UNIVERSITÉ DE SHERBROOKE
DÉPARTEMENT D'INFORMATIQUE

Rapport de fin de session

Dans le cadre des cours:
IFT592 - Cours de projets informatiques
IFT692 - Cours de projets informatique II

Travail présenté
à
Pr Djemel Ziou

Par
Léo Chartrand - CHAL2525 (IFT692)
Jeremi Levesque - LEVJ1404 (IFT592)
Olivier Thiffault - THIO2102 (IFT592)

21 avril 2023

https://github.com/leochartrand/TetrisAI.jl

Table des matières

1 Introduction **3**
1.1 Implications socio-économiques de la recherche en RL . . . 3
1.2 Tetris . . . 3
1.3 Première itération de TetrisAI.jl . . . 5

2 Contexte **6**
2.1 Principes de bases du RL . . . 6
2.2 Problèmes sous-jacents et contraintes du RL . . . 7

3 Volet de développement **8**
3.1 Analyse critique . . . 8
3.1.1 Architecture des classes Agent et Model . . . 8
3.1.2 Apprentissage par Démonstration . . . 8
3.1.3 Extraction de caractéristiques . . . 9
3.1.4 Révision des algorithmes d'apprentissages . . . 10
3.1.5 Affichage de benchmarks . . . 10
3.1.6 Documentation . . . 10
3.1.7 Bogue de fonts dans GameZero . . . 11
3.2 Conception . . . 12
3.2.1 Format des données de jeux . . . 12
3.2.2 Construction du dataset . . . 12
3.2.3 Module d'extraction de caractéristiques . . . 12
3.2.4 Représentation d'un Agent . . . 14
3.3 Implémentation . . . 15
3.3.1 Pipeline de partage de données . . . 15
3.3.2 Interface web pour accès grand public . . . 15
3.3.3 Campagne de promotion . . . 17
3.3.4 Module d'extraction de caractéristiques . . . 18
3.3.5 Classe Agent . . . 19
3.3.6 Affichage de benchmarks . . . 20
3.3.7 Tampons de mémoire . . . 21
3.4 Déploiement du Package Julia . . . 22
3.4.1 Documentation . . . 22
3.4.2 Publication . . . 23

4 Volet de Recherche **24**
4.1 Revue de la littérature . . . 24
4.1.1 DQN . . . 24
4.1.2 PPO . . . 26
4.2 Méthodologie . . . 29
4.3 Résultats . . . 29

5 Conclusion **31**

1 Introduction

1.1 Implications socio-économiques de la recherche en RL

L'apprentissage par renforcement (RL) est un sous domaine de l'intelligence artificielle (IA) qui suscite indirectement l'intérêt des médias depuis quelques années. Plusieurs percées récentes ont permis de susciter un intérêt marqué pour l'apprentissage par renforcemenent, en particulier notamment grâce aux performances impressionnantes des modèles développés par DeepMind dont AlphaGo [1] ou encore les itérations subséquentes nommées AlphaZero [2] et MuZero [3] qui offrent des performances surprenantes grâce à des techniques de RL innovantes. C'est un domaine qui vient créer un grand impact sur de plus en plus d'industries variées dont le domaine automobile, le domaine médical, les jeux vidéos et jusqu'à même, plus récemment, le traitement du langage naturel (notamment ChatGPT[1]). L'apprentissage par renforcement n'en est qu'à ces débuts et son utilisation dans plus de domaines est contrainte par plusieurs facteurs qui sont des sujets de recherche très actifs à ce jour. Par contre, beaucoup de potentiel est envisagé si ces contraintes peuvent être surmontées par la recherche: cela ouvrirait la porte à beaucoup plus d'applications, notamment dans la robotique où, à ce jour, des méthodes heuristiques sont généralement employées, ou encore d'autres domaines comme le développement d'une IA généralisée.

1.2 Tetris

Tetris est un classique du jeu vidéo ayant eu un impact important au niveau socioculturel dès sa sortie où 1.5 millions de copies du jeu ont été vendus dès sa sortie sur la NES en 1989. La création de ce phénomène est attribuée au développeur informatique Soviétique *Alexey Pajitnov* en 1985. Tetris est un jeu où des pièces, nommées Tetrominoes, tombent une par une du haut d'une grille de 10x20 cases. Plusieurs types de Tetrominos (voir la Figure 1) tombent une à une en alternance aléatoire.

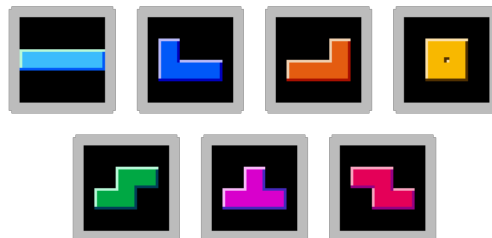


Figure 1: Différents Tetrominoes possibles

Le jeu de Tetris fonctionne comme suit: alors qu'une pièce tombe vers le bas la grille, le joueur peut manipuler la pièce en la faisant bouger horizontalement, tourner, ou tomber immédiatement à son emplacement le plus bas. Lorsque la pièce entre en contact avec le bas de la grille ou une pièce déjà placée, elle est alors figée et une nouvelle pièce commence à tomber du haut de la grille. Les pièces peuvent donc s'empiler une à une jusqu'à ce que la pile atteigne le haut de la grille, dans quel cas la partie est finie. Le joueur doit manipuler les pièces en fonction de leur forme afin de les placer pour remplir des lignes de la grille. Lorsqu'une ligne est remplie de pièces, celle-ci est vidée et toutes les lignes au-dessus de celle-ci tombent vers les bas. Chaque ligne dégagée rapporte des points. Il est possible de remplir plusieurs lignes en un seul coup, dans quel cas les points sont multipliés.

[1] https://openai.com/blog/chatgpt

Ainsi, l'élimination de lignes permet d'accumuler des points et de survivre plus longtemps. Le jeu est donc théoriquement infini en pré-supposant que le joueur puisse éviter d'empiler les pièces jusqu'en haut de la grille indéfiniement. Par contre, le jeu s'accélère à pratiquement chaque niveau et la complétion de 10 lignes permet de monter d'un niveau. Le but pour un joueur est donc de maximiser le score obtenu durant toute la durée de la partie en complétant des lignes.

Il y a une corrélation directe entre le temps d'une partie et le score obtenu. Cependant, certaines stratégies sont plus avantageuses, puisqu'elles pourraient prendre en compte la multiplication des points lors de la complétion simultanée de plusieurs lignes. Par exemple, lorsque 4 lignes sont remplies avec une pièce I placée verticalement, le joueur produit ce qu'on appelle un *Tetris* (d'où le nom du jeu), qui rapporte le maximum de points pouvant être obtenu à la fois. Ce genre de stratégie est considéré comme optimal.

Figure 2: Interface du jeu Tetris sur la NES. On peut voir qu'une pièce en I sera placée au bas de la grille à droite, ce qui déclenchera un Tetris.

Tetris[2] est un environnement non trivial et démontré NP-Complet, [4] ce qui fait de ce jeu un environnement intéressant pour la recherche en apprentissage par renforcement. Le jeu comporte un grand nombre de configurations différentes, soit $7 * 2^{200}$ combinaisons possibles [5], ce qui rend l'apprentissage particulièrement complexe et l'environnement difficile à approximer. De plus, la complétion d'une ligne requiert de placer plusieurs pièces d'une manière particulière, demandant à chaque fois l'exécution d'une longue suite d'actions précises spécifiques à un état unique. Informellement, le jeu peut à première vue sembler aussi complexe que tout autre jeu vidéo de style Atari comme utilisé couramment dans la recherche en RL, mais de toute évidence Tetris se rapproche beaucoup plus des échecs ou du jeu de Go.

[2]Pour plus de détails sur le jeu et l'environnement, voir: https://fr.wikipedia.org/wiki/Tetris

1.3 Première itération de TetrisAI.jl

Dans le contexte de ce cours projet, nous visons à créer un *package*, TetrisAI.jl, écrit entièrement en Julia qui permettrait à tout utilisateur d'entraîner un agent intelligent à Tetris facilement. Afin d'arriver à cet objectif, une première itération du projet a déjà été réalisée à l'été 2022 par Léo Chartrand et Guillaume Cléroux dans le cadre d'un premier cours projet. Cette première étape fournit une bonne fondation à la réalisation étant donné qu'elle fournit l'environnement de jeu Tetris complètement implémenté et relativement bien interfacé qui permet l'entraînement d'un agent. Cependant, l'entraînement initial d'un agent résulte en un agent qui semble faire des actions quasi aléatoires et qui ne parvient pratiquement jamais à compléter une seule ligne. L'objectif de cette deuxième étape serait de compléter ce *package* afin de permettre un entraînement d'agent avec des performances raisonnables et idéalement des performances supérieures à un humain moyen en utilisant des méthodes de RL.

2 Contexte

2.1 Principes de bases du RL

En RL, un agent observe l'état de son environnement et prend une action selon une politique de façon à optimiser le gain de récompenses. Les récompenses permettent à indiquer à l'agent quels comportements sont désirables et lesquels sont indésirables (ultimement pour l'atteinte de certains objectifs). Un agent a son propre modèle de l'environnement qu'il utilise pour prédire les gains futurs selon l'état courant. Le choix de l'action qu'il prend est basé sur une politique (qu'on appelle aussi *stratégie* ou *plan*). En principe, un agent explore et exploite son environnement, opérant ainsi par essai erreur pour augmenter la précision de son modèle et la performance de sa politique.

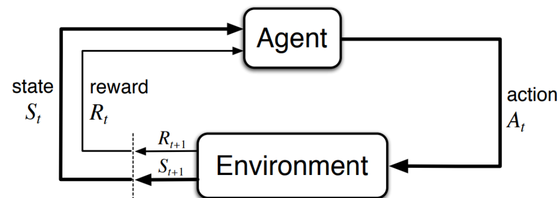


Figure 3: L'interaction de l'agent avec son environnement[6].

En pratique, par contre, la plupart des situations comportent des environnements trop complexes pour pouvoir générer un modèle offrant une distribution de probabilités de transitions d'états de l'environnement. C'est pourquoi la plupart des algorithmes de RL aujourd'hui sont appelés *model-free*, approximant un modèle à partir d'échantillons obtenus à partir d'expériences vécues dans l'environnement par l'agent.

Plusieurs familles d'algorithmes existent, offrant des méthodes particulières pour prédire la meilleure action à prendre pour atteindre les objectifs. On dit qu'un algorithme est *On-Policy* s'il utilise la même politique que celle qu'il améliore, alors qu'un algorithme *Off-Policy* évalue et corrige des politiques différentes [6]. Plusieurs algorithmes, comme le Q-learning (off-policy) et SARSA (le cousin on-policy du Q-learning), utilisent une *value-function* pour évaluer des paires action-état afin de calculer une politique. Étant donné la complexité des environnements et le temps requis pour optimiser les value function et policy function, il est commun d'utiliser des réseaux de neurones pour approximer ces fonctions. On parle alors de Deep-Q-Network (DQN) et de Deep-SARSA-Network (DSN). En contraste, une famille d'algorithme qu'on appelle Policy Gradient cherche plutôt à paramétrer et calculer la politique directement. Un algorithme Actor-Critic est d'ailleurs un policy gradient qui fait usage d'une *value function* (*critic*) afin de calibrer la mise à jour de la politique (*actor*). Aujourd'hui, une des méthodes les plus populaires est la famille des PPO [7], apportant beaucoup d'améliorations aux algorithmes Policy Gradient classiques.

Une alternative aux algorithmes RL, qui sont généralement fondamentalement basés sur le modèle du processus de décision markovien, est la famille des algorithmes génétiques qui est plutôt basée sur des heuristiques, où un processus de sélection naturelle fait évoluer une population d'agents déterministes. Cependant, les algorithmes génétiques se rapprochent des algorithmes RL dans le sens que tous deux opèrent sur le principe d'essai erreur. D'ailleurs, les algorithmes génétiques peuvent souvent offrir une solution de performance comparable aux méthodes RL sur certains problèmes similaires.

Une autre famille d'algorithmes souvent utilisée dans la recherche en RL sont les algorithmes de planification, qui sont par définition *model-based*. Parmi cette famille, on compte en particulier les algorithmes de recherche heuristique, par exemple le Monte Carlo Tree Search, utilisé notamment par AlphaZero [2] et MuZero [3]. Ces algorithmes recherchent l'espace d'états afin d'approximer un modèle qui facilite ensuite l'évaluation et le calcul d'une politique.

2.2 Problèmes sous-jacents et contraintes du RL

Étant donné que l'entraînement d'un agent en RL n'est pas directement supervisé, donc n'utilise pas de données étiquetées, l'agent doit interagir avec son environnement pour recevoir du feedback des actions qu'il entreprend. Afin d'accélérer le processus d'apprentissage et afin d'éviter toute sorte d'accidents dans l'environnement réel, on entraîne généralement un agent à l'intérieur d'une simulation. Un défi particulier avec l'entraînement en simulation est que généralement on cherche à modéliser la simulation le plus près de l'environnement réel dans lequel l'agent final fonctionnerait. Bien évidemment, il s'avère particulièrement complexe de modéliser parfaitement un environnement de la vie réelle (à moins que l'environnement de travail réel soit un environnement digital comme Tetris dans notre cas), ce qui peut créer une divergence de performance entre l'environnement d'entraînement et celui d'application.

Un autre défi spécifique à l'apprentissage par renforcement est la modélisation de la fonction de récompense. Présentement, il n'existe pas de fonction de récompense générale; on doit modéliser une fonction de récompense qui permet de guider l'agent à chaque action vers un certain objectif que l'on veut le voir accomplir dans un environnement particulier. Plusieurs environnements, dont l'environnement de Tetris, sont modélisés par des récompenses clairsemées - il faut effectuer une longue séquence d'actions avant même de pouvoir obtenir une récompense. Déjà qu'il est difficile pour un agent d'apprendre dans l'environnement Tetris à cause de son grand nombre d'états possibles, il n'augmente seulement son score que lorsqu'il complète au moins une ligne. Pour compléter une ligne, il doit tourner, translater et placer plusieurs pièces à des bonnes positions tout en ne recevant aucune récompense entre chaque pièce. Le problème des récompenses clairsemées (*sparse reward*) s'applique aussi à bien d'autres domaines, dont celui de la robotique[8]. Ceci illustre donc l'importance de bien connaître le domaine d'application afin de pouvoir appliquer possiblement des récompenses intermédiaires pour mieux guider l'agent dans son apprentissage.

Évidemment, d'autres facteurs limitent l'utilisation de l'apprentissage par renforcement dans tous les domaines. Comme Tetris est un environnement relativement simple à modéliser, une attention particulière est portée sur les défis de récompenses clairsemées et le grand nombre d'états[3].

[3]Le *Model-Based RL* est un domaine activement en recherche afin de faciliter la modélisation d'un environnement. Voir: https://medium.com/the-official-integrate-ai-blog/understanding-reinforcement-learning-93d4e34e5698

3 Volet de développement

3.1 Analyse critique

Le projet consiste en deux parties: un volet de développement ainsi qu'un volet de recherche. Une grande partie du développement supporte la recherche qui viendra par la suite. C'est pourquoi la première partie de cette reprise du projet porte presque entièrement sur le développement. Plusieurs modifications doivent être apportées au code original et de nouveaux modules doivent être ajoutés.

3.1.1 Architecture des classes Agent et Model

La première itération de TetrisAI.jl n'implémente qu'un seul algorithme RL, soit le DQN. L'architecture logicielle, qui est a priori supposée supporter plusieurs algorithmes d'apprentissages, a visiblement été développée avec seulement le DQN en tête et ne peut donc pas accueillir des algorithmes qui diffèrent considérablement de celui-ci. Par exemple, il n'y a pratiquement aucun support avec les algorithmes de planification tel que le Monte Carlo Tree Search. La classe Agent prend aussi pour acquis qu'un seul réseau de neurones sera utilisé et les fonctions d'entraînement sont codées spécifiquement pour le DQN.

```
Base.@kwdef mutable struct TetrisAgent <: AbstractAgent
    n_games::Int = 0
    record::Int = 0
    ϵ::Int = 0
    memory::AgentMemory = CircularBufferMemory()
    model = TetrisAI.Model.linear_QNet(258, 7)
    opt::Flux.Optimise.AbstractOptimiser = Flux.ADAM(LR)
    criterion::Function = Flux.Losses.mse
end
```

Figure 4: La structure d'un agent de base qu'on retrouve dans le code original.

Une restructuration des classes Agent et Model est de mise. La classe Agent doit pouvoir supporter plusieurs types de modèles ainsi que la combinaison de ceux-ci. Idéalement, la classe Model devrait pouvoir supporter plus que les réseaux de neurones et donc supporter toute information relative aux algorithmes d'apprentissages: les poids d'un arbre, les gènes d'une population, etc. L'implémentation d'algorithmes génétiques et des méthodes de recherche heuristique nécessitent par contre une restructuration en profondeur, ce qui demande beaucoup plus de temps que ce dont nous disposons dans le cadre de ce cours de projet. La révision des classes Agent et Model laissera donc la possibilité d'implémenter de telles méthodes, mais les algorithmes MDP seront privilégiés.

3.1.2 Apprentissage par Démonstration

Un des plus gros obstacles auxquels on fait face lorsqu'on tente d'appliquer le RL à Tetris est l'immense complexité du jeu: l'espace d'états observables distincts atteint une taille de 2^{228}. Le problème des récompenses clairsemées rend les choses plus difficiles, puisque pour obtenir un premier score (en complétant une ligne), une longue séquence d'actions précises doit être effectuée. Il va de soi que ça prendrait un temps énorme pour converger vers une politique qui permet de

dépasser ce premier plateau, spécifiquement si l'apprentissage se fait par essai erreur.

Une solution à ce problème a été proposée et implantée lors de la première itération du projet, soit le *behavioral cloning* [9]. Il s'agit d'une forme d'apprentissage supervisé où l'agent est entraîné sur des paires état-action généré par des humains, permettant à l'agent d'approximer une politique experte. Un module a été développé permettant ainsi aux joueurs d'enregistrer des données étiquetées qui peuvent ensuite être utilisées pour le pré-entraînement des agents. Celui-ci cependant n'est pas complet, offrant seulement la possibilité de générer et utiliser des données locales.

Des modifications doivent y être apportées pour finalement obtenir un pipeline de génération de données étiquetées. Les paires action-état doivent être traitées, augmentées et envoyées dans un répertoire en ligne pour faciliter le partage lors du développement. Ceci permettra d'ailleurs dans le futur de générer un jeu d'entraînement qui pourra être hébergé sur un répertoire en ligne, permettant ainsi son utilisation par les pairs. Cette partie est nécessaire pour assurer une reproductibilité des résultats ainsi que pour faciliter l'entraînement d'algorithmes lors de la phase de recherche.

3.1.3 Extraction de caractéristiques

Afin de pouvoir entraîner des agents RL, la première itération de TetrisAI.jl offre l'état du jeu sans prétraitement. On a donc comme vecteur d'état un vecteur de 228 caractéristiques (une grille 20×10, plus 4 vecteurs *one-hot* de taille 7, soit 3 pour les prochaines pièces et 1 pour la pièce retenue en *hold*). Cependant, la grille de jeu est aplatie et toute information spatiale relevant de la position des cellules au sein de la grille est perdue. Une façon de traiter cette information spatiale serait de donner cette grille à un réseau de neurones à convolutions (CNN), ce qui sera facilité par la restructuration du code.

Une autre manière d'exploiter la structure spatiale de la grille de jeu est d'en extraire des caractéristiques à l'aide d'heuristiques déterminées par des humains, une méthode qui relève de l'extraction de caractéristiques (connu sous le nom de *feature engineering* dans l'industrie). Cette technique permet de 1) réduire la dimensionalité des états qu'on passe aux agents; et 2) éliminer le bruit et faire ressortir les éléments saillants et jugés importants.

Par ailleurs, une technique qui se rapproche de l'extraction de caractéristiques en RL est ce qu'on appelle la modélisation des récompenses (*reward shaping*). Comme expliqué en Section 2.2, le score de Tetris augmente seulement avec une suite de mouvements précis effectués par l'agent, ce qui introduit le problème des récompenses clairsemées (*sparse reward*). La modélisation des récompenses apporte une solution à ce problème, permettant de diriger l'exploration à l'aide de connaissances du domaine et ainsi élaguer l'espace d'état [10]. Des fonctionnalités de modélisation des récompenses doivent être implémentées, faisant usage du module d'extraction de caractéristiques développé conjointement.

L'ensemble de ces techniques sera ajouté au logiciel afin de permettre une diversité de choix dans l'implémentation de différents agents. Il sera donc possible d'utiliser des algorithmes qui font usage de connaissances humaines des règles du jeu pour ainsi faciliter l'apprentissage. Il sera tout de même possible de produire des agents qui traitent des données non prétraitées, devant apprendre les dynamiques de l'environnement, le but étant de pouvoir implémenter une grande diversité d'algorithmes afin de pouvoir les étudier. L'ajout de ces techniques sera nécessaire pour le volet recherche et peut être entièrement complété durant la phase de développement.

3.1.4 Révision des algorithmes d'apprentissages

Le DQN présentement implémenté ne comporte qu'un seul réseau de neurones et malgré l'utilisation d'un tampon pour le *Experience Replay*, il s'agit plutôt d'un algorithme dérivé de SARSA étant donné sa nature on-policy. Suite à la restructuration, il sera possible d'implémenter un vrai DQN à deux réseaux. Lors de la première itération du projet, les tests ont été effectués sur 3 agents: un agent aléatoire, le "DQN" et ce même "DQN" mais pré-entraîné sur quelques milliers de données. La plateforme TetrisAI.jl est conçue ultimement pour tester différents algorithmes RL, et ces 3 agents n'apportent pas une très grande diversité d'algorithmes. Il est impératif de corriger les algorithmes en place et d'en ajouter plusieurs autres.

Tout d'abord, une correction de l'algorithme DQN doit être apportée. Par la suite, les algorithmes sélectionnés suite à une revue de la littérature seront implémentés, avec une possibilité de combiner des modèles. Cela dit, une attention sera portée premièrement sur les algorithmes RL classiques basés sur les MDPs. L'implémentation des autres familles algorithmes (recherche heuristique, algorithmes génétiques) dépendra du temps dont nous disposerons dans le cadre de ce projet. L'intégration des différentes techniques se fera progressivement suite à la restructuration des classes Agent et Model (voir 3.1).

3.1.5 Affichage de benchmarks

Afin d'obtenir un agent un minimum performant, l'entraînement s'étend généralement sur une période pouvant aller de plusieurs heures à plusieurs jours. Dans la première itération du projet, nous pouvons seulement évaluer la performance de l'agent soit par rapport au score maximal atteint globalement qui est affiché et mis à jour à chaque étape d'entraînement. Par contre, cela ne nous donne pas un historique exhaustif de la performance de l'agent et notre seule métrique durant l'entraînement est sa performance maximale jusqu'à présent. Afin d'évaluer la performance des algorithmes implémentés, on voudrait pouvoir introduire des représentations graphiques qui peuvent illustrer les métriques sur l'entraînement d'un agent. Par exemple, le nombre de ticks de chaque partie peut illustrer jusqu'à un certain point la performance de l'agent au début de l'entraînement avant qu'il puisse compléter une ligne et par la suite on voudrait voir une progression graphique du score de l'agent dans la partie.

3.1.6 Documentation

Comme tout bon logiciel en déploiement, un package Julia se doit d'être bien documenté. Il est impératif pour les développeurs d'accompagner les livrables d'une documentation claire et exhaustive afin que tout utilisateur puisse facilement comprendre l'usage des fonctions du logiciel et utiliser ce dernier avec facilité. Par documentation, on entend deux choses: la documentation par commentaire, insérée au sein du code, et la documentation indirecte qui regroupe l'information dans un document à part (fichier README, guide d'utilisateur, etc.).

La première itération de TetrisAI.jl était d'ailleurs supposée répondre à ces critères. Cependant, la majorité de la documentation se retrouve dans le rapport qui a accompagné la livraison de ce projet. Aucun guide ou manuel d'instruction n'a été produit, et une fraction importante du code n'est pas accompagnée de descriptions directes en commentaires.

Il va de soi que pour la continuation de ce projet, la description de toute fonctionnalité produite lors de la première itération doit être générée, comme pour tout nouvel élément ajouté au package.

En ce qui concerne la documentation externe, le package Documenter.jl permet de générer cette documentation automatiquement à partir de *docstrings* (commentaires) accompagnant les éléments du code. Cette documentation peut ensuite être hébergée sur le répertoire du code (GitHub par exemple) en format HTML et Markdown. Il ne suffit donc que de documenter le code directement au fur et à mesure, en respectant certaines règles de syntaxe. Cette partie du projet ne représente pas de défi quelconque et est tout à fait réalisable dans le cadre du cours de projet.

3.1.7 Bogue de fonts dans GameZero

La livraison de la première itération n'était pas sans coquille: un bogue au niveau de l'interface graphique causait un crash régulièrement sur tous les systèmes. Cela dit, ce bug n'était pas causé par le code de TetrisAI.jl en soit, mais plutôt par un package qui était fondamental au logiciel, c'est-à-dire GameZero.jl, qui supportait le rendu du jeu Tetris. Ce problème a été identifié et corrigé directement sur le dépôt en accès libre de GameZero.jl[4]. Comme l'interface de Tetris est rafraîchie plusieurs fois par seconde, on crée un objet graphique, un *TextActor*, pour afficher du texte plusieurs fois par secondes afin de tenir le texte à jour à chaque *tick*. En creusant dans le code de GameZero.jl, nous avons identifié qu'à chaque création de l'objet *TextActor*, qui est fait à chaque *tick*, le fichier de référence pour la police d'écriture est ouvert à chaque fois et n'est jamais fermé. Donc, plus on laissait l'interface de Tetris se rafraîchir longtemps, plus le nombre de fichiers ouverts pour ce processus devenait immense à une vitesse de plusieurs centaines de fichiers ouverts par seconde. Ce problème a passé inaperçu dans la première version du projet puisque tout était majoritairement testé sous Linux et sur Linux la restriction pour le nombre maximal de fichiers ouverts pour un processus est beaucoup moins stricte: on pouvait avoir quelques millions de fichiers ouverts avant que ça ne cause un problème et ça prend plusieurs minutes pour se rendre à ce niveau.

[4] https://github.com/aviks/GameZero.jl/issues/55

3.2 Conception

3.2.1 Format des données de jeux

Afin de pouvoir entraîner notre agent par apprentissage par démonstration, nous devons être en mesure bâtir un dataset assez volumineux pour que l'agent puisse avoir un référant pertinent. Suite à chaque partie de Tetris complétée, nous voulons recueillir sous forme de fichiers les informations pertinentes à l'entraînement de l'agent. Nous considérons que les informations pertinentes dans ce cas sont l'état de la grille du jeu à chaque état de la partie, les actions prises par le joueur ainsi que le score de la partie complétée. De ce fait, l'agent peut évaluer la situation de la grille de jeux pour chaque action prise par le joueur et déterminer par le score si cette stratégie est gagnante.

Nous voulons donc recueillir sous forme de vecteurs les informations d'action et d'état et les transférer sous forme de fichiers JSON. Les scores de parties seront enregistrés dans un fichier texte simple. Pour ce qui est du vecteur d'état, nous voulons recueillir les 200 valeurs pour chaque position de la grille Tetris de 20 lignes par 10 colonnes. Les positions de la grille qui sont présentement occupées par la pièce active seront représentées par 2, les positions occupées par les pièces placées seront représentées par 1 et les positions vides seront représentées par 0. Pour ce qui est du vecteur d'actions, nous voulons recueillir chaque action prise par le joueur. Un mouvement de la pièce active vers la gauche est représenté par 2, un mouvement vers la droite est représenté par 3, un hard-drop est représenté par 4, une rotation de la pièce dans le sens horaire est représentée par 5, une rotation anti-horaire est représentée par 6 et la mise en attente de la pièce est représentée par 7. Il existera une relation 1:1 entre les données d'actions et d'états, c'est-à-dire que l'état de la grille sera enregistré au fichier seulement lorsqu'une action est prise par le joueur.

3.2.2 Construction du dataset

La construction du dataset pour l'entraînement de l'agent passera par la cueillette des fichiers d'action et d'état tel que décrits ci-haut. Pour ce faire, nous voulons premièrement construire un pipeline de partage des données. Ce pipeline nous permettra de téléverser en un endroit centralisé les données de toutes les parties complétées par les joueurs et d'éventuellement télécharger localement sur une machine ces données pour procéder à l'entraînement de notre agent. Nous avons présentement accès à l'interface de jeux directement dans notre projet via l'utilisation du package GameZero. Cela implique donc qu'un joueur doit premièrement avoir Julia d'installé sur sa machine et ensuite d'avoir accès au code source. Pour démocratiser l'accès à l'interface de jeux et ainsi pouvoir recueillir un plus grand volume de données, nous implémenterons une version du Tetris qui sera accessible via un navigateur web et qui sera en communication avec notre pipeline de données. Plus bas se trouvent les détails d'implémentation du pipeline et de l'interface web.

3.2.3 Module d'extraction de caractéristiques

Les fonctionnalités d'extraction de caractéristiques et de modélisation de récompense sont rassemblées et encapsulées dans un module qui peut être utilisé par les agents. Ainsi, lors de l'entraînement ou en inférence lors de démonstrations, un agent peut récolter l'état et la récompense à partir de l'environnement (le jeu), pour ensuite les transformer et ainsi obtenir des données pertinentes qui pourront être traitées. Il s'agit d'un module dont l'utilisation est optionnelle, étant une méthode impliquant une certaine connaissance du domaine et qui peut donc introduire un certain biais.

Le module d'extraction de caractéristiques fonctionne en deux étapes: la première consiste à identifier le type de chaque cellule, i.e. une cellule vide, une cellule pleine un trou, une encoche (*notch*) ou une crevasse.

- Un trou est une cellule vide dont aucun chemin ne peut s'y rendre à partir d'en haut de la grille.
- Une encoche est une cellule vide qui n'est pas un trou, mais dont il existe une cellule pleine au-dessus dans la même colonne.
- Une crevasse est une cellule vide qui n'est pas un trou ni une encoche, qui ne peut être remplie seulement que par une pièce en I.

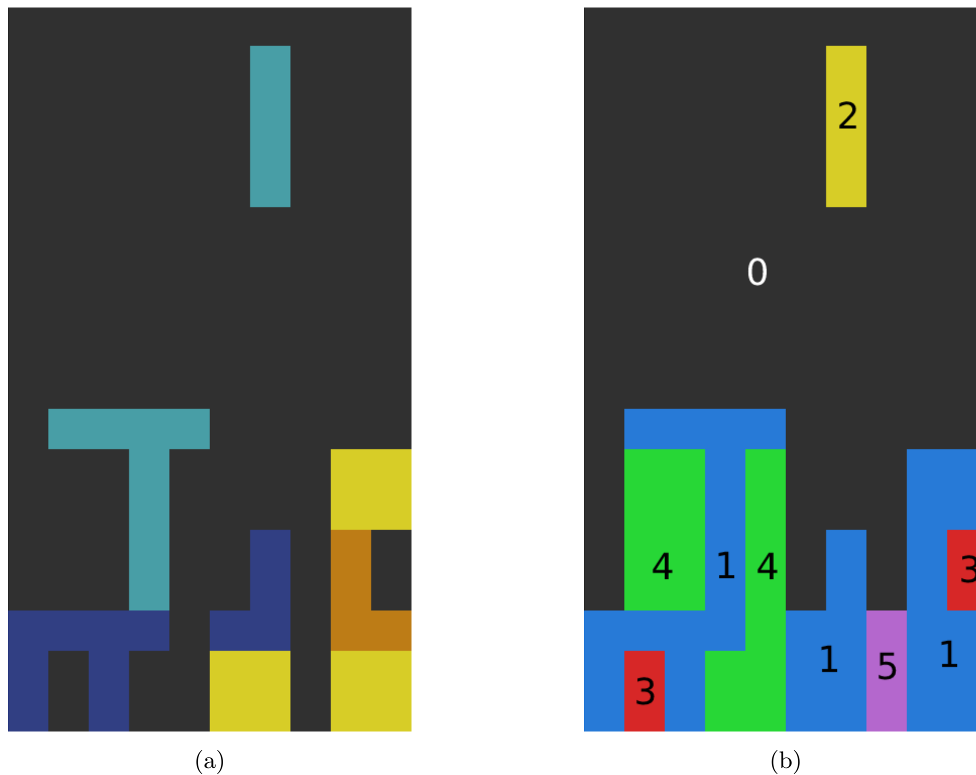


Figure 5: Comparatif d'une grille Tetris avant et après l'identification des types de cellules. En a), la représentation de la grille est telle qu'elle l'est dans le moteur de jeu. En b), les cellules sont identifiées selon leur type: (0) une cellule vide, (1) une cellule pleine, (2) une cellule occupée par la pièce active, (3) un trou, (4) une encoche et (5) une crevasse.

Il existe d'autres motifs qui peuvent être identifiés, parfois s'étalant sur plusieurs cellules, tels que des "balcons" et des "vérandas" [4], par contre l'identification de tels motifs apporte une complexité supplémentaire que nous éviterons pour ce projet. Nous estimons que les 5 types de cellule énoncés dans la Figure 5 suffisent pour faire extraire les informations saillantes de la grille de jeu.

Une copie de la grille peut ainsi être créée avec chaque cellule identifiée par son type. Nous appelons cette copie la grille de caractéristiques (*feature grid*). La deuxième étape du processus d'extraction de caractéristiques consiste à calculer des statistiques à partir de cette grille pour faire ressortir les données importantes. Parmi ces statistiques, on compte la hauteur des colonnes, la hauteur maximale, la moyenne des hauteurs, la *bumpiness*, la hauteur de chute de la pièce active, ainsi que le nombre de trous.

- La hauteur d'une colonne de la grille est définie comme étant le nombre de cases de la première case occupée en partant du haut jusqu'en bas de la grille.
- On dénote la *bumpiness* comme étant la somme des différences de hauteurs absolues entre deux colonnes adjacentes dans la grille.

Beaucoup d'autres caractéristiques peuvent être ajoutées, comme la hauteur des trous, le nombre d'encoches et de crevasses, ainsi que d'autres informations sur la positions de ces cellules. L'extraction d'informations de ce genre appliquée à Tetris a fait l'objet de plusieurs tentatives de réduction de la complexité de Tetris pour faciliter l'apprentissage[11, 12]. Au final, ces données peuvent contribuer à la création d'un vecteur de caractéristiques qui peut être pondéré et ensuite traité par les agents comme une observation de l'état, ou pour aider à modéliser la récompense.

Un autre usage du module est d'utiliser la grille de caractéristiques directement. Plus particulièrement, un usage intéressant est de passer la grille de caractéristiques dans un réseau neuronal à convolutions (CNN). Ainsi, on peut extraire de l'information spatiale à partir de la grille sans calculer des statistiques basées sur des heuristiques. Chaque type de cellule peut être identifié dans une matrice binaire; on a donc une matrice pour chaque type qui devient une carte d'activation pouvant être traité par le modèle. Ainsi, un agent peut utiliser un encodeur parmi le vecteur de caractéristiques ou le CNN afin d'identifier les traits saillants de l'état de l'environnement.

3.2.4 Représentation d'un Agent

La "classe" Agent est revue en entier. Comme à la première itération du projet, une interface abstraite permet un certain polymorphisme. Les différents agents héritant de ce type abstrait sont représentés par différents types composites concrets. Chaque agent a une méthode d'apprentissage particulière qui dépend des algorithmes implémentés. Ainsi, certains agents, comme le DQN, possèdent deux réseaux de neurones plutôt qu'un. De plus, certains algorithmes peuvent nécessiter des hyperparamètres particuliers. Outre tout ceci, les agents disposent aussi d'indicateurs qui spécifient l'utilisation d'extraction de caractéristiques et de modélisation des récompenses ou non. La composition de chaque agent est différente, toute comme le sont ses méthodes d'apprentissage, d'inférence et de pré-entraînement.

D'ailleurs la méthode de pré-entraînement est encapsulée dans un module qui peut être utilisé par un agent. Ceux-ci l'appele en fournissant un réseau neuronal à entraîner. Ainsi, un DQN pré-entraîne son réseau principal en faisant une copie dans son réseau *target*. Un agent dispose aussi de méthodes de chargement et de sauvegarde d'un réseau de neurone dans un fichier qui peut être nommé. Il est donc possible de mettre sur pause l'entraînement d'un agent pour le reprendre plus tard. La méthode de *behavioral cloning* peut aussi être appelée à tout moment, même si la raison de son existence est d'aider l'agent à surpasser les premiers plateaux au début de l'apprentissage.

3.3 Implémentation

3.3.1 Pipeline de partage de données

L'objectif du pipeline de données est de recueillir les données des parties complétées par les joueurs en un endroit centralisé pour construire le dataset qui sera utilisé pour l'entraînement de notre agent intelligent. Tel que mentionné ci-haut, nous voulons recueillir les informations d'états de la grille au courant de la partie ainsi que les actions prises par le joueur dans des fichiers JSON. Nous voulons également conserver un historique des scores des parties complétées. De la sorte, nous serons en mesure de conserver uniquement les parties où le score est supérieur à un seuil déterminé en s'assurant ainsi que l'agent peut s'entraîner avec des données pertinentes.

Nous avons décidé d'utiliser le service de stockage infonuagique d'AWS S3 pour conserver les données des parties complétées. En effet, une fois la partie complétée par le joueur, un fichier JSON pour les états de la grille et un fichier JSON pour les actions prises sont téléversés dans un Bucket dédié et le fichier d'historique de score est mis à jour avec le score de la partie terminée. Pour ce faire, nous avons dû premièrement créer un compte AWS, créer une instance de S3 et créer le Bucket qui est utilisé pour le stockage des fichiers. Il a fallu ensuite créer un rôle dans IAM pour notre API Gateway, créer une politique rattachée à ce rôle pour permettre le téléversement et le téléchargement de fichiers et finalement mettre en place et configurer un REST API pour opérer les requêtes PUT et GET nécessaires. Les étapes à suivre sont décrites de façon détaillée dans la documentation offerte par AWS[13]. Les fichiers JSON générés sont nommés soit states ou actions suivis du moment exact où la partie est complétée (yyyymmddHHMMSS) pour pouvoir facilement les identifier. Cet identifiant est également utilisé pour enregistrer le score de la partie.

Les joueurs peuvent maintenant utiliser deux interfaces pour la cueillette de données de jeux : soit la version en Julia dans le projet même, soit la version web. Les deux versions utilisent le même pipeline de données et génèrent les mêmes types de données. La version dans le projet utilise le package Julia GameZero. Pour ce faire, nous utilisons deux fils d'exécutions : le premier s'occupe de générer l'interface de jeux et le second s'occupe de faire les opérations de téléversement vers AWS S3. En effet une fois la partie terminée, le joueur doit appuyer sur la touche p pour en lancer une nouvelle. C'est à ce moment que les données de la partie sont converties en format JSON et envoyées vers notre Bucket et que le fichier de scores est mis à jour. Ces fichiers sont également stockés localement si jamais l'utilisateur veut les consulter. Nous avons divisé l'exécution sur deux fils séparés puisque les opérations de téléversement peuvent prendre quelques secondes avant d'être complétées et nous ne voulons pas que le programme bloque lors du lancement d'une nouvelle partie. L'interface web quant à elle est implémentée en HTML et Javascript, tel que décrit dans la section ci-bas.

3.3.2 Interface web pour accès grand public

Tel que mentionné précédemment, l'accès à l'interface de jeux Julia passe par le projet directement. Ceci limite le nombre de joueurs et du même coup la quantité de données que l'on peut recueillir pour notre dataset puisque le joueur doit avoir Julia d'installé sur sa machine et avoir accès au code source du projet. Pour démocratiser l'accès à l'interface de jeux, nous avons implémenté une version accessible par un navigateur web. Pour ce faire, nous avons récupéré un projet Tetris existant sur GitHub conçu en HTML et JavaScript[14]. Quelques modifications à la mécanique de jeux ainsi qu'à la logique interne du programme ont dû être apportées pour correspondre à la version existante en Julia utilisant le package GameZero. Les deux version utilisent le même pipeline de partage de

```julia
function upload_data()

    for data in data_list
        arr = []

        suffix = data["suffix"]
        score = data["score"]
        stateFile = data["stateFile"]
        actionFile = data["actionFile"]
        stateFileName = data["stateFileName"]
        actionFileName = data["actionFileName"]
        states = data["states"]
        labels = data["labels"]

        open(stateFileName, "w") do f
            for (idx, state) in states
                state = Dict("state$idx" => state)
                push!(arr, state)
            end
            S3.put_object(BUCKET_NAME, stateFile, Dict(
                "body" => JSON.json(arr),
                "Content-Type" => "application/json"))
            JSON.print(f, arr)
        end

        empty!(arr)
        open(actionFileName, "w") do f
            for (idx, label) in labels
                label = Dict("label$idx" => label)
                push!(arr, label)
            end
            S3.put_object(BUCKET_NAME, actionFile, Dict(
                "body" => JSON.json(arr),
                "Content-Type" => "application/json"))
            JSON.print(f, arr)
        end
        scoreboard_append(suffix, score)
    end
    empty!(data_list)
end
        You, 4 weeks ago • Created thread for data upload to s3
function scoreboard_append(name, score)
    content = S3.get_object(BUCKET_NAME, SCOREBOARD, Dict("response-content-type" => "text/plain"))
    new_score = "$content\n$name : $score"
    S3.put_object(BUCKET_NAME, SCOREBOARD, Dict(
        "body" => new_score,
        "Content-Type" => "text/plain"))
end
```

Figure 6: Module de partage des données (Julia).

données et produisent les mêmes sorties. Voici la liste des modifications importantes que nous avons apportées :

1. Ajout de la rotation anti-horaire
2. Ajout du hard-drop
3. Ajout de la mise en attente d'une pièce et son affichage
4. Enregistrement des trois prochaines pièces à venir
5. Enregistrement des actions et des états de la grille
6. Connexion au pipeline de données
7. Affichage du top 20 des joueurs

La connexion au pipeline de données dans l'interface web passe par des requêtes HTTP pointant vers des fonctions AWS Lambda que nous avons mise en place. En effet au début d'une partie, une requête GET utilisant l'URL de notre première fonction va consulter le fichier des scores enregistrés pour ensuite afficher les meilleurs 20 scores des joueurs. La seconde est une requête POST en fin de partie utilisant l'URL de notre seconde fonction et contient les données de jeux, à savoir les actions, les états, le score, l'adresse courriel du joueur et le nombre de rangées complétées. Cette information est traitée et est ensuite téléversée vers notre Bucket S3 selon la même logique que la version de jeux Julia Gamezero.

Nous avons fait l'acquisition d'un domaine chez Namecheap et le site est hébergé gratuitement chez Netlify. Les joueurs peuvent avoir accès à l'interface de jeux via leur navigateur web préféré en suivant l'URL ci-bas :
`https://tetrisaitrainer.com/`

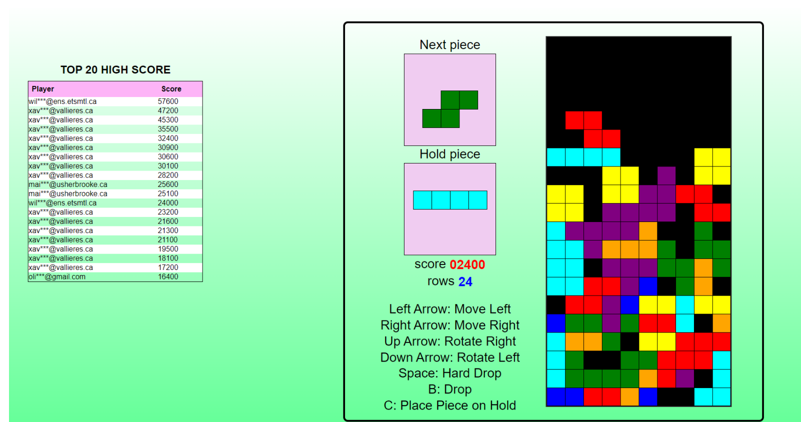


Figure 7: Interface Tetris web.

3.3.3 Campagne de promotion

Pour la création de notre dataset pour l'apprentissage par imitation, notre objectif était de récolter le plus de parties possible provenant de joueurs réels dans la période de temps nous étant disponible. De là provenait à la base notre décision d'implémenter la version web de notre interface de jeux. Une fois celle-ci complétée, nous avons utilisé différentes stratégies pour faire la promotion de notre projet et pour inciter de nouveaux utilisateurs à jouer des parties en utilisant la plateforme web.

La principale stratégie mise en place a été l'élaboration d'un concours pour les utilisateurs. En effet, le concours promettait la remise d'une carte-cadeau de la SAQ d'une valeur de 40$ pour le meilleur pointeur en date du 20 avril 2023. Nous croyons que cet incitatif a été bénéfique envers l'engouement général face à notre projet ainsi qu'au nombre de parties totales enregistrées. La promotion du projet et du concours a passées par des annonces sur les médias sociaux, une annonce dans l'infolettre de l'AGES (L'Association Générale Étudiante en Sciences), des annonces en classe et par bouche à oreille. Au total, nous avons pu récupérer environ 600 parties provenant de joueurs réels, desquels une centaine a été utilisée pour former notre premier dataset.

3.3.4 Module d'extraction de caractéristiques

,

Comme expliqué en Section 3.2.3, le module d'extraction de caractéristiques permet de produire une représentation de la grille de jeu, la grille de caractéristiques, où chaque type de cellule est identifié. Cette grille est utilisée par deux fonctions, `get_state_features` et `computeIntermediateReward`, chacune utilisée pour la création d'un vecteur de caractéristiques et la modélisation des récompenses respectivement.

La création de la grille de caractéristiques est exécutée par la fonction `get_feature_grid`, qui identifie chaque type de cellule séquentiellement. Au départ, la grille de caractéristiques est considérée comme remplie de cellules pleines. Ensuite, la fonction récursive `flood_cell` implémente un algorithme de remplissage par diffusion pour identifier toutes les cellules vides qui sont accessibles à partir du haut de la grille, en comparant avec la grille de jeu originale. Les cellules qui sont vides sur cette grille originale et qui reste remplies sur la grille de caractéristiques sont identifiées comme des trous. Ensuite, toutes les cellules vides dont il existe une cellule pleine au-dessus dans la même colonne sont identifiées comme des encoches. Pour finir, les cellules vides restantes dont les deux cellules du haut sont aussi vides et ont des cellules pleines à chaque côté sont identifiées comme des crevasses. Le résultat est une grille dont chaque cellule est identifiée par un chiffre relevant de son type. Celle-ci peut d'ailleurs être affichée au terminal pour des fins de débogage.

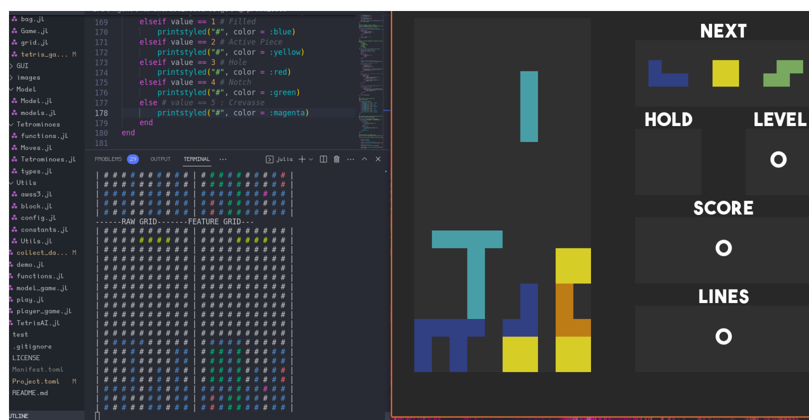


Figure 8: Grille de caractéristiques affichée au terminal, comparée à la grille "crue" à sa gauche. À droite, la fenêtre de jeu qui est rendue en simultané.

Après la génération de cette grille, de simples calculs statistiques (nombre, moyenne, etc.) peuvent être faits sur celle-ci afin de générer un vecteur avec ces informations, soit le vecteur de caractéristiques pondéré qui peut être donné aux réseaux neuronaux des agents. Afin d'encoder l'état de l'environnement sous une forme plus représentative, une alternative à ce vecteur de caractéristiques est l'utilisation d'un réseau de neurones à convolutions. Celui-ci peut être enchaîné par la suite à un réseau profond, où même être utilisé directement pour la classification état-action. Ainsi, des modèles CNN tels que `conv_net` ou `ppo_shared_layers_conv` sont disponibles pour être utilisés par les agents.

Outre cette fonctionnalité, les informations extraites de la grille de caractéristiques servent à la modélisation des récompenses. Un premier type de modélisation des récompenses a été implémenté en utilisant les mêmes modules que l'extraction de caractéristiques, mais en utilisant ces caractéristiques d'une manière un peu différente. À chaque tick du jeu, on calcule un "score" de l'agent qui est implémenté exactement comme la fonction suivante [15]:

$$score = (-0.510066 \times hauteur) + (0.760666 \times lignes) + (-0.35663 \times nb_trous) + (-0.184483 \times bumps)$$

Ce score étant calculé, on considère la récompense donnée à l'agent comme étant la variation de ce score à chaque tick. Si ce score ne change pas, alors aucune récompense n'est donnée. Notez ici qu'on pénalise l'agent lorsqu'il introduit des trous, qu'il augmente la hauteur ou qu'il augmente la *bumpiness*. Comme ce calcul de récompenses est seulement intermédiaire afin de guider l'agent vers les "vraies" récompenses qui sont la complétion des lignes, on introduit une constante Ω (au départ $\Omega = 0$) qui permet de donner de plus en plus de poids aux lignes complétées à mesure que des lignes sont complétées. À chaque tick où on complète une ligne, on incrémente Ω de 0.1. On a donc l'équation suivante qui est implémentée au final:

$$reward = (1 - \Omega) \times (score_{t-1} - score_t) + \Omega \times (lignes)^2$$

Ainsi, après 10 ticks où l'agent a complété des lignes, on ne donne plus de récompenses intermédiaires et on favorise seulement les récompenses données au nombre de lignes complétées au carré. On élève le nombre de lignes au carré puisque la complétion de plusieurs lignes à la fois est favorable et devrait être un comportement encouragé.

3.3.5 Classe Agent

L'interface d'un agent `AbstractAgent`, est un type abstrait dont tous les agents héritent. Un agent est représenté par un `mutable struct` qui est une composition de plusieurs champs, ceux-ci contenants les informations et données pertinentes à l'implémentation des algorithmes utilisés par l'agent.

Tous les agents sont définis dans leur propre fichier, et partagent tous des méthodes du même nom, comme `train!`, `update!` ou `get_action`. Grâce au principe de *multiple dispatch*, une des particularités intéressantes du Julia, il est possible que des méthodes du même nom s'exécutent selon différentes classes, ce qui ressemble beaucoup au polymorphisme qu'on retrouve en programmation OO conventionnelle. Pour ce faire, par exemple, la méthode `train!` définie pour un DQNAgent doit recevoir un DQNAgent en argument.

Pour les méthodes globales, comme certaines définies dans le fichier `functions.jl`, un agent de type `AbstractAgent` doit être passé en argument, donc tous les agents héritant de ce type peuvent faire usage de cette fonction. Un principe similaire pourrait être appliqué dans le futur où par exemple certaines méthodes seraient réservées seulement à des agents qui implémentent des algorithmes MDP. Pour le pré-entraînement, la fonction `pretrain!` a été déplacée dans un module distinct dans le fichier `Agent/behavioral_cloning.jl`. Ce fichier pourra contenir des méthodes de pré-entraînement différentes si le besoin se présente dans le futur, advenant que certains algorithmes fonctionnent d'une manière particulière.

Dans le cadre de la recherche (expliquée en détail en section 4), deux algorithmes implémentent cette classe: DQN et PPO. Ceux-ci implémentent chacun les méthodes définies par l'interface abstraite et d'autres spécifiques à leur fonctionnement. Les champs dont leur structure est composée

```julia
Base.@kwdef mutable struct DQNAgent <: AbstractAgent
    type::String = "DQN"
    n_games::Int = 0
    record::Int = 0
    feature_extraction::Bool = true
    n_features::Int = 17
    reward_shaping::Bool = false
    reward_shaping_score::Float64 = 0.
    ω::Float64 = 0                  # Reward shaping constant
    η::Float64 = 1e-3               # Learning rate
    γ::Float64 = (1 - 1e-2)         # Discount factor
    τ::Float64 = 5e-3               # Soft update rate
    ϵ::Float64 = 1                  # Exploration
    ϵ_decay::Float64 = 1
    ϵ_min::Float64 = 0.05
    batch_size::Int = 128
    memory::CircularBuffer = ReplayBuffer(DQN_Transition).data
    policy_net = (feature_extraction ? TetrisAI.Model.dense_net(
        n_features) : TetrisAI.Model.conv_net()) |> device
    target_net = (feature_extraction ? TetrisAI.Model.dense_net(
        n_features) : TetrisAI.Model.conv_net()) |> device
    opt::Flux.Optimise.AbstractOptimiser = Flux.ADAM(η)
    loss::Function = logitcrossentropy
end
```

Figure 9: Définition du struct DQNAgent.

diffèrent selon les hyperparamètres spécifiques et les modèles utilisés. Par exemple, `DQNAgent` utilise les méthodes `remember`, `experience_replay` et `soft_target_update`. Respectivement, ces méthodes permettent d'enregister une transition dans le tampon mémoire (3.3.7), d'apprendre sur des échantillons de ce même tampon, et finalement de copier les poids du réseau utilisé pour l'inférence sur les poids du réseau *target*. Voir 4.1.1 pour des détails théoriques.

3.3.6 Affichage de benchmarks

Les graphiques de *benchmark* forment une composante centrale à l'entraînement et à l'amélioration de nos agents. Nous avons donc décidé d'implémenter une manière modulaire de pouvoir visualiser nos résultats en temps réel durant l'entraînement. Cela nous permet d'évaluer et de comparer facilement la performance et les courbes d'apprentissage de nos agents. Nous avons implémenté l'affichage avec une structure modulaire afin de pouvoir potentiellement introduire d'autres types de graphiques et de pouvoir avoir une structure un minimum efficace en utilisation mémoire.

Le code de la figure 10 détaille l'implémentation générale du système de benchmark. À chaque étape de l'entraînement, on fait appel à `append_score_ticks` pour enregistrer l'historique des scores et des `ticks` à afficher et on fait aussi appel à `update_benchmark` afin de mettre à jour l'affichage du graphique selon le paramètre `update_rate` afin de ne pas trop ralentir l'entraînement pour l'affichage d'un graphique. De plus, il est possible d'enregistrer les données de l'entraînement dans un fichier CSV afin de pouvoir analyser les courbes d'apprentissage et de pouvoir les exporter comme une image.

```julia
Base.@kwdef mutable struct ScoreBenchMark
    n::Int64
    linecolor               = [:orange :blue]
    labels::Array{String}   = ["Scores" "Ticks" "Rewards"]
    xlabel::String          = "Iterations"
    ylabel::String          = "Score"
    graph_steps::Int64      = ceil(n / 10)
    df::DataFrame           = DataFrame(Scores = Int64[], Ticks = Int64[
        ], Rewards = Float64[])
    i::Int64                = 0
    current_max_y           = 0 # Used for ylims when plotting
    current_min_y           = 0 # Used for ylims when plotting
    xticks                  = 0:graph_steps:n
    linewidth               = 2
end

function append_score_ticks!(b::ScoreBenchMark, score::Int64, tick::
     Int64, reward::Float64 = 0.)
    push!(b.df, [score, tick, reward])
    b.i += 1
    biggest = max(score, tick, reward)
    smallest = min(score, tick , reward)
    b.current_max_y = max(biggest, b.current_max_y)
    b.current_min_y = min(smallest, b.current_min_y)
end

function update_benchmark(b::ScoreBenchMark, update_rate::Int64, iter,
     render::Bool = true)
    if (b.i % update_rate) == 0
        if render
            plot(Matrix(b.df),
                xlims=(0, b.n),
                xticks=b.xticks,
                ylims=(b.current_min_y, b.current_max_y),
                title=string("Agent performance over ", b.n, " games"),
                linecolor = [:orange :blue :green],
                linewidth = b.linewidth,
                label=b.labels)
            xlabel!("Iterations")
            display(plot!(legend=:outerbottom, legendcolumns=3))
        else
            means = mean.(eachcol(last(b.df, update_rate)))
            println(iter, "Avg score: ", means[1], " Avg ticks: ", means
                [2], " Avg rewards: ", means[3])
        end
    end
end
```

Figure 10: Système de benchmark.

3.3.7 Tampons de mémoire

Le fichier `memory.jl` permet de définir tout type d'enregistrement d'expériences disponible pour l'utilisation par les agents. Dans ce fichier, deux types de structures sont définis: une transition et un tampon mémoire. Une transition est une représentation d'une expérience sous forme de code. Plus précisément, il s'agit d'un `struct` contenant une liste d'informations sur une expérience récoltée par l'agent. Par exemple, `DQN_Transition` est une composition de plusieurs champs pertinents à son implémentation. Des détails sur le raisonnement derrière leur utilisation sont expliqués en section 4.1.1. Outre la transition, l'autre structure implémentée dans le fichier `memory.jl` est le tampon mémoire. Celui-ci est de type `CircularBuffer`, permettant une rotation de ses éléments lorsque sa capacité maximale est atteinte. Ainsi, dans le cas de DQN, le tampon mémoire prend le rôle

du *replay buffer* permettant à un agent *off-policy* de réutiliser ses expériences pour l'apprentissage. Pour PPO, le tampon est initialisé à une taille T, soit l'horizon des expériences récoltées pour un déroulement (*roll-out* - Voir section 4.1.2 pour plus de détails).

```
Base.@kwdef struct DQN_Transition <: AbstractTransition
    state::Union{Array{Int64,3},Array{Float64,1}}
    action::Array{Int64,1}
    reward::Float64
    new_state::Union{Array{Int64,3},Array{Float64,1}}
    done::Bool
end
```

Figure 11: Structure de transition pour un agent DQN.

3.4 Déploiement du Package Julia

3.4.1 Documentation

Tel que mentionné dans la section de l'analyse critique, nous avons utilisé le package Julia Documenter.jl pour générer une plateforme de documentation pour notre projet. Ce package traverse l'entièreté du projet pour retrouver les docstrings placées au-dessus de nos fonctions et de nos objets selon la forme retrouvée dans cet exemple :

```
"""
    process_state(agent::AbstractAgent, state::Vector{Int64})

Transform the state into an array of features or layers that can be fed to a CNN
    .
"""
function process_state(agent::AbstractAgent, state::Vector{Int64})
[...]
```

Figure 12: Exemple de fonction avec commentaires en docstring

Le package s'occupe de retracer toutes les sections documentées du code et de générer des blocs de documentation suivant la structure que nous avons déterminée pour ensuite créer les pages HTML en conséquence. Chaque contributeur au projet est invité à développer davantage la documentation et d'y ajouter des précisions au besoin. Il n'a qu'à suivre les normes docstring Julia et suivre le guide d'utilisation du package Documenter.jl[16]. Nous avons créé un sous-domaine dédié à cette documentation utilisant le domaine que nous avons acquis pour implémenter notre interface de jeux web. Le lien est accessible sur la page README.md du projet :
`https://docs.tetrisaitrainer.com/`
La documentation est sectionnée selon les différents modules que nous utilisons à travers le projet:

1. Home : module principal TetrisAI où l'on retrouve les fonctions principales du projet
2. Agent : module Agent et ses fonctions ainsi qu'une sous-section pour chaque type d'agent

3. Game : module Game où la mécanique de jeux et les tetrominoes sont définis
4. GUI : module d'interface de jeux où le package GameZero est utilisé
5. Model : module Model qui est utilisé pour enregistrer et charger les modèles sous forme binaire
6. Util : module Util qui définit différents paths, constantes et les fonctions d'accès aux services AWS S3.

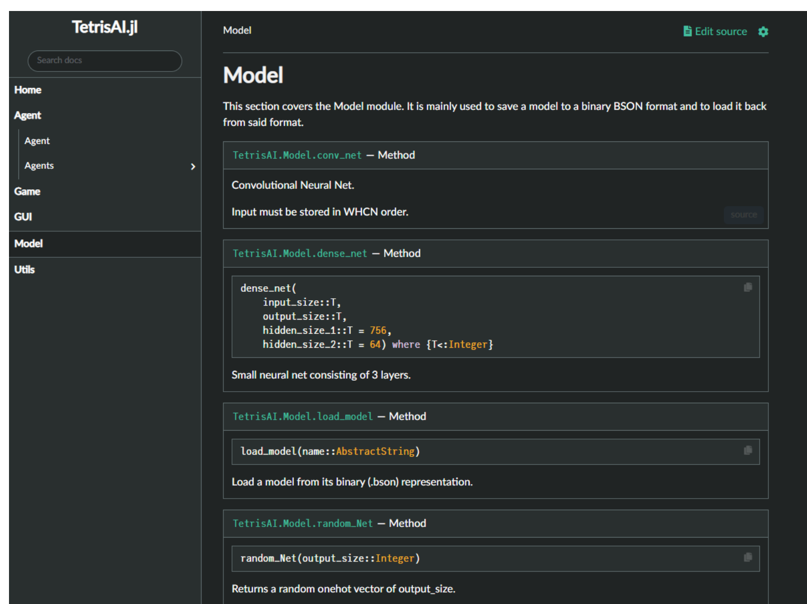


Figure 13: Exemple de page de notre documentation

3.4.2 Publication

L'un des objectifs du projet était de faire la publication officielle du package TetrisAI.jl selon les instructions données par la communauté des développeurs Julia[17]. Selon la documentation officielle, l'enregistrement d'un nouveau package Julia public passe par cinq étapes : premièrement la création d'un package en local selon le format défini, ensuite la création d'un répertoire GitHub pour notre package, suivi de l'écriture des modules de code, suivi de l'implémentation d'un module de tests et finalement l'envoi d'une demande officielle d'enregistrement du package à la communauté de développement. Les étapes 1 et 2 ont été complétées en entier, tandis que l'étape 3 d'écriture de code est toujours en cours. Celle-ci n'étant pas tout à fait complète et étant la principale source de notre attention pour le développement du package, nous n'avons pas eu la chance de développer un module de tests pour l'instant. Il s'agit d'une étape clé qui doit être complétée avant l'envoi d'une demande d'enregistrement. Néanmoins, le package TetrisAI.jl est disponible sur un répertoire GitHub public, donc la communauté est en mesure et est invitée à apporter sa contribution à l'amélioration du projet et éventuellement à la mise en place des tests nécessaires pour l'enregistrement officiel de notre package.

4 Volet de Recherche

4.1 Revue de la littérature

Nous avons effectué une exploration des méthodes déjà proposées dans un environnement similaire à notre environnement Tetris afin de sélectionner des méthodes prometteuses pour la résolution de notre problème. Avec ces algorithmes, nous mettons à l'épreuve ces différentes méthodes afin de voir laquelle performe le mieux ou même si l'une d'entre elles peut compléter quelques lignes et obtenir un score plus grand que zéro en moyenne.

Plusieurs tentatives de résoudre Tetris avec l'apprentissage par renforcement existent dans la littérature et sur le web. Certaines implémentations utilisent un environnement ou un espace d'action simplifié [18, 19, 20, 21], permettant d'atteindre des performances impressionnantes. D'autre utilisent des techniques de recherche comme le Monte-Carlo Tree Search ou d'autres types de recherche heuristique basés sur la simulation des états futurs[22, 23]. Lorsque l'environnement et l'espace d'actions sont identiques à ceux utilisés par un humain, les résultats sont plutôt moindres [24].

Dans le cadre de ce projet, les algorithmes de recherche heuristiques ne seront pas explorés étant donné les contraintes de temps et de ressources à notre disposition. L'accent sera alors plutôt mis sur les algorithmes MDP, qui nécessitent d'ailleurs une restructuration plus légère de la base de code.

4.1.1 DQN

Le Deep-Q-Learning [25] est la première technique à avoir percé dans le domaine de l'apprentissage par renforcement profond, ayant été utilisé par DeepMind en 2013 sur Atari [26], et ensuite en 2015 avec AlphaGo [1], donnant lieu à des résultats prometteurs. Il s'agit maintenant d'un des algorithmes les plus populaires dans le domaine, notamment grâce à sa simplicité. En bref, le DQN est une extension du Q-learning, venant de la famille des techniques de différence temporelle (TD), où l'évaluation d'une paire état-action est approximée par un réseau de neurones profond.

Le principe du Q-learning est basé sur l'idée de la maximisation des récompenses cumulatives, qui peuvent être définies par ce qu'on appelle le *retour*. Formellement, le retour R est défini par $R = \sum_t \gamma^t r_t$, où r_t est la récompense au temps t et γ un facteur d'actualisation entre 0 et 1 permettant de pondérer l'importance des récompenses futures. Afin de maximiser les retours, on définit une fonction sur une paire état-action telle que $Q : S \times A \rightarrow R$ puisse indiquer les retours obtenus lorsqu'une action a est prise à l'état s. Ainsi, il est possible d'établir une politique la politique suivante:

$$\pi(s) = \text{argmax}_a Q(s, a)$$

qui détermine donc l'action la plus avantageuse à prendre selon l'état observé. Cependant, étant un algorithme *model-free*, le Q-learning ne détient pas ces valeurs initialement et doit donc les apprendre. Il faut donc estimer la fonction Q à l'aide d'échantillons Monte-Carlo tirés des expériences de l'agent dans l'environnement. Il est alors possible d'évaluer $Q(s, a)$ en fonction des récompenses obtenues lors d'une expérience ainsi que de l'estimation des retours pour les états subséquents, ce qui donne

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

représentant la mise à jour de la fonction Q. Avec ceci, il est alors possible d'obtenir l'erreur TD:

$$\delta = \underbrace{r + \gamma \max_{a'} Q(s', a)}_{\text{Cible}} - \underbrace{Q(s, a)}_{\text{Prédiction}}$$

qui est la base de la perte devant être minimisée, celle-ci étant L2 dans la plupart des cas. La fonction Q est alors approximée par un réseau neuronal θ sur lequel on utilise la perte pour faire la descente de gradient.

En général, l'implémentation d'un DQN implique l'utilisation de tampon mémoire de relecture (*replay buffer*). Celui-ci permet de stocker les tuples de transitions (s_t, a_t, r_t et s_{t+1}) qui peuvent alors être échantillonnés. Ceci apporte deux avantages. Premièrement, les données brutes des expériences peuvent être réutilisées pour l'apprentissage, permettant de réduire le coût de la génération d'expériences. D'autre part, lors de l'apprentissage profond par descente de gradient stochastique, les données se doivent d'être indépendantes, alors que les expériences générées représentent une séquence d'états subséquents qui sont donc corrélés, ce qui peut déstabiliser l'apprentissage. Il est plus pratique de stocker les transitions dans le tampon de relecture pour pouvoir les échantillonner aléatoirement par la suite. Ainsi, à chaque transition, l'expérience n'est pas directement utilisée mais plutôt ajoutée au tampon, dans lequel un batch d'expériences est tiré au hasard pour l'apprentissage.

Une autre corrélation qui mène à une certaine instabilité prend forme dans la formule de perte, où les deux termes provenant de Q sont dépendants du même réseau θ. Une solution à ce problème est d'utiliser un réseau auxiliaire θ^- qui est figé et utilisé pour prédire la cible (*target*). Celui-ci est mis à jour avec les poids du réseau de prédiction régulièrement, soit à un nombre prédéterminé de C étapes. Ainsi, la prédiction doit se rapprocher d'une cible stationnaire indépendante des apprentissages découlant d'expériences récentes, stabilisant l'apprentissage. (Lillicrap *et al.*, 2016)[27] introduit l'algorithme Deep Deterministic Policy Gradient(DDPG) qui utilise la même technique, mais avec une version alternative de la mise à jour du réseau *target*, où un hyperparamètre τ permet de faire un *soft-update*: au lieu de mettre à jour θ' à chaque C étapes, les mises à jour sont graduelles. Celles-ci sont définies par la formule suivante:

$$\theta^- = \tau\theta + (1 - \tau)\theta^-$$

Cette technique permet de stabiliser l'apprentissage avec des mises à jour constantes et monotones. Nous avons choisi d'ajouter cette méthode au sein de notre implémentation de DQN dans l'espoir de maximiser nos résultats.

Enfin, la politique utilisée couramment avec un DQN est appelée ϵ-*greedy*, où un hyperparamètre ϵ détermine une probabilité de prendre une action aléatoire, au lieu de toujours choisir l'action étant estimée comme la plus avantagée par le Q-network. Cette technique assure une exploration de l'environnement afin d'éviter l'exploitation précoce d'extremums locaux. Au fur et à mesure que la politique converge vers un comportement désiré, le facteur d'exploration peut être réduit à l'aide d'un hyperparamètre auxiliaire, le ϵ-*decay*. Il est alors possible d'établir un point de départ et de fin pour ϵ, en plus d'un taux de dégradation.

```julia
Base.@kwdef mutable struct DQNAgent <: AbstractAgent
    type::String = "DQN"
    n_games::Int = 0
    record::Int = 0
    feature_extraction::Bool = false
    n_features::Int = 17
    reward_shaping::Bool = true
    reward_shaping_score::Float64 = 0.
    ω::Float64 = 0                # Reward shaping constant
    η::Float64 = 1e-3             # Learning rate
    γ::Float64 = (1 - 1e-2)       # Discount factor
    τ::Float64 = 5e-3             # Soft update rate
    ϵ::Float64 = 1                # Exploration
    ϵ_decay::Float64 = 1
    ϵ_min::Float64 = 0.05
    batch_size::Int = 128
    memory::CircularBuffer = ReplayBuffer(DQN_Transition).data
    policy_net = (feature_extraction ? TetrisAI.Model.dense_net(
        n_features) : TetrisAI.Model.conv_net()) |> device
    target_net = (feature_extraction ? TetrisAI.Model.dense_net(
        n_features) : TetrisAI.Model.conv_net()) |> device
    opt::Flux.Optimise.AbstractOptimiser = Flux.ADAM(η)
    loss::Function = logitcrossentropy
end
```

Figure 14: Définition de la structure d'un agent DQN

4.1.2 PPO

Proximal Policy Optimization (PPO) [28] est un algorithme introduit en 2017 par l'équipe de DeepMind qui démontrait d'excellentes performances sur plusieurs environnements de jeux Atari. Encore aujourd'hui, c'est un algorithme de pointe dans le domaine de l'apprentissage par renforcement grâce à sa facilité d'application dans plusieurs domaines et environnements. En général, l'algorithme ne demande pas beaucoup d'ajustements de ses hyperparamètres pour démontrer des performances notables ce qui fait de lui un algorithme très populaire dans ce domaine.

C'est un algorithme de la famille des *Policy Gradients* ce qui implique que le fondement de l'algorithme est basé sur l'apprentissage direct d'une politique stochastique au lieu d'apprendre une fonction d'approximation de valeur pour en déduire la politique. Bien sûr, les valeurs des paires états-actions sont tout de même approximées par une fonction afin de pouvoir guider les modifications apportées à la politique puisque l'algorithme respecte une architecture *Actor-Critic* (l'actor est la politique et la critique est la fonction d'approximation de valeur). La fonction de perte de cet algorithme est ce qui le distingue particulièrement des autres *policy gradients*. Dans le contexte du RL, un pas d'apprentissage trop grand peut être fatal pour l'apprentissage de l'agent. Schulman *et al* proposent donc une fonction de perte qui permet de faire des petits pas incrémentaux pour mettre à jour la politique afin de la mettre à jour de façon pessimiste question de ne pas faire un pas d'apprentissage trop grand:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right) \right] \quad \text{où} \quad r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

$r_t(\theta)$ est le ratio de modification de la politique actuelle par rapport à la politique précédente et $\hat{A}_t$ est l'avantage associé à l'action qui détaille la qualité de l'action prise avec la différence entre l'estimation des valeurs prédites pour l'état actuel et l'état précédent. L'objectif de cette

équation est donc de maximiser $r_t(\theta)\hat{A}_t$, cependant, on peut facilement remarquer le terme $\text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)$ qui permet de limiter le ratio de changement de politique à l'intervalle $[1-\epsilon, 1+\epsilon]$ qui représente notre intervalle de confiance pour mettre à jour la politique (ϵ fonctionne en général avec une valeur de 0.2). Ça permet donc de limiter nos mises à jour ce qui résulte en un apprentissage plus stable et qui converge plus rapidement.

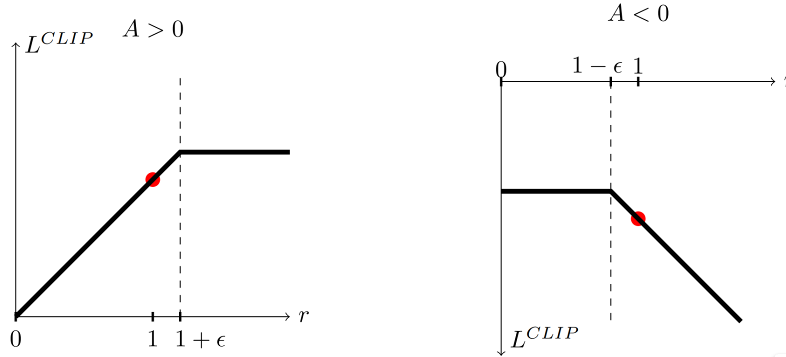


Figure 15: Illustration du clip tiré de l'article de référence de PPO. [28]

L'équation de la fonction de perte qui a été implémentée dans notre module est la méthode un peu plus générale qui complémente la fonction de perte ci-haut:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S\left[\pi_\theta\right](s_t) \right],$$

où c_1 et c_2 sont des constantes, L_t^{VF} est une perte de moindres carrés sur $V(s)$ et $S[\pi_\theta](s_t)$ est l'entropie de la politique $pi\theta$. L'algorithme est ainsi exécuté pour T transitions, en utilisant $\pi_{\theta_{\text{old}}}$ pour générer les avantages $\hat{A}_t$ obtenues par l'estimation des GAEs qui est détaillée par [29].

Nous avons ainsi implémenté cet agent dans le fichier `ppo_agent.jl` où la définition de ce qu'est un agent a été posée. Tous les hyperparamètres utilisés soin des valeurs comparables à ce qui a été utilisé par les auteurs de PPO dans leurs expérimentations (figure 16), bien que ces hyperparamètres soient facilement modifiables. Des nouvelles structures de réseaux de neurones ont été implémentées afin de pouvoir supporter le partage des premières couches de chacun des réseaux. Comme l'environnement de Tetris est complexe, certains apprentissages faits par le modèle de la politique peuvent aussi être pertinents pour l'estimateur de valeur, d'où l'idée de partager certaines couches entre les modèles. Selon notre implémentation, l'agent récolte des informations sur son épisode dans sa mémoire durant la durée d'un horizon et lorsqu'il atteint la fin de l'horizon ou la fin d'un épisode, l'agent entraîne itérativement ses modèles de valeur et de politique sur les informations récoltées durant sa trajectoire jusqu'à une divergence KL cible, dans quel cas, l'épisode reprend ou une nouvelle partie recommence si l'épisode était terminé. De plus, dans l'optique d'un algorithme *policy gradient*, l'action sélectionnée dans chaque état est échantillonnée parmi la distribution de probabilités des actions dans l'état courant qui est donnée par la politique.

```julia
Base.@kwdef mutable struct PPOAgent <: AbstractAgent
    type::String                 = "PPO"
    n_games::Int                 = 0    # Number of games played
    record::Int                  = 0    # Best score so far
    feature_extraction::Bool     = true
    n_features::Int              = 17
    reward_shaping::Bool         = true
    shared_layers                = (feature_extraction ? TetrisAI.Model.
         ppo_shared_layers_dense(n_features) : TetrisAI.Model.
         ppo_shared_layers_dense(n_features)) |> device
    policy_model                 = TetrisAI.Model.policy_ppo_net(512, 7)
         |> device
    value_model                  = TetrisAI.Model.value_ppo_net(512) |>
         device
    policy_max_iters::Integer    = 100
    value_max_iters::Integer     = 100
    policy_lr::Float64           = 1e-4
    value_lr::Float64            = 1e-2
    max_ticks::Integer           = 20000
    ε::Float64                   = 0.02 # Clipping value
    γ::Float64                   = 0.99 # Reward discounting
    β::Float64                   = 0.01
    ζ::Float64                   = 1.0
    λ::Float64                   = 0.95 # GAE parameter
    ω::Float64                   = 0    # Reward shaping constant
    reward_shaping_score::Float64  = 0
    horizon::Int                 = 5  # T timesteps
    target_kl_div::Float64       = 0.01
    memory::CircularBuffer       = TrajectoryBuffer(PPO_Transition,
         horizon).data
    policy_optimizer::Flux.Optimise.AbstractOptimiser = Flux.ADAM(
         policy_lr)
    val_optimizer::Flux.Optimise.AbstractOptimiser = Flux.ADAM(value_lr)
end
```

Figure 16: Définition de la structure d'un agent PPO

4.2 Méthodologie

Afin d'évaluer la performance de nos différentes configurations d'agent pour DQN et pour PPO, nous avons entraîné ces types d'agents sur 100 et 10 000 parties respectivement. DQN n'a été entraîné que sur 100 parties puisque même avec ce nombre d'étapes limité, l'entraînement était considérablement plus long qu'avec PPO sur 10 000 parties, notamment à cause de l'utilisation du tampon de relecture à chaque transition. Le temps d'entraînement était le même pour environ toutes les configurations soit environ 4 heures. Afin de possiblement accélérer l'entraînement, nous avons loué un serveur chez Google Cloud, équipé d'une carte graphique Tesla V100-SXM2-16GB, afin d'accélérer le processus d'entraînement pour l'algorithme étant donné que nous avions remarqué un temps d'entraînement plutôt lent sur nos machines personnelles. Cependant, comme le coût de cette machine s'élevait à quelques centaines de dollars pour l'utilisation sur environ 48 heures d'entraînement, nous avons été limités par notre budget sur le temps d'entraînement des différents modèles.

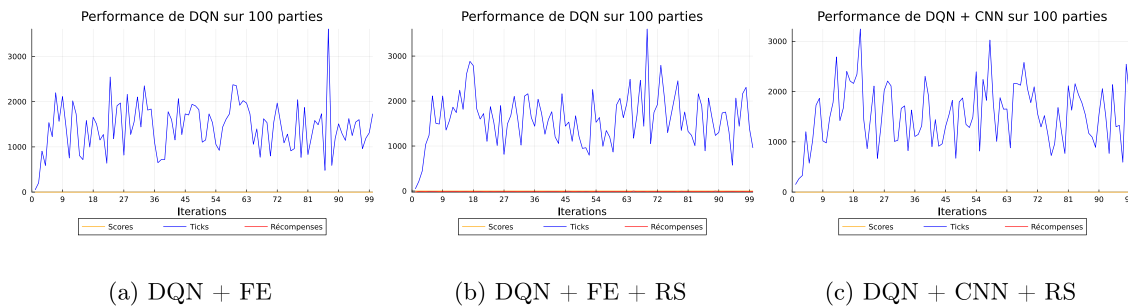
Le système de *benchmark* (Fig. 10) a permis de pouvoir comparer à première vue la performance des agents entre eux. Ainsi, nous avons pu comparer différentes configurations d'algorithmes par rapport à l'évolution de la durée de la partie (nombre de ticks) et même parfois l'évolution des récompenses obtenues. Nous avions à notre disposition aussi l'évolution du score durant l'entraînement, mais comme détaillé dans l'étape suivante, ce n'est pas une métrique qui a été utile pour la comparaison des algorithmes.

De plus, nous avons utilisé toutes les données de parties expertes récupérées par la plateforme en ligne lors de notre concours afin de former notre ensemble d'entraînement final. Un script a été généré afin de sélectionner les meilleures parties jouées et de les conserver dans un nouveau fichier JSON commun pour toutes les parties de l'ensemble d'entraînement. Comme tout se trouve dans un même fichier, nous avons pu mélanger toutes les paires d'état-action entre elles dans l'objectif de maximiser la capacité de généralisation des modèles lors du pré-entraînement. Cependant, un seul petit hic à notre récolte de données s'était présenté: certaines personnes exploitaient l'action *Hold* qui permettait de faire recommencer la pièce au haut de la grille résultant ainsi en des parties où cette action a été utilisée de manière abusive. Donc, nous avons filtré les meilleures parties selon le taux d'action *hold* selon le nombre total d'actions prises dans l'épisode. Si aucun prétraitement n'était fait, l'agent pourrait ainsi apprendre à répéter l'action *hold* afin de survivre le plus longtemps, alors que ce n'est pas le comportement que nous désirons apprendre à notre agent, nous souhaitons qu'il puisse scorer des points le plus efficacement et rapidement possible. Ainsi, nous avons fixé un taux maximum de 5% pour filtrer nos parties afin de se retrouver avec plus de 60 000 paires d'états-actions correspondant ainsi à environ 400 parties. Cet ensemble de pré-entraînement est disponible publiquement pour tous sur le dépôt GitHub du projet. Par faute de temps, aucun résultat n'a été produit pour des algorithmes ayant utilisé le pré-entraînement, même si l'ensemble de données est disponible publiquement.

4.3 Résultats

Nous avons d'abord entraîné les agents DQN avec et sans modélisation de récompenses et avec extraction de caractéristiques (Voir les graphiques à la figure 17). On peut rapidement voir que le score de tous deux des agents est resté à zéro tout au long de l'entraînement, ce qui signifie que l'agent est toujours incapable de compléter une ligne dans l'environnement. Bien que l'entraînement n'était que sur 100 parties, on peut voir que l'algorithme a une tendance à atteindre rapidement un plateau avec une durée qui oscille entre 1000 et 2000 ticks que ce soit avec ou sans modélisation

de récompenses. Les récompenses obtenues lors de l'entraînement des modèles sans modélisation de récompenses sont à zéro puisque l'agent est incapable de compléter une ligne (selon la ligne de score qui reste à zéro pour tous les agents). C'est plus difficilement visible pour DQN avec récompenses modélisées, mais les récompenses obtenues oscillent entre 0 et -10 durant tout l'entraînement. Ainsi, on peut remarquer que la modélisation des récompenses ne semble pas avoir d'impact considérable pour DQN puisque le plateau de la durée de la partie est similaire ce qui était plutôt inattendu, on s'attendait au moins à avoir un petit avantage avec la modélisation de récompenses puisque ça donne plus d'indications à l'agent afin de mieux le guider vers un score non nul. De même, l'utilisation du réseau à convolution (CNN) offre des résultats similaires aux autres entraînements avec extraction de caractéristiques dans la mesure où le nombre de ticks stagne aussi entre 1000 et 2000 après 100 épisodes. Nous nous attendions à des résultats au moins légèrement supérieurs à ceux de l'extraction de caractéristiques puisque l'utilisation de CNN a fréquemment démontré de bonnes performances dans l'extraction de caractéristiques notamment pour la classification d'images.

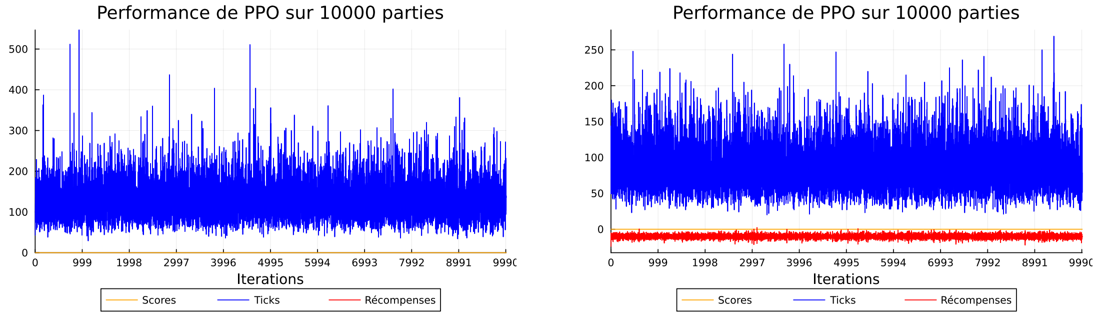


(a) DQN + FE (b) DQN + FE + RS (c) DQN + CNN + RS

Figure 17: Entraînement sur 100 épisodes avec DQN avec ou sans modélisation des récompenses (RS), extraction de caractéristiques (FE) ou avec un CNN.

Les agents PPO ont aussi été entraînés avec des configurations avec et sans modélisation ce qui résulte en une conclusion similaire par rapport à la modélisation de récompenses que pour DQN (voir la figure 18). Le plateau atteint par ces deux configurations pour PPO était pratiquement le même à l'exception que certaines pointes excède 400 ticks pour la configuration sans modélisation de récompenses. De façon similaire à DQN, le score est constamment nul et les récompenses oscillent aussi entre 0 et -10 sauf pour la première configuration où les récompenses restent nulles. Comme PPO est un algorithme ayant démontré des performances supérieures à DQN dans des environnements complexes dans la littérature, nous nous serions aussi attendus à avoir des performances supérieures à DQN. Cependant, le plateau de nombre de ticks pour PPO est environ entre 50 et 175 ticks même après 10 000 parties. Sa performance est donc 10 fois plus faible en nombre de ticks alors qu'il a été entraîné sur 100 fois plus de parties. Cependant, l'entraînement de PPO est beaucoup plus rapide que DQN, possiblement puisqu'il n'utilise pas de replay buffer et possède un horizon relativement petit ce qui permet de l'entraîner plusieurs fois par épisode rapidement.

Nous croyons que le manque de performance des agents entraîné est encore principalement relié à la grande complexité de l'environnement. L'agent semble vraiment avoir de la difficulté à identifier des suites d'actions pertinentes à limiter la hauteur de la grille bien que nous le pénalisions lorsqu'il le fait avec la modélisation de récompenses. Comme nos agents reçoivent ne reçoivent que des récompenses négatives puisqu'ils ne parviennent pas à compléter une ligne, ils semblent ne chercher qu'à repousser le plus possible le moment de leur mort en ne faisant rien la majorité du temps. Il serait donc pertinent d'explorer aussi si un pré-entraînement pouvait considérablement augmenter



(a) PPO + Feature Extraction

(b) PPO + Feature Extraction + Reward shaping

Figure 18: Entraînement sur 1000 épisodes avec PPO

les performances de l'apprentissage de l'agent de sorte qu'il puisse utiliser un point de départ où il est capable de compléter quelques lignes ce qui lui donnerait enfin accès à des récompenses positives et il pourrait potentiellement voir la possibilité de compléter plus de lignes afin de maximiser ses récompenses au lieu de vouloir seulement minimiser ses pertes de récompenses. Aussi, il est possible qu'un point d'apprentissage réel n'ait pas été atteint dû à notre limitation des ressources de calculs. Peut-être qu'en entraînant les agents sur un nombre beaucoup plus grand d'itérations permettrait de surpasser le plateau actuel et d'atteindre un point ou certaines lignes sont complétées, cependant cela nécessiterait beaucoup plus de temps de calcul et ainsi devoir débourser un montant beaucoup plus élevé.

5 Conclusion

Suite à cette deuxième partie du projet, l'entraînement d'un agent Tetris ne semble pas avoir démontré les performances attendues qui devaient être minimalement meilleures que les performances de la première itération. De plus, l'entraînement de PPO offre des performances pires que DQN dans notre environnement après 100 fois plus d'itérations ce qui n'était pas attendu puisque les performances de DQN sont elles aussi très faibles. Par contre, le package de TetrisAI.jl est beaucoup plus complet et facile d'utilisation que la première étape. L'architecture du module est maintenant modulaire ce qui facilite l'ajout d'algorithmes pour toute personne désirant étendre la portée du projet sur de nouveaux algorithmes et le système de *benchmark* facilite l'évaluation rapide de l'entraînement d'un agent. Les quelques bogues réglés dans l'interface ainsi que l'affichage de la grille de caractéristiques permettent d'avoir une visualisation plus robuste de l'agent en action dans l'environnement après l'entraînement. Un ensemble de données pour le pré-entraînement a été récolté et sera publié et libre d'utilisation avec le déploiement de notre *package*. Puisque TetrisAI.jl est publié en accès libre sur GitHub, une documentation automatique a été mise en place afin de pouvoir guider les utilisateurs qui souhaitent l'utiliser dans leur propre projet ou encore pour guider les collaborateurs qui voudraient contribuer au projet.

Le temps et les ressources disponibles pour le développement de ce projet étaient limités alors que les ambitions étaient grandes ce qui nous force à mettre de côté l'implémentation de fonctionnalités pour le moment. Dans un scénario idéal, il aurait été pertinent de faire un entraînement des agents sur beaucoup plus d'itérations afin de confirmer ou d'infirmer la convergence des techniques

utilisées. De plus, une recherche par grille parmi toutes les combinaisons de configurations des agents (i.e. avec/sans modélisation de récompenses, feature extraction ou réseau à convolution, etc.) afin de trouver quelle configuration performe le mieux. De même, peut-être que les hyperparamètres n'étaient pas tout à fait au point, donc une recherche d'hyperparamètres aurait aussi été pertinente pour maximiser les performances. Bien que nous avons récolté et filtré un ensemble de données d'entraînement pour un pré-entraînement supervisé des agents, il n'a pas pu être mis en action pour générer des résultats par faute de temps. Donc, puisque cet ensemble de données sera disponible publiquement, il serait pertinent de tester dans le futur la capacité d'apprentissage des différents modèles suite à un pré-entraînement.

Des apprentissages importants ont été faits durant cette deuxième itération du projet pour tous. Nous avons pu nous familiariser avec le fonctionnement du *multiple dispatch* offert par Julia afin de créer la structure modulaire du code des agents pour ainsi réutiliser au maximum la même structure de code définie pour chaque type d'agent. Nous avons aussi surtout appris en profondeur l'algorithme PPO ainsi que les fondements mathématiques derrière l'algorithme en plus d'avoir une implémentation fonctionnelle pour le projet. Au départ, il était un peu difficile de lire des articles scientifiques et d'en extraire une bonne quantité d'informations pertinentes à cause de leur complexité. Durant la revue de littérature, nous avons pu nous familiariser énormément avec la structure de ce genre d'article et même avec leurs fondements mathématiques qui reviennent souvent dans plusieurs autres papiers reliés. Cela a facilité notre compréhension personnelle de plusieurs concepts en plus de nous aider pour d'autres cours projets où nous devions tirer avantage de certains articles scientifiques d'apprentissage par renforcement. Nous avons aussi pu nous familiariser avec les systèmes infonuagiques qui sont des systèmes en grande expansion dans l'industrie ainsi que pour des outils de documentation automatique ce qui est très pratique pour de futurs projets.

En effet, les performances de nos algorithmes sont non conclusives ce qui ouvre des possibilités de continuation du projet. La majorité des approches d'apprentissage par renforcement très performantes dans des environnements complexes résultent d'une combinaison avec des méthodes heuristiques. Par exemple, AlphaZero [30] et MuZero [31] démontrent des performances impressionnantes en utilisant des simulations Monte Carlo. Nous croyons ainsi que l'utilisation de méthodes heuristiques pourrait permettre de simplifier ainsi l'espace d'états et potentiellement offrir de meilleures performances. De plus, il serait intéressant de supporter l'entraînement sur plusieurs environnements de Tetris en parallèle de sorte à pouvoir accélérer le processus d'entraînement. Ainsi, les perspectives futures de ce projet sont basées sur deux points principaux: l'intégration d'algorithmes de recherche heuristique et l'implémentation d'un système d'apprentissage parallèle distribué.

Le projet étant libre d'accès à tous, le public pourra facilement contribuer ou tester facilement notre module, mais il serait aussi très facile pour une autre équipe de prendre la relève de ce projet dans une session future. Cela permettrait de mettre une fois de plus les agents au défi dans cet environnement si complexe avec de nouvelles méthodes.

References

[1] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), pp. 484–489.

[2] David Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419 (2018), pp. 1140–1144. DOI: 10.1126/science.aar6404. eprint: https://www.science.org/doi/pdf/10.1126/science.aar6404. URL: https://www.science.org/doi/abs/10.1126/science.aar6404.

[3] J. Schrittwieser, I. Antonoglou, and T Hubert. "MuZero: Mastering Go, chess, shogi and Atari without rules". In: *nature* 588 (2020), pp. 604–609.

[4] Erik D Demaine, Susan Hohenberger, and David Liben-Nowell. "Tetris is hard, even to approximate". In: *COCOON*. Springer. 2003, pp. 351–363.

[5] Simón Algorta and Özgür Şimşek. "The game of tetris in machine learning". In: *arXiv preprint arXiv:1905.01652* (2019).

[6] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[7] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[8] Souradip Chakraborty et al. "Dealing with Sparse Rewards in Continuous Control Robotics via Heavy-Tailed Policies". In: *arXiv preprint arXiv:2206.05652* (2022).

[9] S.J. Russell, S. Russell, and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson series in artificial intelligence. Pearson, 2020. ISBN: 9780134610993. URL: https://books.google.ca/books?id=koFptAEACAAJ.

[10] Abhishek Gupta et al. "Unpacking Reward Shaping: Understanding the Benefits of Reward Engineering on Sample Complexity". In: *Advances in Neural Information Processing Systems*. Ed. by Alice H. Oh et al. 2022. URL: https://openreview.net/forum?id=D-X3kH-BkpN.

[11] Jonas Balgaard Amundsen. "A comparison of feature functions for Tetris strategies". In: 2014.

[12] Christophe Thiery and Bruno Scherrer. "Building Controllers for Tetris". In: *J. Int. Comput. Games Assoc.* 32 (2009), pp. 3–11.

[13] Amazon AWS. *How do I upload an image or PDF file to Amazon S3 through API Gateway?* 2022. URL: https://aws.amazon.com/premiumsupport/knowledge-center/api-gateway-upload-image-s3/ (visited on 02/01/2023).

[14] Jake S. Gordon. *javascript-tetris*. https://github.com/jakesgordon/javascript-tetris. 2011.

[15] Yiyuan Lee. Apr. 2013. URL: https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/.

[16] Documenter.jl. *Documenter.jl: Package Guide Manual*. 2023. URL: https://documenter.juliadocs.org/stable/man/guide/ (visited on 04/20/2023).

[17] JuliaRegistrator. *How to develop a Julia package*. 2023. URL: https://julialang.org/contribute/developing_package/ (visited on 04/20/2023).

[18] @nuno-faria. *tetris-ai*. https://github.com/nuno-faria/tetris-ai. 2021.

[19] Rex L. *Reinforcement Learning on Tetris.* 2023. URL: https://medium.com/mlearning-ai/reinforcement-learning-on-tetris-707f75716c37 (visited on 04/20/2023).

[20] S Melax. *Reinforcement Learning Tetris Example.* 2023. URL: https://melax.github.io/tetris/tetris.html (visited on 04/20/2023).

[21] Mohamed Ashry Hassan Mahmoud. "Applying Deep Q-Networks (DQN) to the game of Tetris using high-level state spaces and different reward functions". MA thesis. Ulm University, 2020, p. 112.

[22] @Code-Bullet. *Tetris-AI-Javascript.* https://github.com/Code-Bullet/Tetris-AI-Javascript. 2020.

[23] @hrpan. *tetris mcts.* https://github.com/hrpan/tetris_mcts. 2020.

[24] Karol Kuna. *Learning to Play Tetris using Reinforcement Learning.* https://nlp.fi.muni.cz/uiprojekt/ui/kuna_karol2016/tetris-documentation.pdf. 2016.

[25] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518 (Feb. 2015), pp. 529–33. DOI: 10.1038/nature14236.

[26] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning.* 2013. arXiv: 1312.5602 [cs.LG].

[27] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning.* 2019. arXiv: 1509.02971 [cs.LG].

[28] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[29] John Schulman et al. "High-dimensional continuous control using generalized advantage estimation". In: *arXiv preprint arXiv:1506.02438* (2015).

[30] David Silver et al. "Mastering chess and shogi by self-play with a general reinforcement learning algorithm". In: *arXiv preprint arXiv:1712.01815* (2017).

[31] Julian Schrittwieser et al. "Mastering atari, go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (2020), pp. 604–609.