Rapport de projet :

Introduction

Le jeu Quixo est un jeu de société abstrait se jouant sur un plateau de 5x5 cases. Chaque case est occupée par un cube, chaque cube étant vierge ou marqué d'une croix ou d'un rond, représentant respectivement les deux joueurs. Le but du jeu est simple : réussir à aligner cinq cubes marqués de son symbole (croix ou rond) horizontalement, verticalement ou en diagonale. La particularité de Quixo réside dans la manière dont les cubes sont manipulés : lors de chaque tour, un joueur choisit un cube en périphérie du plateau, le fait pivoter à sa couleur, puis le replace à un autre endroit en faisant coulisser la rangée ou la colonne de cubes. Ce mécanisme introduit un élément de stratégie particulièrement subtil et exigeant.

Dans le cadre de ce projet, en plus de proposer une interaction classique joueur contre joueur, il nous est demandé de développer un agent capable de jouer de manière autonome contre un humain, avec l'objectif de le battre. Cet agent doit être en mesure d'analyser les situations de jeu, de prendre des décisions stratégiques et d'adapter ses actions pour maximiser ses chances de victoire. Afin de garantir une expérience de jeu stimulante, plusieurs approches d'intelligence artificielle ont été envisagées et implémentées.

Ce le rapport de projet suivant concerne le travail fait par Andéol FOURNIER dans le cadre de ce projet. Andéol a exploré des techniques d'apprentissage par renforcement pour développer un agent capable d'apprendre de ses erreurs et d'améliorer sa performance au fil du temps. Il a implémenté un agent utilisant le Q-learning, une méthode classique d'apprentissage par renforcement, ainsi qu'un agent basé sur le Deep Q-learning, qui intègre des réseaux de neurones pour analyser des situations complexes du jeu, ainsi qu’un Monte Carlo Tree Search.

Q_learning :

avant propos :

Pour ce repérer entre les different fichier python utilisé pour le q learning voici un sommaire de ce qu'ils contiennent :

- Le fichie q_learning.py contient la première version du jeu avec la classe agent de q_learning classique et une fonction train pour entrainer les agents.

- le fichier q_learning_parallele.py contient le jeu en deux parties (graphique et logic) avec la classe agent q_learning classique et une fonction train pour entrainer les agents.

- Le fichier q_learning cano_parallele contient le jeu en deux parties (graphique et logic) avec la classe agent q_learning adapté a un représentation canonique des état et des deu fonction train, une classique et une pour parraléliser les entrainement sur les cpu.

- Le fichier Deep_q_learning.py contient le jeu en deux parties (graphique et logic) avec la classe d'un agent de deep q learning (est donc aussi une classe d'un modèle linéaire Pytorch) avec les fonctions nécessaire pour l'entrainer.

-Le fichier Deep_q_learniing_MCTS contient le jeu en deux partie (graphique et logic) avec la class Agent_MCTS_DQL et les fonctions nécessaires pour l'entrainer.

De plus il est necéssaire d'avoir poetry d'installé, d'ouvrir le shell associé à ce projet et de faire un poetry install pour lancer le jeu sans problème de dépendance.

Implémentation initiale du jeu :

La première étape de ce projet Quixo consistait à créer un jeu fonctionnel permettant une interaction fluide. La partie graphique a été réalisée avec Tkinter, une bibliothèque choisie pour sa simplicité, afin de rapidement obtenir une version jouable du jeu. Cela nous a permis de commencer à explorer les possibilités d'intégration d'une intelligence artificielle (IA) rapidement.

L'approche orientée objet de Python a été particulièrement avantageuse dans ce contexte, rendant le jeu facilement configurable et exécutable selon nos besoins. Cela s'est avéré très pratique lors des phases de débogage inévitables. Le jeu repose sur un plateau de 5x5, où les pièces sont déplacées à travers un système de « pousse-pousse ». Seules les cases extérieures peuvent être sélectionnées.

Pour jouer un coup, il faut d'abord choisir une case valide, c'est-à-dire une case située sur la bordure extérieure du plateau, appartenant soit au joueur en cours, soit étant libre (cela est vérifié via la fonction est_premier_carre_intérieur et grâce à la variable d'instance self.current_player). Une fois la case sélectionnée, le principe du « pousse-pousse » est mis en œuvre pour déplacer les autres pièces. Cette mécanique a été implémentée en ajoutant un système de flèches autour des cases extérieures. Ainsi, lors de la sélection d'une case, deux ou trois flèches apparaissent pour indiquer les directions possibles dans lesquelles les pièces peuvent être poussées. Cette logique est

gérée par les fonctions arrow, move_piece, et push, qui permettent de mettre à jour le plateau après chaque coup.

Il est important de noter que le board, la table principale qui contient l'état du jeu est en réalité une matrice de 7x7, où le plateau de jeu (en 5x5) se situe au centre, entouré de cases dédiées à l'affichage et à la sélection des flèches. Une action est donc caractérisée par un triplet (i, j, flèche), où i et j représentent la position de la case sélectionnée, et la flèche indique la direction vers laquelle la ligne sera poussée.

La fonctions reset game et check_victory permettent de voir si le joueur à gagné, d'afficher un message si c'est le cas et de clean le board. Elles sont appeler après chaque coup

Partie IA :

Pour implémenter une IA capable de jouer à ce jeu, l'approche choisie fut d'utiliser l'apprentissage par renforcement, et plus précisément l'algorithme de Q-learning. Cet algorithme permet à un agent (ici, le programme informatique) d'apprendre à prendre des décisions optimales en interagissant avec son environnement de manière autonome.

Principe général du Q-learning

Le Q-learning repose sur la notion de maximisation des récompenses cumulées à long terme. L'agent explore différentes actions possibles à partir d'un état donné et observe les récompenses obtenues après chaque action. L'objectif est de découvrir la séquence d'actions qui maximisera la récompense totale sur le long terme.

Pour cela, l'algorithme construit une Q-table, aussi appelée table des Q-valeurs, qui stocke pour chaque paire (état, action) une valeur Q(s,a)Q(s,a) représentant la "qualité" de l'action a lorsqu'elle est effectuée dans l'état s. Ces valeurs sont mises à jour à chaque itération selon l'équation de Bellman, clé du Q-learning.

L'agent apprend en mettant à jour les valeurs de la Q-table via l'équation de mise à jour suivante (appeler équation de Bellman:

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

- •s : l'état actuel.
- •a : l'action effectuée dans cet état.
- •r : la récompense obtenue après avoir pris l'action aa.
- •s' : l'état résultant après l'action aa.
- •α : le taux d'apprentissage (ou learning rate), qui contrôle la vitesse à laquelle l'agent met à jour ses Q-valeurs.
- •γ : le facteur de réduction (discount factor), qui détermine l'importance accordée aux récompenses futures par rapport aux récompenses immédiates.
- •$\max_{a'} Q(s',a')$: la meilleure valeur d'action possible dans l'état suivant s'.

Cette formule combine la récompense immédiate r et la meilleure récompense possible à long terme (estimée via max_a′Q(s′,a′)), actualisant ainsi les Q-valeurs en fonction de ce que l'agent apprend au fil des interactions.

Politique d’action : l’équilibre entre exploration et exploitation

En apprentissage par renforcement, il est crucial de choisir une politique d'action qui guide l’agent dans sa prise de décision. L'agent doit équilibrer deux comportements antagonistes :

- Explorer : essayer de nouvelles actions pour découvrir des stratégies encore inconnues.
- Exploiter : utiliser les actions déjà connues pour obtenir les meilleures récompenses immédiates.

La politique choisie ici est la politique epsilon-greedy, qui fonctionne ainsi :

- Un paramètre ϵ, compris entre 0 et 1, contrôle le compromis entre exploration et exploitation.
- À chaque étape, un nombre aléatoire est tiré. Si ce nombre est inférieur à ϵϵ, l'agent choisit une action aléatoire parmi celles possibles (exploration). Sinon, il sélectionne l'action qui maximise la Q-valeur (exploitation).

Cette politique permet de moduler l'exploration en ajustant ϵ. En général, ϵ est initialement élevé pour encourager l'exploration, puis diminue au fil du temps pour favoriser l'exploitation des actions optimales découvertes.

Paramétrage et alternatives

Le Q-learning peut être ajusté grâce à des paramètres clés :

- Le taux d'apprentissage αα détermine la vitesse d'adaptation de l'agent.
- Le facteur de réduction γγ fixe la balance entre les récompenses immédiates et futures.
- ϵϵ dans la politique epsilon-greedy ajuste le compromis entre exploration et exploitation.

D'autres politiques peuvent être utilisées, comme :

- La Boltzmann exploration, où les actions sont choisies en fonction de probabilités pondérées par leurs valeurs Q.
- L'Upper Confidence Bound (UCB), une méthode qui favorise l'exploration des actions sous-évaluées tout en maximisant les récompenses.

Ces méthodes alternatives offrent une meilleure adaptation à des environnements spécifiques, mais la simplicité et la flexibilité de la politique epsilon-greedy en font un excellent choix pour une première approche de Q-learning.

L'implémentation de cet agent a été réalisée en créant une classe dédiée. Les variables d'instance représentent les hyperparamètres essentiels tels que α (taux d'apprentissage), γ (facteur de réduction) et ϵϵ (facteur de politique epsilon-greedy). La Q-table est implémentée sous forme de dictionnaire pour stocker les valeurs Q. Les

méthodes update_q_value, choose_action et get_q_value sont responsables des comportements décrits précédemment, assurant la mise à jour des valeurs Q et la sélection des actions appropriées.

La méthode train est utilisée pour entraîner l'agent en lui permettant d'interagir avec l'environnement de manière répétée. À chaque étape, l'agent choisit une action en fonction de l'état actuel, puis la passe à la classe du jeu, qui met à jour le plateau via la méthode step. Ce processus est répété sur de nombreuses parties, l'agent jouant souvent contre lui-même pour s'améliorer. Les progrès sont enregistrés régulièrement sous forme de fichiers Pickle, qui stockent la Q-table.

Récompenses et tests de l'agent

Une fonction de récompense a été implémentée pour guider l'agent dans son apprentissage. Au départ, elle était assez simple :

- une pénalité de -1 pour chaque coup non gagnant,
- une récompense de +50 pour un coup gagnant,
- et un bonus de +25 lorsque le plateau est complètement rempli (une règle ajoutée pour éviter des parties interminables).

Pour tester les performances de l'agent, une première approche a été de jouer, nous personnelement, contre lui. Pour cela, une nouvelle variable d'instance agent a été ajoutée à la classe du jeu, ainsi qu'une méthode ia_play. Cette méthode est appelée lors du tour de l'IA, avec un délai de 500 ms pour rendre l'expérience plus agréable visuellement.

Défis rencontrés

C’est ici que les principales difficultés sont apparues. L'agent ne progressait pas comme espéré et restait peu performant, même face à des stratégies basiques. En observant les valeurs Q à chaque coup, il est rapidement devenu évident que l'agent n’était plus guidé par sa Q-table. Il avançait dans l’inconnu, car il ne reconnaissait pas les états rencontrés, malgré une augmentation du paramètre ϵ et un nombre important de parties jouées.

Optimisations et exploration de nouvelles stratégies

Une première solution envisagée fut de créer deux agents distincts, l’un jouant exclusivement les "X" et l’autre les "O". Une méthode train_rival fut développée pour les faire s’affronter, dans l’espoir de spécialiser chaque agent dans son rôle respectif. Cependant, cela n'a pas suffi à résoudre le problème, et les agents restaient médiocres en phase de test.

Une autre tentative d'amélioration a consisté à augmenter drastiquement le nombre de parties jouées, afin de forcer l'agent à explorer davantage d'états. Pour cela, il a été nécessaire de réduire le temps de calcul par partie. L'optimisation la plus évidente fut de séparer la logique du jeu de l'interface graphique gérée par Tkinter. Ainsi, deux nouvelles classes furent créées : QuixoLogic pour la gestion interne du jeu, et QuixoQame pour la partie graphique. En isolant la partie logique, l'agent pouvait s'entraîner beaucoup plus rapidement, sans être ralenti par le rendu graphique.

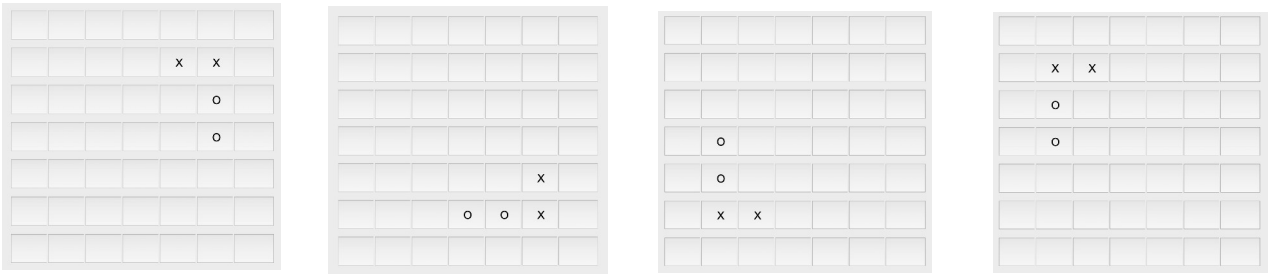
Malgré ces améliorations, les performances de l'agent restaient limitées. Une autre idée fut alors d'exploiter davantage les ressources de l'ordinateur en jouant plusieurs parties simultanément. En

consultant le gestionnaire de tâches, il est apparu que Python n’utilisait pas pleinement les capacités du CPU et de la mémoire. L'objectif était donc d'exécuter des parties en parallèle pour accélérer l’apprentissage. Les bibliothèques functools et multiprocessing furent utilisées pour coder la fonction parallel_train, qui permettait de lancer plusieurs parties en même temps, ainsi que la fonction merge_q_table pour fusionner les résultats des différentes parties.

Cependant, cette approche s'est révélée contre-productive. Les parties jouées simultanément prenaient plus de temps qu'une seule partie jouée de manière séquentielle rendant la parallélisation pas si performante que espéré, et des arrêts inattendus survenaient fréquemment, corrompant parfois les fichiers Pickle contenant la Q-table. Cela rendait ces fichiers illisibles, forçant ainsi à recommencer l’entraînement depuis zéro, ce qui fut extrêmement frustrant. Finalement, cette méthode a été abandonnée.

Il fallait alors trouver une autre méthode. Après visualisation de la taille du dictionnaire enregistrer par pickles, il était clair qu’une méthode direct était impossible ici. En effet la q_table dépacer la centaine de Mega octet et s’approcher du Giga octet sans que l’agent soit plus informé sur les action à prendre.
Il fallait donc trouver une méthode pour réduire la taille de cette q_table et aider l’agent. Elle fut trouver en implémentant une représentation canonique du plateau de jeu. En effet en observant le plateau de jeu, on peut remarquer que certains états sont similaire, à la rotation/simétrie près :

Ici, on voit que les rotation à 90° ne change pas la stratégie



Pour optimiser le processus de décision de notre agent et réduire le nombre d'états uniques à traiter, le concept d'état canonique a été introduit. Un état canonique regroupe sous une seule représentation plusieurs états équivalents du plateau de jeu en tenant compte de ses symétries et rotations.

En effet, un plateau de jeu de 5x5 peut être transformé de diverses façons sans pour autant changer la disposition stratégique des pièces. Plus précisément, chaque état classique du plateau peut être représenté de huit manières différentes : quatre rotations successives de 90° (en comptant l’état de départ), et quatre symétries correspondant aux axes de symétrie du plateau (horizontal, vertical, ainsi que les deux diagonales principales).

Réduction du nombre d'états

Initialement, le nombre d'états possibles sur le plateau de jeu est colossal. Puisque chaque case du plateau peut être occupée par trois valeurs possibles ("X", "O", ou vide) le nombre total d'états distincts est de 3^25, ce qui correspond à environ 850 milliards états différents. En prenant en compte les symétries et rotations, le nombre d'états peut être réduit d'un facteur 8, ce qui divise le

nombre d'états possibles par huit, ramenant ainsi ce total à environ 100 milliards d'états. Cela représente une réduction significative de l’espace de recherche pour notre agent.

Afin d'uniformiser ces états équivalents, une méthode a été choisie pour identifier un seul état canonique représentant toutes les formes équivalentes d’un même plateau. Pour cela, un critère lexicographique fut adopté : parmi les huit représentations possibles d’un même plateau (correspondant aux différentes rotations et symétries), l’état canonique est défini comme étant celui qui apparaît en premier dans l'ordre lexicographique. En pratique, chaque état du plateau est converti en une chaîne de caractères, et l'état canonique est celui correspondant à la chaîne lexicographiquement la plus petite.

Cela garantit que chaque ensemble d'états équivalents est représenté par un unique état canonique, rendant notre Q-table plus compacte et mieux structurée. Ce choix assure non seulement l'unicité de la représentation canonique, mais également sa détermination facile et systématique pour n’importe quel état du plateau.

Pourquoi ne pas inverser les symboles "X" et "O" ?

Une autre possibilité pour réduire davantage l’espace des états aurait été de traiter "X" et "O" comme équivalents par inversion, c'est-à-dire de considérer que l'échange des deux symboles pourrait encore réduire le nombre d'états. Cela aurait effectivement permis de diviser par 16 le nombre d'états distincts (en combinant rotations, symétries et inversions de symboles). Cependant, cette approche aurait rendu le choix lexicographique moins intuitif et plus complexe, car il aurait fallu traiter séparément les différentes permutations de symboles, complexifiant le calcul de l'état canonique. Par souci de simplicité et de cohérence, l’inversion de symboles "X" et "O" ne fut pas choisi , garantissant ainsi que notre méthode de sélection lexicographique reste fiable.

Implémentation technique

Afin de permettre à notre agent d'utiliser cette optimisation, plusieurs méthodes supplémentaires ont été intégrées dans sa classe. Ces méthodes, liées exclusivement à la manipulation et la transformation des états du plateau, sont définies comme des méthodes statiques (@staticmethod) puisqu'elles n'ont pas besoin d'accéder aux données d'instance de l'agent lui-même. Leur rôle est de transformer l'état classique du plateau en son état canonique, facilitant ainsi la recherche dans la Q-table.

Le processus fonctionne ainsi : lorsque l’agent reçoit l’état actuel du plateau de jeu, il doit décider s’il choisit d’explorer de nouvelles actions ou d’exploiter celles déjà apprises. Si l’agent opte pour l'exploitation, il commence par convertir l’état du plateau en son état canonique. Ensuite, il consulte la Q-table pour récupérer la valeur Q maximale associée à cet état canonique, et choisit l’action correspondante. Il joue alors cette action.

Gains d'efficacité

Cette méthode de normalisation des états a permis de réduire la taille de la Q-table, rendant l'apprentissage plus rapide et efficace. Plutôt que de devoir traiter un vaste ensemble de combinaisons redondantes, l'agent peut se concentrer sur un nombre réduit d'états uniques, améliorant ainsi sa capacité à apprendre et à prendre des décisions optimales. En réduisant l’espace

des états, nous avons non seulement optimisé l'entraînement de l'agent, mais surtout réduit la taille mémoire de la Q_table.

Cependant

Cependant, malgré les efforts déployés et la réduction théorique du nombre d'états grâce à l'introduction de l'état canonique, cette approche s'est révélée moins efficace en pratique que ce que nous espérions. L'optimisation théorique du nombre d'états ne s'est pas traduite par une amélioration significative de la vitesse d'apprentissage de l'agent. En réalité, le processus de conversion entre un état initial et son état canonique, puis l'inverse, a introduit des coûts computationnels élevés qui ont ralenti l'entraînement de manière significative.

L'une des principales raisons de ce ralentissement est liée aux opérations nécessaires pour transformer un état du plateau. Passer d'un état initial à son état canonique implique d'appliquer plusieurs transformations : rotations et symétries. Bien que ces transformations aient été implémentées en utilisant la bibliothèque NumPy, réputée pour ses performances optimisées en algèbre linéaire, elles restent des opérations relativement lourdes en termes de calcul. Chaque coup joué nécessite de réaliser jusqu'à 16 transformations (8 pour les rotations et les symétries, et 8 pour le chemin inverse), car il faut non seulement convertir l'état initial en son état canonique pour identifier la stratégie optimale, mais aussi reconvertir cet état canonique en son état initial pour déterminer la direction exacte de la flèche, c'est-à-dire l'action à réaliser concrètement sur le plateau.

Bien que la réduction théorique du nombre d'états uniques grâce à l'état canonique soit indéniablement un atout, ce gain est malheureusement annulé par le temps de calcul excessif nécessaire pour gérer ces transformations. L'agent passe un temps considérable à transformer les états au lieu de se concentrer sur l'apprentissage stratégique. Le temps requis pour traiter chaque coup devient ainsi un obstacle majeur à l'efficacité de l'algorithme, ce qui ralentit considérablement le processus d'entraînement global.

Bilan du Q-Learning

Malgré les multiples approches testées pour entraîner un agent avec le Q-learning, les résultats sont restés décevants. Les agents peinent à être performants, même contre des stratégies humaines simples. Le principal obstacle réside dans un compromis complexe entre les coûts computationnels et les limites de mémoire.

L'utilisation de la représentation canonique, visant à réduire le nombre d'états uniques grâce aux symétries du jeu, a certes réduit la taille de la Q-table, mais au prix d'un lourd coût en calculs. Chaque coup nécessitait de nombreuses opérations de transformation (rotations, symétries) pour passer de l'état initial à son état canonique et inversement, ce qui a considérablement ralenti l'entraînement.

À l'inverse, sans cette optimisation, la Q-table devient très grande et rapidement ingérable, surchargeant la mémoire. Cette balance entre rapidité et gestion de l'espace d'état a rendu la méthode difficile à implémenter de manière efficace.

Malgré cela, beaucoup de choses furent appris, notamment sur le Q-learning et ses limites pour des jeux complexes. Il devient clair que des méthodes plus avancées devront être explorées pour surmonter les défis liés à la taille des états et à l'optimisation du temps d'entraînement.

Le Deep Q-Learning

Certains jeux présentent une complexité considérable en raison de la taille immense de leur espace d'états, de leur espace d'actions, ou parfois des deux. Un bon exemple se trouve dans les jeux Atari ou encore dans le célèbre jeu de Go. Pour les jeux Atari, la complexité provient du nombre de pixels à traiter à chaque image, car chaque état est représenté visuellement par une grille d'environ 210x160 pixels en couleurs (donc x3 encore pour le RGB). Quant au Go, la complexité vient de la taille de son plateau de 19x19 intersections, où le nombre d'états possibles dépasse de loin le nombre d'atomes dans l'univers. Ces jeux posaient un défi presque insurmontable pour les approches classiques d'apprentissage par renforcement, comme le Q-learning.

Cependant, ces défis ont été résolus grâce à des méthodes plus avancées basées sur l'apprentissage par renforcement, et plus particulièrement grâce à l'utilisation du Deep Q-Learning, une méthode qui combine le renforcement avec la puissance du Deep Learning.

Qu'est-ce que le Deep Learning ?

Le Deep Learning est une sous-discipline de l'intelligence artificielle qui repose sur des réseaux de neurones artificiels profonds. Contrairement aux algorithmes classiques, ces réseaux sont capables d'apprendre directement à partir de données brutes, en identifiant des motifs complexes dans des ensembles de données très volumineux. Ces réseaux sont composés de plusieurs couches de neurones, chacune transformant les données en représentations de plus en plus abstraites à mesure qu'elles se déplacent à travers le réseau. Ce processus permet de capturer des informations complexes et des relations cachées dans les données, comme des images, du texte, ou dans notre cas, les états d'un jeu.

Lorsque le Deep Learning est appliqué à l'apprentissage par renforcement, il permet à un agent d'apprendre à partir d'un espace d'états beaucoup plus vaste que ce qui serait possible avec des approches traditionnelles. Plutôt que de stocker chaque état et sa valeur correspondante (comme dans le Q-learning classique avec une Q-table), le Deep Q-Learning utilise un réseau de neurones pour approcher les Q-valeurs. Ce réseau est capable de généraliser les expériences précédentes de l'agent, ce qui signifie qu'il peut prendre de bonnes décisions même pour des états qu'il n'a jamais rencontrés auparavant.

L'algorithme de Deep Q-Learning (DQN) a révolutionné le domaine en résolvant des jeux complexes comme ceux d'Atari. Il fonctionne donc via cette idées majeures :L'approximation de fonction : Le réseau de neurones prend en entrée un état (par exemple, une image d'un jeu Atari) et sort des Q-valeurs pour chaque action possible. Cela permet de gérer des espaces d'états très grands sans avoir besoin de les parcourir entièrement. De plus il suffit alors de stocké les poids du modèle en non plus un dictionnaire de Q-value, ce qui résous la problématique d'espace mémoire.

Grâce à ces techniques, Deep Q-Learning permet à des agents d'apprendre à jouer à des jeux complexes en exploitant à la fois la puissance du deep learning pour représenter les états, et les principes de l’apprentissage par renforcement pour guider l’agent vers des stratégies optimales.

Optimisation du Deep Q-Learning : Méthodes Avancées

Suite à une recherche approfondie dans la littérature sur les méthodes d'optimisation du Deep Q-Learning (DQL), trois techniques supplémentaires ont été intégrées dès le début pour améliorer l'efficacité et la robustesse de l'apprentissage :

1.Replay d'expérience (Experience Replay)

Replay d'expérience est une technique qui permet de maximiser l'efficacité de l'apprentissage en réutilisant les expériences passées de l'agent. Plutôt que d'apprendre directement à partir des transitions (état, action, récompense, nouvel état) en temps réel, cette méthode consiste à stocker un ensemble d’expériences dans une mémoire tampon appelée mémoire de replay. L'agent échantillonne ensuite aléatoirement des mini-batchs de cette mémoire pour entraîner le réseau de neurones. Cette approche a plusieurs avantages : elle réduit la corrélation entre les échantillons successifs, ce qui rend l'apprentissage plus stable, et permet à l'agent de tirer parti de ses expériences passées de manière plus efficace.

2.Cible Q Fixe (Fixed Q-Target)

La technique de cible Q fixe vise à stabiliser l'entraînement en utilisant des cibles de Q-valeurs fixes pendant un certain nombre d'itérations. En DQL, le réseau de neurones est généralement composé de deux parties : un réseau principal qui propose des Q-valeurs pour chaque action, et un réseau cible dont les poids sont copiés périodiquement du réseau principal. Pendant l'entraînement, les cibles de Q-valeurs pour la mise à jour de la fonction de perte sont obtenues à partir du réseau cible fixe, ce qui empêche le réseau principal de se propager trop rapidement et d'induire des oscillations ou des divergences dans l'apprentissage.

3.Double Deep Q-Learning

Double Deep Q-Learning est une technique qui aborde le problème de la surdévaluation des Q-valeurs. Dans les méthodes de Q-learning traditionnelles, il est fréquent que les Q-valeurs estimées soient biaisées à la hausse. Cela se produit parce que les actions sélectionnées pour la mise à jour des Q-valeurs sont également celles pour lesquelles les Q-valeurs sont évaluées, créant un biais d'optimisme. Double Deep Q-Learning remédie à ce problème en séparant l'action choisie de l'évaluation de la Q-valeur. Concrètement, le réseau principal choisit l'action à prendre, mais la Q-valeur associée est évaluée par le réseau cible. Cette séparation permet d'obtenir des estimations de Q-valeurs plus précises et moins biaisées.

4.

L'intégration de ces trois techniques dans le cadre du Deep Q-Learning vise à améliorer la stabilité et la performance de l'apprentissage. Le Replay d'expérience maximise l'utilisation des données d'entraînement en permettant une révision répétée des expériences passées. La Cible Q Fixe stabilise le processus d'entraînement en utilisant des cibles fixes pour les mises à jour des Q-valeurs. Enfin, le Double Deep Q-Learning adresse le problème de surévaluation des Q-valeurs en

séparant les processus de sélection et d'évaluation des actions. Ces méthodes combinées permettent de créer des agents d'apprentissage par renforcement plus robustes et plus performants.

Mise en œuvre pratique

La mise en œuvre de ce projet a été réalisée par la création de deux classes principales : une classe pour le réseau de neurones et une autre pour la gestion de la mémoire.

Le premier réseau de neurones a été conçu comme une simple pile de couches linéaires, chacune suivie d'une fonction d'activation. Cette fonction ajoute une non-linéarité au réseau, évitant ainsi le problème de collapse des couches fully-connected, ce qui permet d'obtenir un réseau plus performant et complexe. Pour améliorer davantage les résultats, il est envisagé d'augmenter la taille et la profondeur du réseau, un paramètre qui sera testé ultérieurement.

L'utilisation d'une structure de type deque (double-ended queue) de taille fixe fut utilisé pour la mémoire, généralement de 10 000 éléments. Cela permet de conserver un historique d'expériences qui sera ensuite utilisé pour optimiser le réseau.

L'agent, quant à lui, est représenté par une nouvelle classe qui intègre à la fois le modèle de réseau de neurones et la mémoire. En plus des paramètres classiques comme l'epsilon, le gamma, etc., l'agent dispose d'une méthode optimize, qui effectue l'optimisation du réseau en sélectionnant aléatoirement des expériences dans la mémoire, en fonction d'un batch size prédéfini.

L'un des défis rencontrés a été la conversion du plateau de jeu 5x5, initialement constitué de chaînes de caractères, en un vecteur compatible avec l'entrée du réseau. De même, il a fallu convertir les actions, représentées par des triplets (i, j, flèche), en un indice numérique afin d'exploiter les sorties du réseau de manière efficace.

Le passage du plateau de jeu à un vecteur se fait à l'aide d'un encodage one-hot. Chaque case du plateau est représentée par un tableau de dimension 3, où un 1 apparaît à la position correspondant à la nature de la case. Par exemple, une case vide ("") sera encodée en [1,0,0], une case contenant un "X" en [0,1,0], et une case avec un "O" en [0,0,1]. Grâce à cet encodage, le plateau 5x5 composé de chaînes de caractères devient un tableau 5x5x3 d'entiers. Ensuite, en aplatissant ce tableau (avec la méthode .unsqueeze[0]), on obtient un vecteur de taille 75, prêt à être traité par le réseau de neurones. Ce processus d'encodage et de décodage de l'état est géré par les fonctions encode_state et decode_state.

Le traitement des actions a été un peu plus complexe. Il a fallu définir une fonction bijective capable de mapper chaque triplet (i, j, flèche) à un indice unique compris entre 0 et 43. Comme il y a un maximum de 44 actions possibles (3 actions pour chaque case, sauf aux coins où il n'y en a que deux), cela a nécessité deux étapes. Tout d'abord, une fonction, action_to_index, a été créée pour associer un indice unique à chaque triplet. Ensuite, pour réduire la plage d'indices entre 0 et 43, un dictionnaire a été utilisé pour établir une correspondance entre les indices non valides et les indices valides, assurant une complexité en temps constante (O(1)) lors de l'accès. Ces dictionnaires sont appelés faux_index_to_valid_index et valid_index_to_faux_index.

Une nouvelle fonction récompense :

La fonction récompense (nommé compute_reward$_3$, 3 car d'autre essaie de fonction reward fut testé avant), simple qu'alors, fut amélioré pour mieux représenté et juger l'état du plateau afin de donner plus d'indication à l'agent sur ce qu'est un bon ou un mauvais coup.

Voici les grandes étapes de son fonctionnement :

1. Extraction de l'état du plateau : La fonction commence par extraire l'état actuel du plateau de jeu.

2. Analyse des séquences : Pour chaque joueur (le joueur actif et l'adversaire), la fonction identifie les séquences de symboles alignés (lignes, colonnes et diagonales) et les motifs spécifiques dans ces alignements. Les séquences plus longues sont favorisées, car elles se rapprochent d'une victoire.

3. Contrôle central : La fonction vérifie également quelles positions centrales du plateau sont contrôlées par chaque joueur. Le contrôle de ces positions est valorisé car elles sont souvent stratégiques.

4. Vérification de victoire : Si un joueur a une séquence de 5 symboles alignés, cela signifie qu'il a gagné (ou perdu si c'est l'adversaire), et la fonction renvoie une récompense maximale ou minimale.

5. Calcul du score : La fonction combine les informations des séquences, des motifs alignés et du contrôle central pour calculer un score pour le joueur et un score pour l'adversaire. Le score de l'adversaire est légèrement amplifié pour encourager le joueur à jouer de manière plus défensive.

6. Retour de la récompense : Enfin, la fonction renvoie la différence entre le score du joueur et celui de l'adversaire pour évaluer la situation actuelle.

Ainsi, toutes les pièces sont en place pour entraîner notre agent. Les différentes fonctions sont soigneusement organisées dans la fonction principale DQL_rival, permettant une bonne interaction entre les composants. Un booléen, TEST, fut égalment ajouté, il désactive certaines parties du code, comme l'optimisation et le stockage en mémoire, afin de tester les performances du modèle. Par ailleurs, une décroissance progressive de l'epsilon est implémentée pour favoriser une meilleure exploitation à mesure que l'entraînement progresse. Tout cela est réuni dans la fonction train, qui prend en charge l'entraînement sur un grand nombre de parties.

Le temps d'exécution d'un épisode, c'est-à-dire d'une partie, varie en fonction de la taille du réseau et des calculs de back propagation nécessaires. Cela peut aller de 10 itérations par seconde pour un petit réseau, jusqu'à 2-3 itérations par seconde pour les plus grands réseaux testés.

Résultats :

Pour les petits réseaux de neurones (composés de deux à trois couches), les résultats initiaux furent décevants. Bien que les agents parviennent parfois à prendre de bonnes décisions et réussissent à vaincre des adversaires utilisant des stratégies aléatoires, ils échouent généralement face à un joueur humain. Ce dernier peut facilement exploiter les faiblesses de l'agent, surtout lorsque l'agent ne réagit pas lorsque l'humain se contente de maximiser une ligne. Dans ces situations, l'agent ne parvient pas à anticiper les menaces adverses et se fait rapidement dominer.

Nouvelle tentative d'entrainement

Pour remédier à ce problème, une nouvelle classe d'agents, nommée BruteForceAgent, a été créée. L'objectif de cet agent est de préparer les agents généralistes à contrer des stratégies agressives et directes. Le BruteForceAgent fonctionne de manière simple : au début de chaque partie, il sélectionne un numéro entre 0 et 19, qui correspond à une ligne et à une direction spécifique sur le plateau. Ensuite, tout au long de la partie, il tente de maximiser cette ligne le plus rapidement possible. S'il ne parvient pas à poursuivre sa stratégie initiale, il se contente de jouer de manière aléatoire.

Correspondance numéro tiré, ligne rushé par le BruteForceAgent (par exemple, 0 signifie , que le BruteForceAgent va essayé remplir la colonne de gauche en poussant depuis le haut) (en bleu c'est le plateau de jeu 5x5 du Quixo)

	0 ↓	1 ↓	2 ↓	3 ↓	4 ↓	
10 →						15 ←
11 →						16 ←
12 →						17 ←
13 →						18 ←
14 →						19 ←
	5 ↑	6 ↑	7 ↑	8 ↑	9 ↑	

Le BruteForceAgent n'utilise pas de modèle d'apprentissage. Bien qu'une variable d'instance self.model lui ait été assignée pour assurer la compatibilité avec la fonction d'entraînement train_DQL, celle-ci ne sert qu'à éviter de trop modifier la structure de la fonction. Sa logique reste purement déterministe : remplir la ligne désignée par son numéro de départ.

L'entraînement des agents généralistes contre ce type d'adversaire permet de les préparer aux stratégies directes souvent utilisées par des joueurs humains, qui cherchent à compléter rapidement une ligne. Il est essentiel de les entraîner sur de nombreuses parties pour qu'ils apprennent à reconnaître et contrer ces stratégies de manière plus efficace.

Cependant, même avec cette préparation, les résultats des petits réseaux restent limités lorsqu'ils sont opposés à des joueurs humains. Les agents ne réagissent toujours pas suffisamment aux actions

complexes et coordonnées de leurs adversaires. Il est donc nécessaire de tester des réseaux de neurones plus grands et plus profonds pour espérer améliorer leur performance face à des stratégies humaines plus sophistiquées.

Un réseau plus grand fut alors tester. Ce réseau comprend 6 couche dense relié. Des couches de drop_out ont été ajouté pour améliorer (rendre le réseau moins dépendant de l'input) et stabilisé l'entrainement. De plus au vu de la taille du réseau, des connexion résiduel furent ajoutées pour évité le gradient vanishing, très présent dans les réseaux avec de nombreuses couches (bien que ici, étant donné que les fonction d'action choisit sont la fonction ReLU (donc de dérivé nul ou égal à 1) et que le réseau ne fait que 6 couches, cela peux être trop mais c'est pour amélioré un maximum l'entrainement, afin d'avoir un agent performant).

Cependant, après plus de 8 heure d'entrainement, encore une fois ce ne fut pas assez performant. L'agent contrer parfois le BruteForce d'un humain mais parfois non, malgré l'entrainement spécialisé pour cela. L'agent faisait trop d'erreur et perdait à chaque fois en un temps plus ou moins long.

Monte Carlo Tree Search (MCTS) et Deep Learning pour une meilleure prise de décision

L'algorithme Monte Carlo Tree Search (MCTS) est une méthode d'exploration utilisée dans les jeux de stratégie comme le Go ou les échecs. Sa puissance réside dans sa capacité à explorer de manière aléatoire les différentes possibilités de jeu, à construire un arbre de décisions, et à affiner progressivement ses choix en fonction des résultats obtenus par des simulations. Pour un jeu comme Quixo, MCTS pourrait s'avérer particulièrement efficace en permettant à l'agent de simuler plusieurs parties et de s'adapter dynamiquement aux évolutions du jeu.

Lorsqu'il est couplé au deep learning, MCTS devient encore plus performant. En associant l'estimation de la valeur Q à un réseau de neurones, il est possible de simuler plusieurs états de jeu tout en effectuant des prédictions plus précises sur les meilleures actions à prendre. Pendant la simulation, le réseau de deep learning permet de réduire considérablement l'espace de recherche en explorant les possibilités les plus prometteuses. Ainsi, l'agent peut non seulement tirer parti de la puissance de calcul d'un ordinateur, mais aussi utiliser la capacité d'apprentissage et de réflexion d'un réseau de neurones pour prendre des décisions plus éclairées.

Fonctionnement du MCTS

Le processus MCTS est structuré en quatre étapes principales (compris dans la fonction select_action de la class agent MCTS_DQL) :

1. Sélection : À partir de l'état initial, l'algorithme descend dans l'arbre de recherche en choisissant à chaque étape le nœud qui maximise une métrique appelée UCT (Upper Confidence Bound for Trees) (via la fonction uct). Ce critère équilibre l'exploration de

nouvelles possibilités et l'exploitation des nœuds déjà jugés prometteurs. (fonction mcts_select)

2.Expansion : Une fois un nœud non exploré atteint, l'algorithme génère tous les enfants possibles à partir de cet état, ce qui représente les actions que l'agent peut entreprendre. (via la fonction mcts_expand)

3.Simulation : Le réseau de neurones intervient pour estimer la récompense potentielle de chaque nouvel état. En encodant l'état du jeu et en le passant à travers le modèle, l'algorithme obtient une prédiction qui guide les décisions. Via la fonction mcts_simulate)

4.Rétropropagation : La récompense obtenue lors de la simulation est ensuite propagée en remontant dans l'arbre de décision. Cela permet de mettre à jour les statistiques des nœuds visités et d'affiner les choix futurs. (via la fonction mcts_backprop)

Le MCTS répète ce processus plusieurs fois afin de s'assurer que l'agent choisit la meilleure action possible. L'algorithme se termine après un nombre défini de simulations, et l'action choisie est celle qui a été la plus explorée.

Pourquoi combiner MCTS et Deep Learning ?

Le couplage de MCTS avec un réseau de neurones permet une exploration plus efficace de l'espace des actions. Le deep learning apporte une puissance de calcul essentielle pour estimer les états de jeu et simplifie la prise de décision en anticipant les résultats des simulations. En résumé, ce duo offre à l'agent une capacité d'adaptation et d'optimisation accrue, maximisant ses chances de succès dans des environnements complexes comme les jeux de stratégie.

Les résultats sont meilleurs mais cela reste moins performant qu’un humain, cependant il faut noter que l’agent arrive parfois jusqu’a l’égalité. De plus on note une réel amélioration entre la méthode MCTS seul, avec une simulation random et la méthode couplé avec une simulation qui utilise de DQL. Il faut peut être continuer d’entrainer l’agent et le réseau de DQL car avec cette méthode, le nombre de partie joué à la seconde est bien plus faible, donc le réseau s’entraine moins (mais mieux en théorie).

Après de nombreuses heures d'entraînement, le réseau parvient enfin à surpasser un humain, ou tout au moins à garantir un match nul dans le pire des cas. Ce résultat est obtenu après un grand temps d’entrainement (plusieurs heures), et il est probable qu'en poursuivant l'entraînement, le modèle pourra réduire ce nombre, devenant ainsi plus performant en un laps de temps plus court.

Néanmoins, le temps de réflexion pour chaque coup reste inévitablement plus long qu'en l'absence de la méthode MCTS (on peut alors,retirer le délai de 500 ms). Cela est dû au nombre de simulation élevé nécessaire par coup Pour un jeu complexe comme le Go, la machine prenait environ deux minutes pour jouer un coup (source : documentaire sur AlphaGo, disponible sur YouTube), un délai plus long que celui observé ici, mais justifié par la complexité du jeu.

Possibles évolutions et améliorations

Plusieurs pistes d'amélioration et d'évolution pourraient être explorées dans l'avenir afin de rendre l'agent plus performant et mieux adapté aux complexités stratégiques du jeu Quixo.

L'une des premières pistes pourrait être l'intégration de réseaux neuronaux convolutifs (CNN). Ces réseaux sont généralement utilisés pour traiter des données visuelles, car ils excellent à capturer les relations locales dans des images, comme des motifs ou des contours spécifiques. Cependant, cette approche n'était peut-être pas la plus adaptée au jeu Quixo, en raison de la taille réduite du plateau (5x5) et de la nature de la stratégie à adopter. Les réseaux convolutifs se concentrent principalement sur des détails locaux, tandis que dans Quixo, ce sont les stratégies globales et les interactions à l'échelle du plateau entier qui sont cruciales pour remporter la victoire. Par conséquent, l'intérêt de détecter des motifs locaux au détriment d'une vue d'ensemble du plateau n'était pas pertinent dans ce contexte, ce qui a motivé la décision de ne pas privilégier cette approche pour le moment.

Ces pistes de développement montrent qu'il existe de nombreuses façons de continuer à améliorer l'agent et d'explorer des approches d'IA plus sophistiquées, selon les besoins spécifiques du jeu et les objectifs de performance que nous souhaitons atteindre.