Faculté des Sciences et Ingénierie - Sorbonne université

Master Informatique - parcours DAC

PLDAC: Projet Logiciel DAC

Rapport de Projet

Algorithmes de renforcement génériques pour les jeux

Réalisé par :

Eliott Rakotondramanitra
Raphaël Renard
Luc Salvon

Supervisé par :

Nicolas Baskiotis
Jean-Noël Vittaut

Mai 2024

CONTENTS

CHAPTER 1

INTRODUCTION

Dans ce projet, nous explorons l'application des algorithmes de renforcement pour résoudre des jeux proposés par la plateforme CodinGame. L'apprentissage par renforcement (RL) est une branche de l'intelligence artificielle qui traite de la façon dont un agent doit prendre des actions dans un environnement afin de maximiser une notion cumulative de récompense. Notre objectif est d'implémenter et de comparer plusieurs algorithmes de RL pour deux jeux distincts : Ultimate Tic-Tac-Toe et Mad Pod Racing.

Nous allons implémenter et évaluer les algorithmes suivants :

1. Monte Carlo Tree Search (MCTS) : Algorithme de recherche basé sur des simulations aléatoires pour prendre des décisions optimales dans les jeux à états discrets comme Ultimate Tic-Tac-Toe.
2. Q-Learning : Algorithme d'apprentissage par renforcement hors ligne qui apprend une fonction de valeur d'action représentant la qualité d'une action dans un état donné, applicable aux environnements discrets.
3. Deep Q-Network (DQN) : Extension du Q-Learning utilisant des réseaux de neurones profonds pour gérer des espaces d'état plus larges et plus complexes, adapté aussi bien aux environnements discrets qu'aux environnements continus.
4. REINFORCE : Algorithme basé sur des *policy gradients* pour les environnements continus, permettant d'apprendre directement une politique sans nécessiter une fonction de valeur explicite.

Pour valider et comparer nos algorithmes, nous suivrons le protocole expérimental suivant :

- Nous commencerons par coder des environnements de simulation en interne pour chaque jeu. Ces environnements serviront à entraîner nos agents de manière contrôlée et à ajuster les hyperparamètres des algorithmes.
- Après avoir obtenu des performances satisfaisantes en environnement interne, nous déploierons nos agents sur la plateforme CodinGame. Cela permettra de tester nos solutions dans un cadre réel et compétitif.
- Les performances de nos agents seront comparées en utilisant les classements obtenus sur CodinGame. Nous établirons également une comparaison avec des heuristiques qui utilisent des stratégies pré-programmées et non apprises, pour évaluer l'efficacité relative de nos algorithmes de RL.

En suivant ce protocole, nous pourrons non seulement tester la capacité de nos algorithmes à apprendre et à s'adapter à différents types de jeux, mais aussi mesurer leurs performances de manière objective dans un environnement compétitif.

1.1 Contexte historique

L'apprentissage par renforcement – ou *reinforcement learning* (RL) en anglais – a évolué sur une période de 70 ans, permettant aux algorithmes de prendre des décisions basées sur la méthode essai-erreur (*trial and error*) dans des environnements donnés.

Avant 1992, la recherche en RL poursuivait trois fils conducteurs concurrents : la méthode essai-erreur, le contrôle optimal et le *Temporal Difference* (TD) *learning*. Ces trois axes, initialement indépendants, se sont fusionnés dans les années 1980 pour former la base du RL moderne.

L'idée d'apprendre par essai et erreur vient de l'observation de l'intelligence animale, et consiste à associer à un résultat une récompense, positive ou négative, et à la redistribuer parmi les nombreuses décisions qui ont contribué à produire le résultat. En essayant de maximiser la récompense, nous apprenons comment arriver au meilleur résultat.

Bellman a développé une méthode spécifique au RL qui définit dynamiquement une équation fonctionnelle, en utilisant l'état d'un système dynamique, et renvoie une fonction de valeur optimale. Cette fonction de retour optimale est appelée équation de Bellman. Bellman a ensuite introduit le processus de décision markovien, qu'il définit comme une version stochastique discrète du problème de contrôle optimal.

S'inspirant de la différentiation mathématique, l'apprentissage par différence temporelle vise à dériver une prédiction à partir d'un ensemble de variables connues en partant de l'estimation actuelle de la fonction de valeur.

Le travail de Sutton dans la théorie de l'apprentissage animal a conduit à des développements significatifs en RL, y compris l'architecture acteur-critique. Ces développements ont ouvert la voie à de nouvelles recherches, conduisant au développement du Q-learning par Watkins en 1989.

En 1992, le programme TD-Gammon de Tesauro a atteint le "niveau maître" dans le jeu de Backgammon. Cela a poussé les chercheurs à essayer d'appliquer le RL aux jeux d'échecs et de Go, aboutissant à la victoire d'AlphaGo en 2016 qui utilisait des techniques avancées telles que les simulations Monte Carlo et l'optimisation bayésienne. Les algorithmes de RL ont également été appliqués à des jeux d'arcade, démontrant leur adaptabilité et leurs capacités d'apprentissage dans des environnements dynamiques. Ces avancées marquent un progrès significatif en RL, démontrant son potentiel dans divers domaines, des jeux de société aux applications complexes dans le monde réel.

Dans ce rapport, nous nous intéressons à quatre algorithmes de RL.

Développé par Watkins en 1989, le Q-Learning est l'un des premiers algorithmes de RL. Il met à jour de manière itérative la Q-table pour estimer la valeur des paires état-action, permettant aux agents de prendre des décisions optimales. Plus précisément, le Q-learning apprend une fonction de valeur notée Q. Cette fonction Q estime le gain potentiel, c'est-à-dire la somme des récompenses $Q(s, a)$ sur le long terme, apportée par le choix d'une certaine action a dans un certain état s en suivant une politique optimale. Cette fonction Q s'appelle fonction de valeur actions-états, puisqu'elle prend comme argument un état s et une action a. Un des points forts du Q-learning est qu'il permet de comparer les récompenses probables de prendre les actions accessibles sans connaître le processus de décision markovien qui modélise le système, il ne fait qu'interagir avec l'environnement.

Proposé par Ronald J. Williams en 1992, REINFORCE est un algorithme basé sur les gradients, utilisé pour entraîner des politiques stochastiques dans des espaces d'action continus. Il a été fondamental dans la formation de réseaux neuronaux pour les tâches de RL. Un réseau de neurones est utilisé comme une politique de prise de décision, c'est-à-dire comme stratégie guidant l'agent à prendre certaines actions en fonction de l'état de l'environnement. L'algorithme met à jour les paramètres du réseau neuronal en fonction des récompenses obtenues, dans le but d'améliorer la probabilité d'actions conduisant à des récompenses cumulées plus élevées.

Introduit dans les années 2000, la recherche arborescente Monte Carlo ou *Monte Carlo tree search* (MCTS) et sa capacité à simuler des trajectoires possibles dans un arbre de décision par

échantillonnage aléatoire ont considérablement fait progresser le RL dans des environnements complexes. L'algorithme MCTS est un algorithme qui explore l'arbre des possibles. La racine est la configuration initiale du jeu. Chaque nœud est une configuration et ses enfants sont les configurations suivantes. MCTS conserve en mémoire un arbre qui correspond aux nœuds déjà explorés de l'arbre des possibles. L'objectif est l'analyse des coups les plus prometteurs, en élargissant l'arbre de recherche sur la base d'un échantillonnage aléatoire de l'espace de recherche reposant sur de nombreuses simulations. Dans chaque simulation, le jeu se joue jusqu'à la fin en sélectionnant des mouvements au hasard. Le résultat final du jeu est ensuite utilisé pour pondérer les nœuds dans l'arbre de jeu afin que les meilleurs nœuds soient plus susceptibles d'être choisis lors des simulations futures.

Introduit par DeepMind en 2015, le Deep Q Network (DQN) combine le Q-Learning avec des réseaux neuronaux profonds. Il peut apprendre directement à partir d'entrées sensorielles brutes, comme les pixels d'un écran de jeu, et a obtenu un succès remarquable dans divers jeux Atari. Le DQN utilise un réseau de neurones profond afin d'approximer la fonction Q. Contrairement au Q-learning qui utilise une Q-table pour stocker ces valeurs, le DQN utilise donc le réseau de neurones pour généraliser sur l'ensemble de l'espace d'état sans stocker explicitement chaque paire état-action. Il se base sur une technique de "replay" : les expériences passées (états, actions, récompenses et nouveaux états) sont stockées dans une mémoire de replay. Durant l'entraînement, des mini-batchs de ces expériences sont aléatoirement échantillonnés pour mettre à jour les poids du réseau. Le DQN utilise également un système de réseau cible et réseau principal afin d'aider à stabiliser les prédictions des valeurs de la fonction Q. Le réseau principal est utilisé pour sélectionner l'action et évaluer sa qualité, mais les mises à jour du réseau se font en comparant les sorties du réseau principal à celles du réseau cible, qui est une version légèrement plus ancienne du réseau principal. Le réseau cible est mis à jour périodiquement avec les poids du réseau principal.

Figure 1.1: Illustration d'une partie de Mad Pod Racing. Les structures circulaires sont les checkpoints numérotés selon l'ordre dans lequel il faut les valider.

1.2 Présentation du problème

1.2.1 CodinGame

CodinGame est une plateforme d'apprentissage et de compétition en ligne sur laquelle il est possible de participer à des "combats de bots", une fonctionnalité qui permet de créer des bots et de les faire s'affronter dans des jeux multijoueurs en temps réel. Le programmeur peut choisir le langage de programmation qu'il veut utiliser. Nous avons choisi Python.

À la fin de chaque combat, les utilisateurs peuvent analyser les performances de leurs bots en examinant les statistiques fournies par la plateforme telles que le nombre de victoires ou le classement du bot.

Cependant, la plateforme est très contraignante. Le code ne doit pas faire plus de 100 000 caractères et chaque tour doit se faire dans un temps imparti. Le bot doit pouvoir jouer son premier coup en moins de 1000ms. Puis les tours suivants doivent prendre moins de 100ms pour le jeu Ultimate Tic-Tac-Toe et 75ms pour Mad Pod Racing.

1.2.2 Jeux étudiés

Parmi les jeux proposés par la plateforme CodinGame, nous avons choisi de nous pencher sur deux jeux différents : Ultimate Tic-Tac-Toe et Mad Pod Racing. En effet, ces deux jeux sont de types bien distincts, ce qui est intéressant pour notre analyse. Ultimate Tic-Tac-Toe est un jeu de plateau et de stratégie à états discrets. Au contraire, Mad Pod Racing est un jeu de course à états continus.

Pour Ultimate Tic-Tac-Toe, une approche discrète basée sur des états bien définis et des décisions stratégiques est donc appropriée, tandis que pour Mad Pod Racing, une approche continue utilisant des algorithmes capables de gérer des actions et états continus est nécessaire. Nous mettrons notamment en évidence dans la suite le fait que les méthodes continues marchent mal sur Ultimate Tic-Tac-Toe et bien sur Mad Pod Racing.

- on peut également expliquer que d'autres jeux ont été choisis mais non retenus et pourquoi

Mad Pod Racing

Mad Pod Racing [2] est un jeu de course qui peut se jouer à plusieurs joueurs, il se déroule sur une carte où chaque joueur contrôle un pod et où le circuit de la course sera déterminé par des points de passages (check-points) placés aléatoirement. Pour terminer un tour il faudra que le pod passe chaque check-points dans l'ordre et revienne au point de départ. Ainsi la condition de victoire pour remporter la course est de franchir le check-point de départ lors du dernier tour.

L'environnement sera donc la carte, les pods et les différents check-points. Contrairement aux jeux de plate-forme, les états sont représentés par la position du pod (coordonnées x et y),

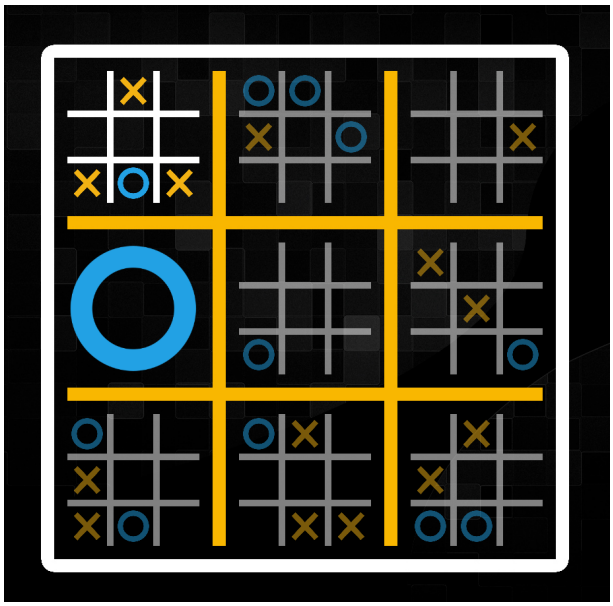


Figure 1.2: Illustration d'une partie d'Ultimate Tic-Tac-Toe. Le joueur courant doit placer sa marque dans la sous-grille supérieure gauche. Le joueur rond a déjà remporté la sous-grille centre droite.

la position du prochain check-point, la distance entre le pod et le prochain check-point et enfin l'angle en degrés entre l'orientation du pod et la direction que le pod doit viser pour atteindre le prochain check-point. Les actions possibles sont de pouvoir contrôler la destination du pod à l'aide d'une position (x et y) et la puissance du pod (allant de 0 à 100). La zone de jeu à l'écran où se situeront les checkpoints a une taille de 16 000×9 000 pixels, en notant que les pods peuvent dépasser cette zone et explorer une zone virtuellement infinie. De ce fait, une approche continue semble plus adaptée à la résolution de ce problème qu'une approche discrète au vu du nombre considérable d'états possibles.

Pour jouer à ce jeu sans utiliser de RL, on peut suivre une heuristique simple : si l'angle avant le prochain checkpoint dépasse 45 degrés, réduire la puissance en soustrayant à la puissance initiale la différence entre cet angle et une constante pour mieux aligner le pod, et si le pod est à moins de 3000 unités du checkpoint en distance, diminuer progressivement la puissance selon la proximité pour assurer une arrivée controlée.

Ultimate Tic-Tac-Toe

Ultimate tic-tac-toe [4] est un jeu de plateau à deux joueurs composé de neuf grilles de tic-tac-toe (ou morpion en français) arrangées en une grille 3 × 3. Comme dans une partie de morpion classique, les joueurs marquent chacun leur tour une case d'un rond ou d'une croix (chacun a son propre symbole), la sous-grille dans laquelle le joueur peut placer une marque est définie par l'emplacement de la marque adverse (un joueur qui marque la case au centre d'une sous-grille force son adversaire à jouer dans la sous-grille au centre). S'il n'est pas possible pour le joueur de jouer dans la sous-grille définie (grille déjà gagnée ou toutes les cases occupées) alors le joueur peut décider de la grille dans laquelle il jouera. Pour gagner une petite grille, il faut aligner trois croix (ou ronds) verticalement, horizontalement ou en diagonale. Cependant, il ne suffit pas de gagner une petite grille. Il faut gagner trois grilles alignées verticalement, horizontalement ou en diagonale. C'est donc une sorte de mise en abîme de partie de morpion (voir figure 1.2).

Ainsi, l'environnement du jeu sera représenté par les neuf grilles, remplies avec des ronds et des croix. Les actions possibles sont les coordonnées des cases encore vides dans la petite grille déterminée par le coup de l'adversaire (ou toutes les cases vides qui ne sont pas sur des grilles déjà gagnées dans le cas où jouer sur la petite grille prévue n'est pas possible).

Pour jouer à ce jeu sans utiliser de RL, on peut suivre une heuristique simple : s'il est possible de compléter un alignement on le complète, sinon, si l'adversaire a déjà aligné deux symboles,

on le bloque, et si on n'est dans aucun des deux cas, on joue au hasard.

Jeux non retenus

Nous avions considéré deux autres jeux : Seabed Security [3] et Ghost In The Cell [1]. Cependant, ces deux jeux ont de nombreux paramètres à prendre en compte dans l'état, et donc difficilement simplifiables. Nous avons donc décidé de maintenir notre attention sur des jeux plus simples sur lesquels il serait possible d'appliquer des algorithmes de base.

CHAPTER 2

PROTOCOLE EXPÉRIMENTAL

Un des premiers axes du projet est de pouvoir implémenter des algorithmes génériques qui vont pouvoir être utilisés sur plusieurs jeux (plate-forme, course, cartes, combats...). Pour cela nous avons suivi un protocole qui nous a permis d'implémenter nos algorithmes sur nos deux différents jeux.

2.1 Implémentation d'environnements internes

Nous avons développé des environnements de simulation internes pour nos deux jeux spécifiques : Ultimate Tic-Tac-Toe et Mad Pod Racing. Ces environnements répliquent la logique (interaction utilisateur et jeu) et les règles des versions existantes sur CodinGame, mais sont conçus pour surmonter les limitations de la plateforme comme les contraintes de temps d'exécution et les limites de longueur de code. Ces environnement définissent les états, les actions disponibles et offrent la possibilité de faire des simulations de parties sans contraintes.

Pour MadPodRacing, quelques simplifications ont été de mise, ayant privilégié un environnement plus simple, tout en étant compatible avec la version sur codingame. On ne conserve ainsi qu'un pod par joueur, en retirant les collisions. Les physiques du jeu n'étant pas clairement spécifiées dans codingame, nous nous sommes appuyés sur un site tiers [7] les spécifiant, nous permettant ainsi d'avoir un environnement très similaire à celui de codingame.
Afin d'entraîner les divers algorithmes, nous avons aussi décidé de créer un environnement encore plus simple, dans lequel un pod, de vitesse et orientation aléatoire, doit aller à un checkpoint placé à un endroit aléatoire de la zone de jeu.
Nous devons également traiter la représentation de l'état pour un bot, ainsi que d'éventuelles discrétisations des états/actions pour les algorithmes non continus.

2.2 Validation sur Cartpole

Avant de tester nos algorithmes sur nos environnements, nous avons commencé par les évaluer sur l'environnement Cartpole de Gym (OpenAI) qui est un environnement souvent utilisé pour tester des algorithmes de renforcement de par sa simplicité. En effet, l'objectif est de maintenir un poteau vertical sur un chariot qui peut se déplacer horizontalement. Tester nos algorithmes sur Cartpole nous permet de vérifier leurs capacités à apprendre des politiques de base avant de les tester sur nos environnements plus complexes.

La version de cartpole de la bibliothèque gym proposant uniquement des actions continues, nous avons utilisé une adaptation en version continue [5] pour tester l'algorithme policy gradient.

2.3 Entraînement et Analyse des performances en interne

Après avoir confirmé le fonctionnement efficace de nos algorithmes sur Cartpole, nous les avons appliqués à nos environnements internes pour pouvoir entraîner nos agents et les tester. L'analyse des performances lors de l'entraînement est effectuée en observant par exemple pour les algorithmes neuronaux les courbes de perte (loss) et de récompenses (reward), afin d'évaluer l'efficacité de notre agent à apprendre et améliorer ses performances. Nous analysons également la performance d'un algorithme en le faisant jouer contre nos autres algorithmes ou encore contre des agents qui utilisent des politiques aléatoires, afin de s'assurer qu'ils puissent au moins battre des stratégies non optimisées et démontrer une compréhension de base des mécaniques du jeu.

2.4 Déploiement sur CodinGame

Enfin, après un entraînement intensif dans nos environnements internes, les algorithmes sont adaptés et déployés sur la plateforme CodinGame. Les adaptations nécessaires pour satisfaire les exigences spécifiques de la plateforme, telles que les contraintes de temps d'exécution et de longueur de code, sont implémentées, impliquant souvent l'optimisation du code pour l'efficacité assurant ainsi que les algorithmes fonctionnent même dans les conditions limitées de CodinGame.

CHAPTER 3

PASSAGE SUR CODINGAME

Comme mentionné plus haut, il y a deux contraintes principales sur CodinGame : le temps de calcul et le nombre de caractères.

Pour les algorithmes neuronaux, le problème du temps de calcul est résolu en entraînant nos réseaux Pytorch en local avant de récupérer les poids pour les reproduire sur CodinGame. Pour le Q-learning, la Q-table se calcule en local puis est utilisée directement sur CodinGame. Pour MCTS, l'entraînement ne peut pas se faire à l'avance. Il n'y a donc pas d'autre choix que d'optimiser au maximum notre code et de faire tourner un nombre réduit de simulations à chaque coup sur la plateforme.

Le second problème est le nombre de caractères limité. En effet, lorsque l'on veut copier-coller les poids enregistrés sur l'IDE de la plateforme, la limite nous empêche de faire tourner notre bot. Ainsi nous devons compresser nos poids. Pour cela, nous suivons les étapes ci-dessous.

1. Convertir en octets avec `pickle.dumps`
2. Compresser avec `zlib.compress`
3. Convertir en ASCII avec `base64.encode`
4. Coller cette chaîne ASCII dans le code sur CodinGame
5. Décoder la chaîne de caractères dans l'IDE de CodinGame

Cependant, même en appliquant cette méthode, les tailles des chaînes de caractères sont beaucoup trop élevées. Par exemple, pour un réseau à deux couches cachées linéaires de taille 512, la chaîne de caractères compressée fait environ 1.7 millions de caractères. Nous sommes donc obligés de simplifier grandement l'architecture des réseaux et de réduire la taille des couches.

En plus de ces deux contraintes données explicitement dans les règles des combats de bots de CodinGame, il existe en réalité un dernier problème : la plateforme ne reconnaît pas Pytorch. Il nous faut donc coder à la main un réseau neuronal qui ne soit pas trop compliqué pour ne pas prendre trop de caractères et qui puisse récupérer les poids de nos réseaux entraînés en local. Pour ce faire, nous avons utilisé un réseau linéaire simple disponible sur GitHub [8].

CHAPTER 4

RÉSULTATS

4.1 Algorithmes non neuronaux

4.1.1 Monte Carlo tree search (MCTS)

Le premier algorithme que nous avons implémenté et testé sur Ultimate Tic-Tac-Toe est le Monte Carlo Tree Search. En effet, c'est l'algorithme le plus simple à mettre sur CodinGame car il n'y a pas de modification propre à la plateforme à faire au code.

MCTS n'a pas besoin de reward complexe. On donne donc simplement à la fin de la partie +1 si le bot gagné, -1 s'il a perdu et 0 s'il y a match nul.

En local, le bot donne de bons résultats, meilleurs que notre heuristique. On peut voir sur la figure 4.1 que notre bot ne perd jamais contre l'heuristique et s'améliore bien lorsque l'on augmente le nombre de simulations faites à chaque coup.

Pour avoir de bons résultats sur CodinGame, toute la difficulté du problème réside dans le fait que les simulations doivent être faites en direct et à chaque coup. Or celles-ci prennent du temps, ce qui n'est pas un problème lorsque l'on fait tourner l'algorithme en local, mais sur CodinGame réduit grandement le nombre de simulations possibles. Nous avons donc dû optimiser notre code au maximum pour qu'il soit le plus rapide possible.

Après optimisation du code, nous arrivons à faire une centaine de simulations en 100ms sur CodinGame. Pour le facteur d'exploration, la figure 4.2 montre qu'à 100 simulations il n'a pas

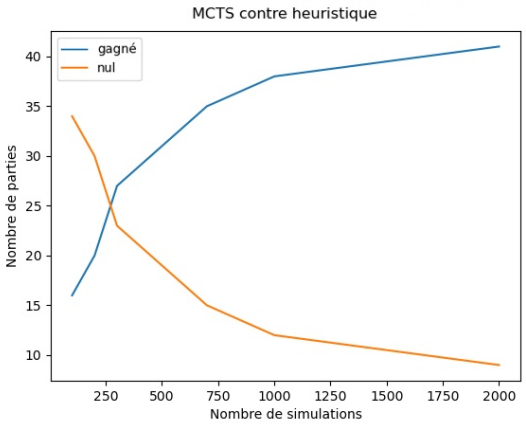


Figure 4.1: Sur un total de 50 parties d'Ultimate Tic-Tac-Toe jouées, le graphe ci-dessus montre le nombre de parties gagnées par le bot MCTS contre l'heuristique et le nombre de matchs nuls en fonction du nombre de parties simulées par le bot à chaque coup.

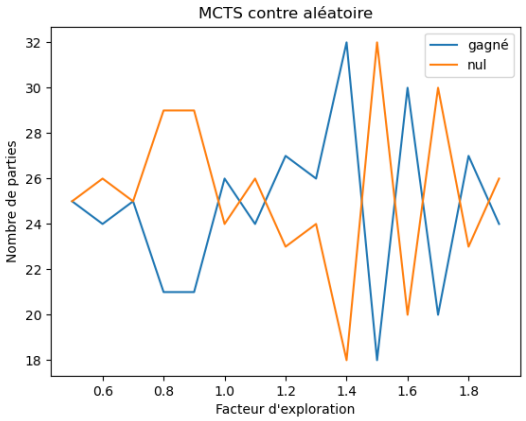


Figure 4.2: Sur un total de 50 parties d'Ultimate Tic-Tac-Toe jouées, le graphe ci-dessus montre le nombre de parties gagnées par le bot MCTS à 100 simulations contre un bot aléatoire et le nombre de matchs nuls en fonction du facteur d'exploration.

grande importance. Nous avons donc choisi 1.4 arbitrairement.

Le faible nombre de simulations ne permet pas à l'agent de bien apprendre et comprendre l'environnement. Ainsi notre bot MCTS est classé 854/897 dans la ligue d'argent, soit 2288/7239 en comptant les ligues d'or, d'argent et de bronze.

Cependant, le bot MCTS reste tout de même meilleur que le bot basé sur notre heuristique qui, lui, a été classé 860/897 dans la ligue d'argent, soit 2294/7239 toutes ligues confondues.

Nous avons choisi de ne pas appliquer MCTS à Mad Pod Racing car dans Mad Pod Racing, les états sont représentés par des variables continues telles que la position, la vitesse, et l'orientation du pod. Les actions, comme les changements de direction et d'accélération, sont également continues. MCTS, conçu pour les espaces discrets, est inefficace pour explorer un espace continu infini de manière exhaustive.

Le temps de calcul pour MCTS augmente rapidement avec la complexité et la taille de l'espace d'états et d'actions. Pour un jeu de course en temps réel comme Mad Pod Racing, où des décisions doivent être prises en une fraction de seconde, MCTS serait trop lent pour être applicable même en discrétisant les actions.

4.1.2 Q-learning

Nous avons choisi de ne pas appliquer le Q-learning sur Ultimate Tic-Tac-Toe en raison de la taille de la Q-table. En effet, si on prend comme état toute la grande grille de 81 cases - ce qui est nécessaire pour avoir une compréhension complète de la situation - où chaque case peut être vide, occupée par un rond ou occupée par une croix, le nombre d'états serait de l'ordre de 3^{81} (moins en réalité car il doit y avoir autant de croix que de ronds). Et il y aurait 81 actions.

L'application de cet algorithme au jeu Mad Pod Racing semble pour autant pertinente, en effet bien que le jeu présente une grande quantité d'états, on peut discrétiser ces derniers pour se ramener à un nombre plus raisonnable.
Une fois la discrétisation faite, on peut appliquer l'algorithme sur notre environnement.

En local, on obtient des résultats similaires à l'heuristique classique (figure 4.3). On note une certaine instabilité des résultats mais cela est attendu d'un tel algorithme. Nous pouvons alors tester les résultats sur codingame. On obtient alors une place de 42702/49358 en ligue bronze, 97 065/202 618 en prenant toutes les ligues en compte. On remarque que ces classements ne sont pas élevés, dû en grande partie à la difficulté de battre des heuristiques travaillées spécifiquement pour ce jeu, en utilisant un algorithme de Q-learning prenant en compte un nombre faible d'états dû à la limite de caractères de CodinGame exigeant une très petite Q-table.

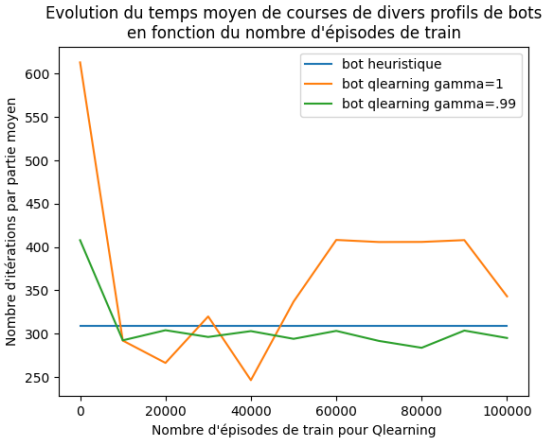


Figure 4.3: Evolution du nombre d'itérations pris pour finir une partie par les algorithmes en fonction du nombre d'épisodes d'entraînement

4.2 Algorithmes neuronaux

4.2.1 Deep Q Learning (DQN)

Ultimate Tic-Tac-Toe

Nous avons implémenté un réseau DQN sur les deux jeux.

Pour Ultimate Tic-Tac-Toe, le problème auquel nous avons fait face est que le réseau n'a pas été capable de comprendre les règles, c'est-à-dire quels coups sont légaux et illégaux. Donner des récompenses négatives lorsque le bot jouait un coup illégal n'a pas été suffisant pour que le réseau apprenne. Il a donc fallu, lors de la prédiction de la meilleur action, après le calcul des q-values, augmenter "à la main" les valeurs des coups légaux pour être sûr que lors de la sélection de l'action par un argmax seul un coup légal soit susceptible d'être tiré.

Cependant, même en faisant cela, le réseau n'arrive pas à apprendre. On voit sur la figure 4.6a que notre agent DQN ne fait quasiment que des parties nulles contre un agent aléatoire en local, et de même contre l'heuristique (figure 4.6b). Notons que le réseau que l'on utilise n'est pas très complexe : deux couches cachées de taille 512 chacune pour une entrée et une sortie de taille 81, avec uniquement des couches linéaires, et une MSE pour la fonction de loss.

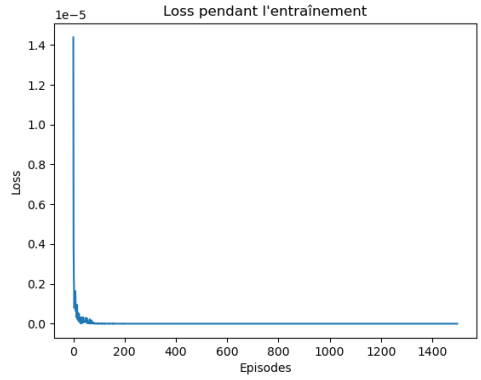
Lorsque l'on observe la loss du réseau, on remarque qu'elle décroît extrêmement rapidement (figure 4.4a), ce qui n'est pas censé être le cas. Pour comprendre le problème, nous avons testé un autre système de récompenses, où l'agent obtient en plus des rewards finales, +0.5 lorsqu'il gagne une petite grille et -0.5 lorsqu'il en perd une. Cependant, cette approche ne marche pas non plus. La figure 4.4b montre que le réseau n'arrive plus du tout à faire décroître sa loss.

Néanmoins, notre réseau marche sur une simplification du problème, où la partie s'arrête lorsqu'une petite grille est gagnée par l'un des joueurs. On peut voir sur le graphe 4.4c que la loss décroît normalement, et sur le graphe 4.5 que les résultats sont bien meilleurs contre un bot aléatoire.

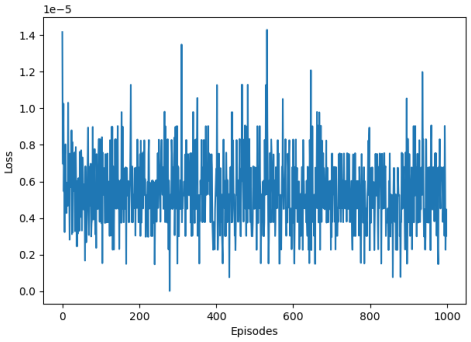
Lorsque l'on veut passer sur CodinGame, la taille du réseau doit être fortement réduite. On est obligé de passer à un réseau à une seule couche cachée de taille 100. Avec un tel réseau, notre bot DQN se classe 871/897 dans la ligue d'argent, soit 2305/7239 en comptant les ligues d'or, d'argent et de bronze.

Mad Pod Racing

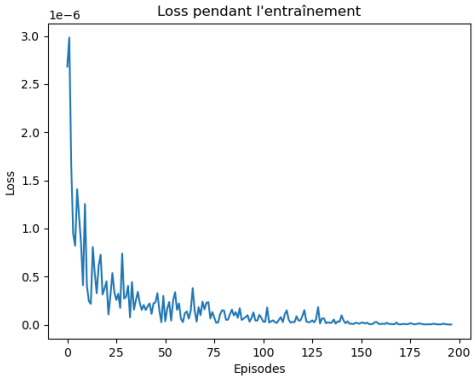
Concernant Mad-Pod-Racing, la typologie du jeu est plus propice à l'utilisation d'un réseau DQN, car c'est un algorithme particulièrement bon dans les environnements où les états sont continus. Comme nous l'avons défini plus haut, les états du jeu seront représentés par la position du joueur, la position du prochain check-point, la distance entre le pod et le prochain check-



(a) avec des récompenses données uniquement en fonction du résultat de la partie



(b) avec des récompenses données à chaque fois qu'une petite grille est gagnée ou perdu



(c) en arrêtant la partie à la première petite grille gagnée

Figure 4.4: Evolution de la loss d'un réseau DQN pendant l'entraînement jouant à Ultimate Tic-Tac-Toe

point et l'angle qui seront donc passés en continu au réseau. Concernant les actions, nous les avons discrétisées en 9 actions possibles pour chaque état, et le reward sera simplement moins la distance pod/checkpoint ainsi que +100 si on l'atteint. Nous utilisons le même réseau que pour Ultimate Tic-Tac-Toe, deux couches cachées de taille 512 mais une sortie de taille 9, chacune avec uniquement des couches linéaires et une MSE pour la fonction de loss.

En analysant les graphiques de la figure 4.7, on remarque que la fonction de perte a une tendance décroissante, indiquant une amélioration progressive de l'apprentissage du modèle, tandis que le graphique des récompenses, représentant la négation de la distance entre le pod et le checkpoint, montre une augmentation, signifiant que le pod se rapproche de manière plus efficace des checkpoints au fil du temps.

Les résultats de la figure 4.8 nous montrent que notre agent DQN s'en sort bien en local contre notre heuristique avec un total de parties gagnées de 926 contre 368, et un pourcentage de victoires de 92.60% contre 36.80%. Cependant lorsque nous avons déployé notre agent sur Codingame, nous avons également dû baisser la taille des couches du réseau à 100. Avec cette taille de réseau, le pod n'arrive pas à passer un tour et fini par faire du hors piste.

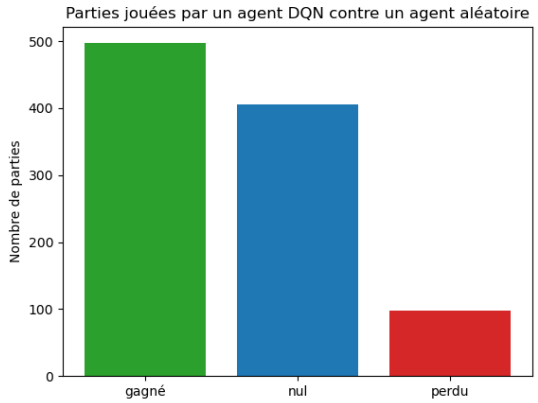
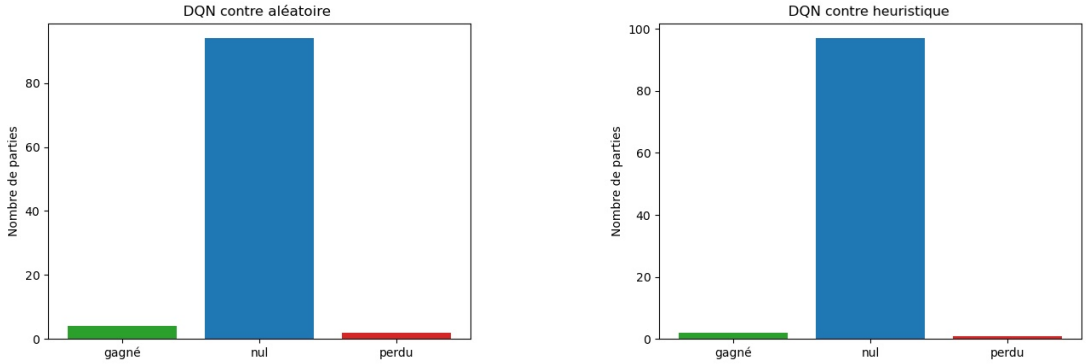


Figure 4.5: Résultats de 1000 parties d'Ultimate Tic-Tac-Toe où la partie s'arrête à la première petite grille complétée jouées par un agent DQN entraîné sur 200 épisodes contre un agent aléatoire



(a) contre un agent aléatoire

(b) contre un agent utilisant notre heuristique

Figure 4.6: Résultats de 100 parties d'Ultimate Tic-Tac-Toe jouées par un agent DQN entraîné sur 1000 épisodes

4.2.2 REINFORCE

Ultimate Tic-Tac-Toe

Les résultats avec l'algorithme REINFORCE sont similaires à ceux de DQN.

Pour Ultimate Tic-Tac-Toe, les problèmes rencontrés sont aussi les mêmes. Encore une fois, le réseau a du mal à apprendre. Comme on peut le voir sur la figure 4.11a, une fois de plus notre agent n'arrive pas à battre l'aléatoire en local.

On voit pourtant que la loss du réseau décroît lors de l'apprentissage (figure 4.9), mais la figure 4.10 montre que le bot n'en devient pas meilleur pour autant.

Le réseau utilisé est le même qu'avant : deux couches cachées de taille 512 chacune pour une entrée et une sortie de taille 81, avec uniquement des couches linéaires.

Pour passer sur CodinGame, la taille du réseau doit donc être réduite à une couche cachée de taille 100. Notre bot utilisant l'algorithme REINFORCE se classe 823/897 dans la ligue d'argent, soit 2257/7239 en comptant les ligues d'or, d'argent et de bronze.

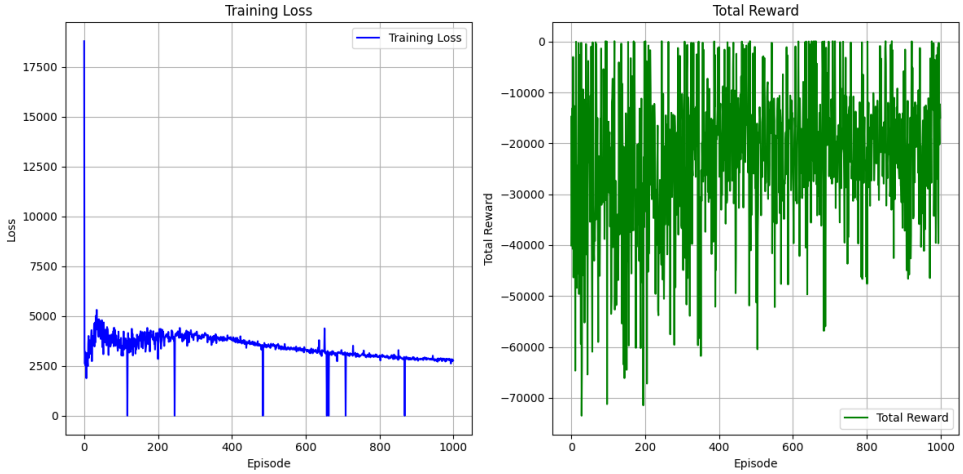


Figure 4.7: Graphiques de loss et de reward pour agent DQN entraîné sur 1000 épisodes de Mad Pod Racing

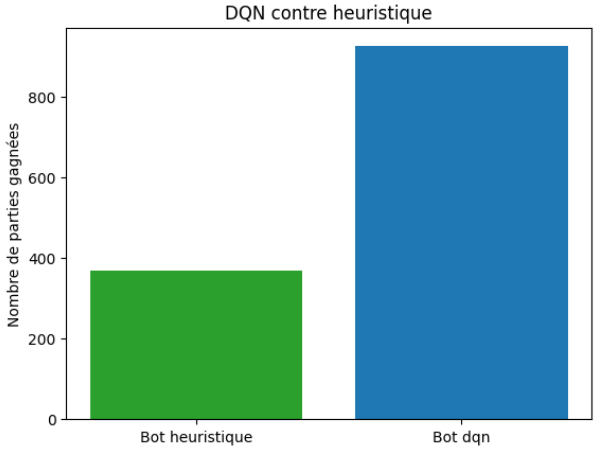


Figure 4.8: Résultats de 1000 parties de Mad-Pod-Racing jouées par un agent DQN contre notre heuristique

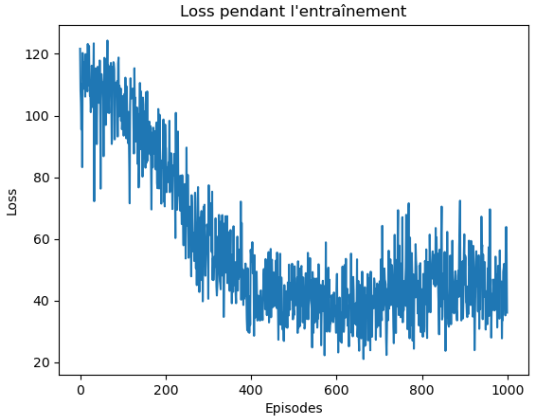


Figure 4.9: Evolution de la loss d'un réseau REINFORCE jouant à Ultimate Tic-Tac-Toe sur 1000 épisodes

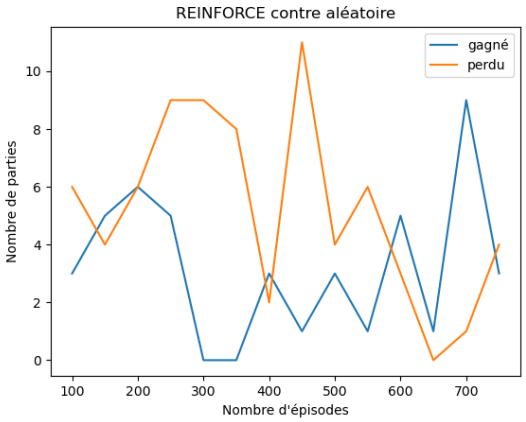
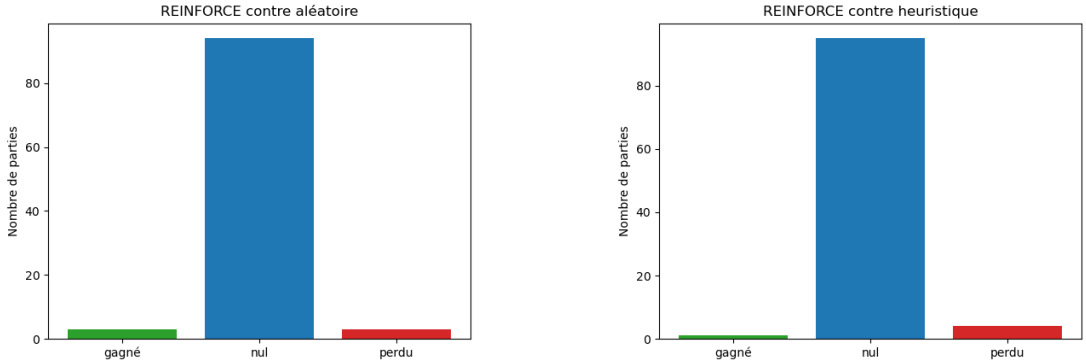


Figure 4.10: Sur un total de 100 parties d'Ultimate Tic-Tac-Toe jouées, le graphe ci-dessus montre le nombre de parties gagnées par le bot REINFORCE contre un bot aléatoire et le nombre de matchs nuls en fonction du nombre d'épisodes lors de l'entraînement.



(a) contre un agent aléatoire

(b) contre un agent utilisant notre heuristique

Figure 4.11: Résultats de 100 parties d'Ultimate Tic-Tac-Toe jouées par un agent REINFORCE entraîné sur 1000 épisodes

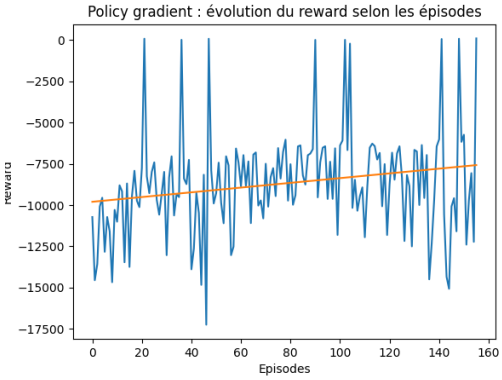
Mad Pod Racing

Bien que l'algorithme REINFORCE fonctionne correctement sur CartPole, il ne semble pas apprendre sur Mad Pod Racing. Bien qu'il semble apprendre (figure 4.12a), l'augmentation ne semble pas significative et est probablement une coïncidence, au vu de la loss incohérente (figure 4.12b).

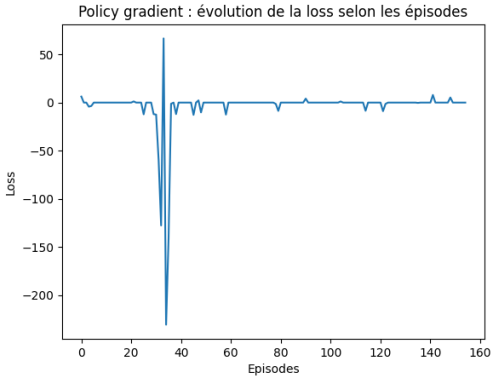
N'ayant pas réussi à faire finir une partie en local à cet agent, nous ne l'avons pas implémenté sur CodinGame.

4.3 Résultats combinés

La figure 4.13 synthétise les résultats des agents que nous avons pu implémenter sur la plateforme CodinGame.



(a) évolution du reward



(b) évolution de la loss

Figure 4.12: Evolution du reward et de la loss lors de l'entraînement de l'agent REINFORCE sur Mad Pod Racing

	Mad Pod Racing	Ultimate Tic-Tac-Toe
Heuristique	72642	2294
MCTS	Non implémenté	2288
Q-learning	97065	Non implémenté
DQN	102862	2305
REINFORCE	Non implémenté	2257
Nombre total de joueurs	202 619	7239

Figure 4.13: Placement (toutes ligues confondues) de l'implémentation de chaque algorithme sur CodinGame

CHAPTER 5

CONCLUSION

En conclusion, nous avons pu comparer l'application d'algorithmes de reinforcement learning génériques sur des jeux présentant de nombreuses heuristiques spécifiques proposées. En vue de nos résultats, nous pouvons constater que les limites imposées par CodinGame, en terme de taille de code (et donc de mémoire) comme de temps contraignent fortement l'expressivité de nos algorithmes, qui n'obtiennent donc pas nécessairement des hauts placements.

A travers ce projet, nous avons pu explorer et implémenter divers algorithmes de reinforcement learning, variés dans leurs approches, allant des méthodes non neuronales aux techniques neuronales plus avancées. Cela nous a servi d'introduction au reinforcement learning et ses difficultés, en particulier nous avons dû nous confronter aux défis de devoir simplifier nos modèles pour les adapter aux contraintes d'une plateforme, et donc bien les maîtriser. Les résultats mitigés des algorithmes proposés sont probablement en partie liés à notre manque d'expérience en ce domaine, mais nous ne percevons pas cela comme un échec, mais une opportunité d'apprendre et de faire mieux dans nos prochaines implémentation d'algorithmes de reinforcement learning.

LIST OF FIGURES

BIBLIOGRAPHY

[1] Codingame. *Ghost In The Cell*. https://www.codingame.com/multiplayer/bot-programming/ghost-in-the-cell. Consulté le 15 mai 2024.

[2] Codingame. *Mad Pod Racing*. https://www.codingame.com/multiplayer/bot-programming/mad-pod-racing. Consulté le 15 mai 2024.

[3] Codingame. *Seabed Security - Fall Challenge 2023*. https://www.codingame.com/multiplayer/bot-programming/seabed-security. Consulté le 15 mai 2024.

[4] Codingame. *Ultimate Tic-Tac-Toe*. https://www.codingame.com/multiplayer/bot-programming/tic-tac-toe. Consulté le 15 mai 2024.

[5] Ian Danforth. *Continuous Cartpole for OpenAI Gym*. https://gist.github.com/iandanforth/e3ffb67cf3623153e968f2afdfb01dc8. 2018.

[6] Ivo Grondman, Lucian Busoniu, Gabriel Lopes, and Robert Babuska. "A survey of actor-critic reinforcement learning: standard and natural policy gradients". In: *IEEE Transactions on Systems, Man, and Cybernetics* Part C: Applications and Reviews (2012), pp. 1291–1307. DOI: https://hal.science/hal-00756747.

[7] magusgeek. *Coders Strikes Back*. http://files.magusgeek.com/csb/csb.html. Consulté le 27 février 2024.

[8] Jason Rute. *Neural Network*. https://github.com/jasonrute/Neural-Network. 2017.

[9] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning, second edition: An Introduction (Adaptive Computation and Machine Learning series)*. Bradford Books, 2018. ISBN: 9780262039246.