans environ 97,5 % des cas et la tuile 2048 dans près de 85,8 %. La progression vers des rangs supérieurs se fait avec la tuile 4096 obtenue dans environ 46,3 % des parties, tandis que la tuile 8192 n'est atteinte que dans environ 2,8 % des cas.

Le box plot confirme ces observations.

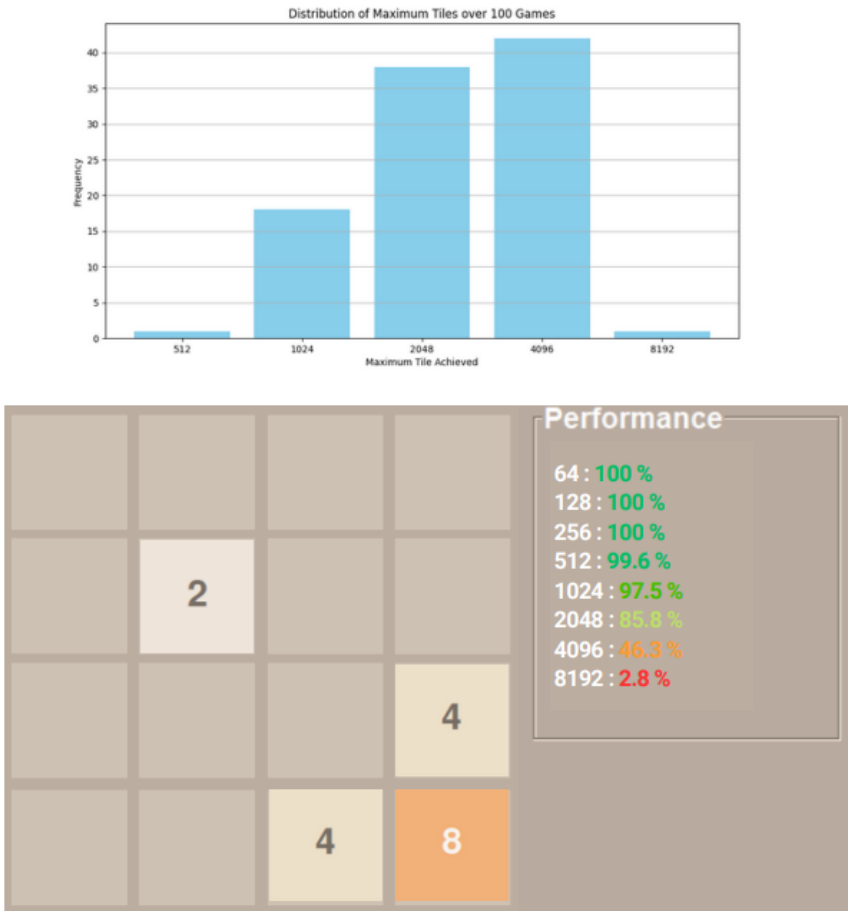

Figure 4.8: Résultat du jeu 4x4 sur 100 parties.

Ces résultats mettent en évidence la solidité de l'agent dans les premières phases du jeu, où il parvient à optimiser efficacement ses décisions. Toutefois, la baisse progressive des performances à mesure que la complexité augmente souligne les limites actuelles de la méthode. Bien que l'agent affiche une robustesse notable jusqu'à des tuiles élevées, la chute plus marquée des taux de réussite pour les tuiles supérieures révèle deux potentielles choses. Tout d'abord, plus les tuiles sont élevées sur la grille et plus l'agent aura moins de manœuvre. De plus, il y a des marges d'amélioration, telle que le nombre d'épisode ou l'utilisation d'autre tuple. Ainsi, des ajustements stratégiques et des optimisations complémentaires pourraient être envisagés afin d'accroître la capacité de l'agent à atteindre des scores plus élevés de manière plus régulière.

4.8 Démonstrateur

4.8.1 Introduction

Cette interface graphique permet de jouer au 2048, de choisir différentes configurations (grille 3×3 ou 4×4) et de bénéficier de l'aide d'un agent entraîné par un modèle d'apprentissage par renforcement.

4.8.2 Écran d'accueil

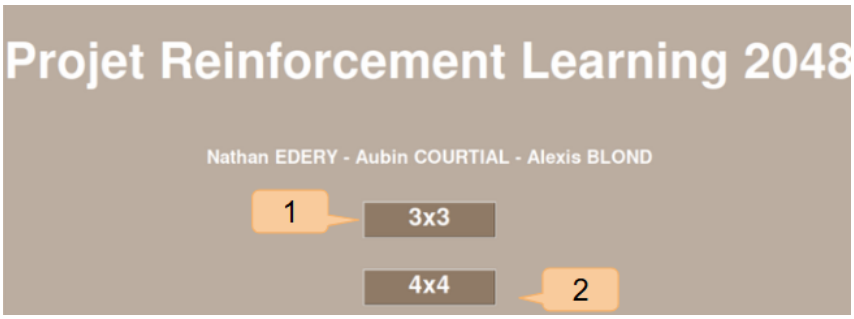

Figure 4.9: Écran d'accueil

- **Élément 1 :** Bouton pour lancer une partie de 2048 sur une grille 3×3
  Possibilité de bénéficier de l'aide de différents agents intelligents.
- **Élément 2 :** Bouton pour lancer une partie de 2048 sur une grille 4×4
  Possibilité de bénéficier de l'aide d'un agent RL entraîné avec la méthode Temporal Differencing for N-Tuple Networks

4.8.3 Interface de jeu 2048 sur grille 3×3 ou 4×4 assistée par un agent

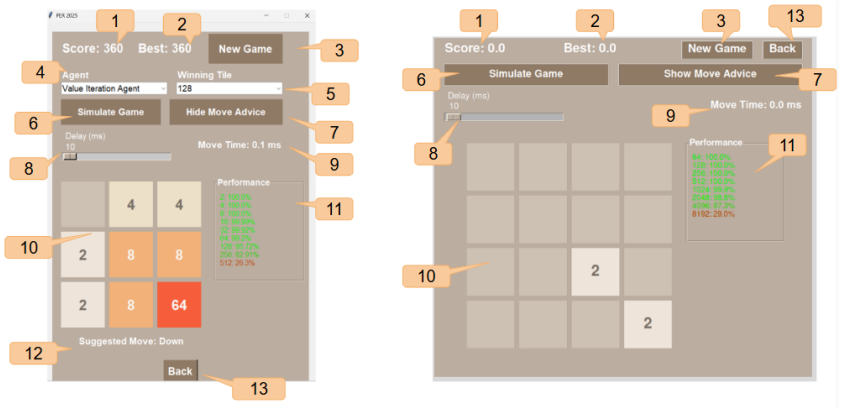

Figure 4.10: démonstrateur 3*3

- **Élément 1 :** Score de la partie en cours
- **Élément 2 :** Meilleur score enregistré
- **Élément 3 :** Bouton « Relancer la partie »
- **Élément 4 :** Curseur déroulant pour le choix de l'agent
  - Permet de sélectionner l'agent parmi Random Agent, Value Iteration Agent et Expectimax Agent.
- **Élément 5 :** Curseur déroulant pour choisir la tuile objectif (16, 32, 64, . . .)
- **Élément 6 :** Bouton « Jouer de manière autonome »

  - Lance automatiquement les mouvements générés par l'agent sélectionné.

- **Élément 7 :** Bouton pour afficher ou masquer l'aide
- **Élément 8 :** Curseur horizontal de vitesse des mouvements
  - Contrôle la vitesse des mouvements lorsque la partie est jouée automatiquement par l'agent.
- **Élément 9 :** Vitesse de mouvement pour l'agent Expectimax
- **Élément 10 :** Grille de jeu
- **Élément 11 :** Performance de l'agent sélectionné
- **Élément 12 :** Mouvement suggéré par l'agent
- **Élément 13 :** Bouton Retour à l'accueil

Chapter 5

Difficultés rencontrées

5.1 Deep Q-Learning

Dans le cadre de notre projet, nous avons tenté d'appliquer le Deep Q-Learning (DQN) pour aborder l'immense espace d'états du jeu 2048, en utilisant un réseau de neurones pour approximer la fonction $Q(s, a)$ [15]. Cette approche, bien que prometteuse sur le plan théorique, s'est heurtée à plusieurs obstacles majeurs :

- **Faible signal de récompense et atténuation par le discount :** Dans 2048, la majorité des actions, notamment les fusions de petites tuiles (2 et 4), génèrent des récompenses quasi nulles. L'équation de Bellman se simplifie souvent en

$$\delta \approx \gamma \max_{a'} Q(s', a') - Q(s, a),$$

  ce qui, combiné à un facteur de discount γ (par exemple $\gamma = 0.95$), réduit considérablement l'impact des récompenses obtenues après de longues séquences d'actions [16]. Une chaîne idéale de fusions pour atteindre une tuile élevée (comme 256) montre ainsi comment un signal faible se voit encore atténué au fil des étapes.

- **Explosion combinatoire de l'espace d'états :** Pour un plateau $n \times n$ avec m valeurs possibles par case, le nombre total d'états est de l'ordre de m^{n^2}. Cette complexité combinatoire, accentuée par la chaîne de fusion où la majorité des transitions ne génère qu'un faible signal, complique fortement l'approximation de $Q(s, a)$ par un réseau de neurones [16].

- **Limitations du DQN par rapport à une approche de type value iteration :** Une méthode antérieure, s'appuyant sur une table de valeurs calculée précisément pour chaque état, gérait mieux les faibles signaux de récompense. En revanche, le DQN introduit des erreurs de généralisation et d'estimation. L'augmentation du facteur de discount pour compenser l'atténuation du signal mène à une amplification des erreurs de bootstrapping et à une instabilité de l'apprentissage, surtout dans un environnement aussi stochastique que le 2048 [15].

Synthèse : L'application du Deep Q-Learning au 2048 se heurte principalement à deux obstacles : un espace d'états combinatoire massif et des récompenses faibles fortement atténuées par le facteur de discount. Ces défis limitent l'efficacité du signal d'apprentissage et rendent la convergence du DQN particulièrement difficile. Cette expérience suggère que des approches alternatives ou hybrides pourraient être mieux adaptées pour exploiter la complexité inhérente du jeu.

5.2 Défis avec les méthodes Monte Carlo et MCTS

Nous avons également exploré les méthodes Monte Carlo (MC) et le Monte Carlo Tree Search (MCTS), motivés par leur pertinence dans les cours et leur succès dans des jeux comme le Go ou les échecs. Cependant, ces approches se sont heurtées à des difficultés spécifiques au 2048. La méthode Monte Carlo first-visit, qui estime la valeur d'un état en moyennant les retours cumulés obtenus après chaque première visite de cet état sur de multiples épisodes, semblait prometteuse en théorie [16]. Le retour G_t pour un état s_t à l'instant t est calculé comme :

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T,$$

où T est la fin de l'épisode. Cependant, dans le 2048, la faible densité des récompenses significatives (principalement obtenues lors de fusions de tuiles élevées) et la taille de l'espace d'états ont rendu cette méthode inefficace. Le nombre d'épisodes requis pour obtenir des estimations fiables de $V(s)$ était prohibitif, et nos tentatives limitées ont montré une convergence extrêmement lente, probablement due à un entraînement insuffisant par manque de temps [16].

Le MCTS, quant à lui, construit un arbre de recherche en simulant des parties depuis l'état courant, en équilibrant exploration et exploitation via des critères comme l'UCT [9] :

$$UCT = \overline{X_i} + C\sqrt{\frac{\ln N}{n_i}},$$

où $\overline{X_i}$ est la valeur moyenne du nœud i, N le nombre de visites du nœud parent, n_i celui du nœud i et C une constante d'exploration. Bien que cette approche excelle dans des jeux déterministes [2], le caractère stochastique du 2048 complique son application. L'apparition aléatoire des tuiles après chaque coup génère de multiples branches probabilistes par nœud, augmentant la complexité computationnelle et diluant la précision des estimations. De plus, les récompenses retardées et souvent faibles dans le 2048 limitent la propagation d'informations utiles à travers l'arbre. Contrairement aux échecs, où des heuristiques positionnelles facilitent l'exploration, le 2048 manque d'indicateurs clairs pour guider les simulations, ce qui réduit l'efficacité du MCTS et explique nos résultats décevants avec cette méthode.

5.3 Contraintes de temps et de ressources

Enfin, les contraintes temporelles et matérielles ont fortement influencé notre projet. Réalisé dans un cadre académique avec des cours en parallèle, le temps disponible pour explorer et optimiser des approches telles que Monte Carlo ou MCTS a été limité [16]. Ces méthodes, nécessitant des phases d'entraînement prolongées pour produire des résultats significatifs, n'ont pas pu être pleinement exploitées, accentuant l'écart entre nos ambitions initiales et les résultats obtenus. Par exemple, augmenter le nombre d'épisodes pour Monte Carlo ou de simulations pour MCTS aurait pu améliorer leurs performances, mais cela dépassait les capacités du calendrier imparti. De plus, nous avons choisi de privilégier d'autres approches, comme le Temporal Differencing, qui ont montré des résultats plus concluants dans des travaux antérieurs [8].
Le choix de Python comme langage de programmation a également posé des défis. Fort de notre familiarité avec ce langage et de l'accès à des bibliothèques comme PyTorch, Python a facilité le prototypage rapide, notamment pour le DQN [15]. Cependant, sa

faible performance pour des tâches intensives – telles que les simulations ou les calculs matriciels répétés – a ralenti les processus d'apprentissage. Par exemple, un entraînement nécessitant 10^6 itérations pouvait prendre plusieurs heures en Python, alors qu'un langage comme C++ aurait permis de réduire ce temps grâce à une exécution plus rapide des boucles et des opérations bas niveau [4]. Rétrospectivement, adopter C++ aurait pu être bénéfique, mais le temps requis pour adapter notre code et maîtriser ce langage dans le contexte du Reinforcement Learning aurait introduit d'autres contraintes. Ce dilemme entre rapidité de développement et efficacité computationnelle a ainsi amplifié les limitations imposées par nos ressources, rendant certains aspects du projet plus difficiles à finaliser.

Chapter 6

Conclusion et perspectives

La réalisation de ce projet nous a permis d'acquérir une compréhension approfondie de l'apprentissage par renforcement appliqué au jeu 2048, en alliant théorie et pratique de manière significative. Nous avons pu mettre en œuvre et tester différentes méthodes, notamment la Value Iteration et le Temporal Difference Learning couplé aux N-Tuple Networks, nous offrant ainsi l'opportunité d'explorer en profondeur les mécanismes de prise de décision dans des environnements complexes.

Au cours de cette étude, malgré des difficultés initialement sous-estimées, les résultats obtenus ont finalemen comblés nos attentes, l'obtention de la fameuse tuile 2048 par un agent de reinforcement learning ! Nous avons été surpris d'obtenir nos meilleures performances en adoptant des techniques alternatives, contrairement à ce que nous avions envisagé avec le Deep Q-Learning, les réseaux de neurones ne se sont pas révélés être la solution miracle dans notre contexte. Ce constat souligne l'importance de diversifier les approches pour appréhender efficacement la complexité du problème.

Par ailleurs, la rigueur de la modélisation mathématique a été primordiale pour comprendre les enjeux et sélectionner les méthodes adaptées. La formalisation du jeu sous forme de processus de décision markovien nous a permis d'identifier avec précision les points forts et les limites de chaque approche, et a orienté efficacement le développement de nos algorithmes.

L'optimisation du code représente un autre enseignement majeur de ce projet. L'utilisation judicieuse de techniques telles que Numba, la représentation par bitboard et des constructions idiomatiques en Python nous a démontré qu'une optimisation précoce et rigoureuse peut considérablement réduire le temps de calcul et ouvrir la voie à des simulations plus rapides et nombreuses.

Enfin, l'expérience acquise a mis en lumière l'importance d'effectuer des recherches approfondies sur l'état de l'art afin d'éviter de réinventer la roue. Une bibliographique rigoureuse permet de concentrer nos efforts sur des aspects réellement innovants et spécifiques au problème traité.

En perspective, plusieurs axes d'amélioration méritent d'être explorés pour donner une nouvelle dimension à ce projet. Tout d'abord, adapter et optimiser nos méthodes pour des grilles de plus grande taille (par exemple, passer de 4×4 à 5×5) représenterait un défi stimulant, nécessitant une révision de notre modélisation pour gérer l'explosion combina-

toire des états. Par ailleurs, il serait intéressant d'enrichir notre approche en s'appuyant sur des techniques issues d'autres domaines, telles que l'apprentissage par transfert ou l'optimisation par métaheuristiques, afin d'améliorer la performance et la robustesse de notre système. Un autre axe à considérer serait l'optimisation algorithmique par le recours à des langages compilés comme le C++, qui permettrait de réduire significativement le temps de calcul et de mieux exploiter les ressources matérielles disponibles. Enfin, approfondir l'analyse théorique des conditions de convergence et des limites de nos méthodes pourrait offrir des perspectives d'application dans d'autres problématiques complexes, au-delà du cadre ludique du jeu 2048.

En somme, ce projet a permis de développer de solides compétences en modélisation, en optimisation de code et en techniques d'apprentissage par renforcement, tout en ouvrant des perspectives prometteuses pour des travaux futurs, où l'innovation et la rigueur scientifique demeureront les clés du succès.

Bibliography

[1] Simon M. Lucas Aisha A. Abdullahi. Temporal difference learning with interpolated n-tuple networks: initial results on pole balancing. *IEEE Xplore*, 2010.

[2] Cameron B. Browne, Ewan Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Simon Tavener, Diego Perez, Stamatios Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[3] Martin C. Smith Donald F. Beal. First results from using temporal difference learning in shogi. *Lecture Notes in Computer Science ((LNCS,volume 1558))*, 1999.

[4] Micha Gorelick and Ian Ozsvald. *High Performance Python*. O'Reilly Media, 2014.

[5] Hung Guei. On reinforcement learning for the game of 2048. *arXiv preprint arXiv:2212.11087*, 2023.

[6] I-Chen Wu Hung Guei, Lung-Pin Chen. Optimistic temporal difference learning for 2048. *arXiv preprint arXiv:2111.11090*, 2021.

[7] Wojciech Jaśkowski. Systematic n-tuple networks for position evaluation: Exceeding 90% in the othello league. *arXiv preprint arXiv:1406.1509*, 2014.

[8] Wojciech Jaśkowski. Mastering 2048 with delayed temporal coherence learning, multi-stage weight promotion, redundant encoding and carousel shaping. *arXiv preprint arXiv:1604.05085*, 2016.

[9] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European Conference on Machine Learning*, pages 282–293. Springer, 2006.

[10] Marcin Szubert Krzysztof Krawiec. Learning n-tuple networks for othello by coevolutionary gradient search. *Institute of Computing Science, Poznan University of Technology*, 2011.

[11] Chu-Hsuan Hsueh Chia-Chuan Chang Chao-Chin Liang Han Chian Kun-Hao Yeh, I-Chen Wu. Multi-stage temporal difference learning for 2048-like games. *arXiv preprint arXiv:1606.07374*, 2016.

[12] S. Lam, A. Pitrou, and S. Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015.

[13] Wojciech Jaśkowski Marcin Szubert. Temporal difference learning of n-tuple networks for the game 2048. *IEEE Xplore*, 2014.

[14] K. Matsuzaki. Systematic selection of n-tuple networks with consideration of inter-influence for game 2048. *Proc. 21st Int. Conf. Technol. Appl. Artif. Intell. (TAAI 2016)*, 2016.

[15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Cory Beattie, Aadil Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[16] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018.

[17] Chess Programming Wiki. Bitboards, n.d.