RENDU MASTERIAL

- Apprenons à jouer à Pacman -

M2 - SA

Équipe :

- FOURNIER Jean-Charles
- BOUFALA Yacine
- IDOUMOU Zein
- GHOMARI Ryan

Responsable :

- LEDOUX Franck

Sommaire :

1. Introduction

L'objectif de ce projet est de développer une IA pour jouer au jeu Pacman en utilisant des techniques d'apprentissage par renforcement et de comparer les différents algorithmes entre eux. Nous utiliserons un projet de l'université de Berkeley[1] pour l'environnement du jeu.
Nous commencerons par expliquer ce qu'est l'apprentissage par renforcement et présenter l'algorithme Q-Learning qui est servira de base à nos explications.
Nous expliquerons ensuite les algorithmes issus d'un premier article de recherche[2] basé sur le même environnement et qui explore les algorithmes SARSA, Deep Q-Learning et un Approximate Q-Learning, que nous simplifierons par la suite par ALP (Approximate Learning par Poids). Puis nous explorerons l'algorithme SARSA(λ), abrégé ALV (Approximate Learning par Vecteur) d'un autre article[3] qui utilise cette fois un environnement différent.
Nous testerons par la suite chaque algorithme sur des terrains différents et l'analyse des performances de chacun se fera en comparant les pourcentages de victoires sur 100 parties.
Nous conclurons enfin par une comparaison des algorithmes testés et des pistes qui pourraient être explorées pour aller plus loin dans ce projet.

[1] http://ai.berkeley.edu/reinforcement.html
[2] Article 1
[3] Article 2

2. L’apprentissage par renforcement

L'apprentissage par renforcement est une méthode d'apprentissage automatique dans laquelle un agent doit apprendre à prendre des décisions en interagissant avec un environnement. Le but de l'agent est de maximiser une récompense numérique qui lui est donnée en fonction de ses actions. Les algorithmes étudiés reposent sur un processus de décision Markovien de par leur politique (que nous expliquerons plus loin) et le comportement aléatoire des fantômes de notre environnement.

A. Markov Decision Process (MDP)

Un processus de décision markovien est un modèle où un agent doit prendre des décisions dans un environnement et où le résultat de ses actions est aléatoire.

Il se définit par un quadruplet {S, A, T, R} où :

- S est l’ensemble des états possibles de l’environnement
- A est l’ensemble des actions possibles pour les états
- T est une fonction de transition qui associe une probabilité de se retrouver dans l’état s’ si on fait l’action a dans l’état s
- R est une fonction de récompense qui associe une valeur (positive ou négative) après chaque action de notre agent dans un état s

L’idée est de fixer des récompenses de manière à pousser l’agent à faire des actions « positives ». Dans le cas de notre projet où l’environnement est un jeu Pacman, notre MDP est défini comme cela :

- S est toutes les combinaisons des positions possibles de fantôme, de nourriture et de notre agent.
- A est {haut, bas, gauche, droite, Stop (ne pas bouger)} pour tout S
- T dépend de la valeur epsilon qu’on lui donnera (0.05 dans notre projet) et pour laquelle l’agent aura une chance de choisir une action aléatoire. Il dépend également du mouvement des fantômes.
- R vaut :
    - -1 à chaque frame où la partie n’est pas gagnée
    - 10 si l’agent mange une gomme
    - 200 si l’agent mange un fantôme
    - -500 si la partie est perdue
    - +500 si la partie est gagnée

B. Q-Learning :

Le Q-Learning est un algorithme d'apprentissage par renforcement classique. Bien qu'il ne soit pas présent dans les articles en tant que tel, il est important de le présenter pour bien comprendre les autres algorithmes.

Cet algorithme repose sur un tableau qui associe chaque paire état-action possible dans l'environnement à une valeur de récompense. On appellera le tableau la Q-table de l'agent, et la valeur estimée la Q-value de la paire état-action. La Q-fonction associe chaque état à l'action préférable.
Initialisée vide, l'agent va explorer son environnement et mettre à jour sa Q-table en utilisant la formule suivante :

$$\mathbf{Q(s,a) = Q(s,a) + \alpha\ (r + \gamma * max(Q(s',a')) - Q(s,a))}$$

Où :

- s est l'état actuel
- a est l'action courante
- s' est l'état suivant
- a' est l'action suivante
- r est la récompense reçue pour l'action a dans l'état s
- γ est le facteur d'actualisation. Il détermine l'importance des récompenses futures. Si γ -> 0, l'agent sera focalisé sur l'instant présent. Si γ -> 1, l'agent sera focalisé sur les récompenses futures.
- α est le taux d'apprentissage. Il détermine la vitesse à laquelle l'agent apprend. Entre 0 et 1, + cette valeur est grande, + l'agent prendra en compte les nouvelles expériences. À 0 ou 1, l'agent n'apprendra pas.
- **max(Q(s', a'))** correspond à la Q-value maximum parmi celles obtenables à l'état s'.

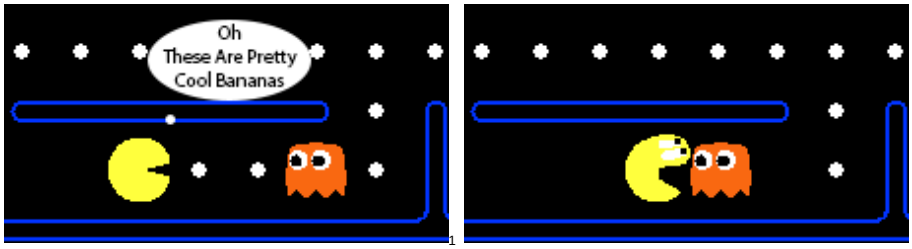
Pour apprendre, l'agent va enchaîner les parties pour avoir la Q-table la plus représentative de son environnement, il s'agit de la phase d'exploration. À chaque état, on lui demandera l'action qu'il souhaite faire.
L'agent suivra alors une politique, qui est la stratégie qu'il va utiliser pour maximiser ses récompenses. Nous étudierons ici uniquement la politique ε-greedy qui consiste à avoir un pourcentage de chance ε de choisir une action aléatoire (potentiellement celle qu'il aurait choisi) pour l'agent. Cela permet de ne pas limiter l'agent à une partie de l'environnement en le forçant à explorer des choix qu'il ne prendrait pas.
Lorsque que son apprentissage est terminé, l'agent passera en phase d'exploitation. Il n'apprendra plus, donc ε et α vaudront 0 (dans notre cas).

3. Les algorithmes des articles :

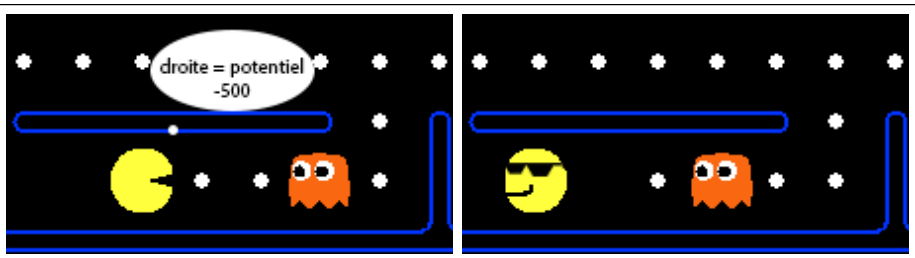
A. SARSA

SARSA (State-Action-Reward-State-Action) est une variante du Q-Learning. Il est très proche de celui-ci et diffère simplement dans la façon dont il choisira l'action à prendre en compte pour mettre à jour sa Q-table. Là où le Q-Learning utilise la valeur maximale qu'il pourra obtenir au prochain état (off-policy), SARSA va utiliser la valeur qu'il prendra avec sa politique (on-policy). Dans le cas de ε-greedy, il aura donc une probabilité ε de prendre la Q-value d'une action aléatoire et 1 - ε de prendre la Q-value max pour s'.

Cette différence amène un agent entraîné avec SARSA à éviter des actions qui pourraient le mener à un potentiel danger. Il sera donc meilleur dans des situations où l'environnement évolue de manière aléatoire. Il convient de mentionner que l'algorithme SARSA peut être moins robuste que le Q-Learning, en particulier dans les environnements déterministes où la prochaine action est toujours la même. Dans de tels cas, le Q-Learning convergera vers la politique optimale, tandis que SARSA peut converger vers une politique sous-optimale. On note également que SARSA converge moins rapidement vers une solution optimale que le Q-Learning.



Q-Learning (r + r' = 10 + -500)



SARSA (r + r' = -1 + -1)

Ci-dessus une situation où l'agent choisit sa première action mais où sa deuxième est aléatoire. Q-Learning va chercher les points mais perds la partie sur l'action aléatoire, là où SARSA va « prendre de la marge » et continuer la partie.

1 référence

B. Les techniques d'approximation

I. L'objectif de ces méthodes

Le problème avec les deux premiers algorithmes est celui de la dimensionnalité qu'ils engendrent. Puisqu'ils doivent considérer chaque action pour chaque état, leur Q-table devient rapidement très grande. Il est nécessaire pour évaluer correctement les actions pour un état d'y passer à plusieurs reprises, et d'y tester plusieurs fois les différentes actions possibles, donc le temps pour entraîner un agent sur un environnement grand est énorme.

C'est pourquoi l'Approximate Learning est un type d'algorithme par renforcement où l'on va simplifier l'environnement en récupérant uniquement certains paramètres que l'on jugera importants et entraîner l'agent à partir de cette simplification. Peu importe la taille de l'environnement, l'agent sera limité à la combinaison de facteurs qu'on lui donnera, ce qui réduit énormément le nombre d'état différents.

Comme chaque nouvel état regroupe en fait des états très différents, il se peut qu'un de ces algorithmes n'atteigne jamais une fonction optimale pour le jeu. Le but est alors de trouver le meilleur compromis entre la simplification / le nombre d'états et la précision de la représentation de l'environnement.

Deux méthodes d'approximation learning seront traitées, la première présente dans le premier article et dans le projet de Berkeley, et la deuxième est celle utilisée dans l'article 2.

II. Approximate Learning par Poids (ALP)

Dans cette première méthode, les états sont représentés par un dictionnaire de features et utilise le fonctionnement du Q-Learning pour le mettre à jour. Notre Q-table devient ici un dictionnaire (w) de poids pour chacune des features.

La Q-fonction utilisée devient alors :

$$\boldsymbol{Q(s,a)} \;=\; \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \boldsymbol{f_i(s,a)} \;*\; \boldsymbol{w_i}$$

Le dictionnaire de poids est mis à jour de la même manière que la Q-value vue précédemment :

$$\boldsymbol{w_i \leftarrow w_i + \alpha * différence * f_i(s,a)}$$
$$\boldsymbol{différence = (r + \gamma * max_{a'} Q(s',a')) - Q(s,a)}$$

Une feature positive comme par exemple « de la nourriture est proche » aura un poids positif, là où « un fantôme est proche » sera grandement négatif.

Cet algorithme est présent dans le premier article et dans un des exercices proposés avec le projet Berkeley qui nous fournit l'environnement. Les features retenues sont différentes pour les deux.
Pour celui du projet Berkeley, les features sont :

- si un fantôme est à une case de Pacman, « il y a un fantôme proche »
- si ce n'est pas le cas, « tu peux manger »
- la distance de la nourriture la plus proche
- un biais de 1

Ces paramètres permettent à Pacman de "voir" juste autour de lui et le guider vers la nourriture si elle est éloignée. Il n'y a pas de nombre de feature minimal ou maximal, on multipliera simplement toute celle données par l'état avec les poids de notre dictionnaire de poids w.

Pour les features de l'article, ils ont retenu :

- la distance à la nourriture la plus proche
- le nombre de fantôme actif à 1 pas
- la distance jusqu'au fantôme actif le plus proche
- l'inverse de la distance jusqu'au fantôme actif le + proche (?)
- le nombre de fantôme effrayés à proximité (à 1 ou 2 cases)
- la distance minimale jusqu'à une capsule
- la distance minimale jusqu'à un fantôme effrayé
- une feature binaire qui vaut 1 si pacman peut manger tranquillement et 0 si il y a un fantôme proche

Ces features donnent une vision un peu plus précise à l'agent.

Nous avons uniquement implémenté les features proposées par le projet pour des raisons que nous évoquerons dans la partie « Résultats ».

III. Approximate Learning par vecteur (ALV)

Dans l'article 2, les auteurs mettent en place un vecteur de taille 10 pour estimer l'état. Ils utilisent avec ces nouveaux états l'algorithme SARSA. La Q-table est alors largement réduite puisqu'elle ne contient "que" $4 * 2^9 = 2048$ états.
Le vecteur proposé dans l'article :

- S1 à S4 : La présence ou non d'un mur de chaque côté de Pacman (0 ou 1).
- S5 : Une recommandation de direction basée sur : si un fantôme est proche (distance <8), si un fantôme MANGEABLE est proche (distance < 5), si c'est la direction avec la nourriture la plus proche.
- S6 à S9 : La présence ou non d'un fantôme proche (distance < 8), de chaque côté de Pacman (0 ou 1).
- S10 : Si Pacman est piégé par les fantômes (0 ou 1)

Notez que S5 n'est pas précisément expliqué et laisse place à l'interprétation.

Pour améliorer les résultats, une technique de traces d'éligibilité (eligibility traces[4]) est utilisée. (Algorithme détaillé dans l'annexe X)
L'objectif de cette technique est de garder en mémoire les paires état-action par lesquels l'agent est passé récemment pour leur attribuer une partie de la récompense actuelle. Plus une paire a été visitée récemment, plus sa Q-value sera modifiée par la récompense obtenue. Grâce à cette méthode, l'apprentissage se fait plus rapidement et plus précisément.

Il existe plusieurs implémentation différente de cette technique qui change le coefficient avec lequel la récompense se propage. L'article a retenu la méthode qui consiste à donner le coefficient 1 à l'état actuel et à réduire la trace de toutes les autres paires par $\boldsymbol{\alpha} * \boldsymbol{\gamma} * \mathbf{trace\_paire}(\mathbf{s}, \mathbf{a})$ (on rappelle que $\boldsymbol{\alpha}$, le taux d'apprentissage, vaut 0.2 et $\boldsymbol{\gamma}$, le taux d'actualisation, vaut 0.8). Il est également possible d'ajouter 1 au coefficient au lieu de le fixer à 1, si la même paire état-action est visitée plusieurs fois. La dernière possibilité évoquée est de multiplier la valeur de la trace actuelle par un petit coefficient qui dépend du facteur d'actualisation, et d'y ajouter 1.

Cette méthode est utilisée avec l'algorithme SARSA ce qui donne l'algorithme Sarsa(λ), donné en annexe (Annexe 1).

[4] Livre d'où est tirée la technique. Chapitre 7, page 167

C. Deep Q-Learning

Le Deep Q-Learning est une variante du Q-Learning qui remplace la Q-table et les Q-values par un réseau de neurones. L'idée de cette méthode est là aussi de réussir à fonctionner avec un environnement important, là où les méthodes de base ne peuvent plus fonctionner dans un temps correct. Le principe est de donner en entrée du réseau l'environnement de l'agent et qu'il donne en sortie l'action de notre agent.
Le réseau de neurones s'entraîne à partir des expériences de l'agent et de nombreuses techniques ont été développées pour augmenter l'efficacité de cet entraînement et de ces résultats. Une équipe de DeepMind a combiné beaucoup de ces techniques dans ce qu'ils ont appelé « L'algorithme Rainbow »[5]. Il avait pour but de jouer à des jeux ATARI et les résultats qu'ils ont obtenus dépassent de loin les techniques individuellement et les performances d'un humain moyen.

Deep Q-Learning par CNN

Dans l'article 1, les auteurs convertissent l'environnement en une image où chaque élément de l'environnement est un pixel d'une certaine couleur. Puis ils font passer cette image dans un réseau à convolution pour obtenir l'action que doit effectuer Pacman.

Cette technique est combinée avec le mécanisme de replay memory qui consiste à stocker les x dernières expériences et d'en tirer un petit batch aléatoire pour se les « remémorer » en le faisant passer dans le réseau de neurones.

En plus de cela, les auteurs ont implémenté deuxième réseau de neurones reprenant la même architecture et ayant pour but de générer les valeurs cibles du premier. Cela permet de réduire les oscillations et de rendre la méthode plus stable.

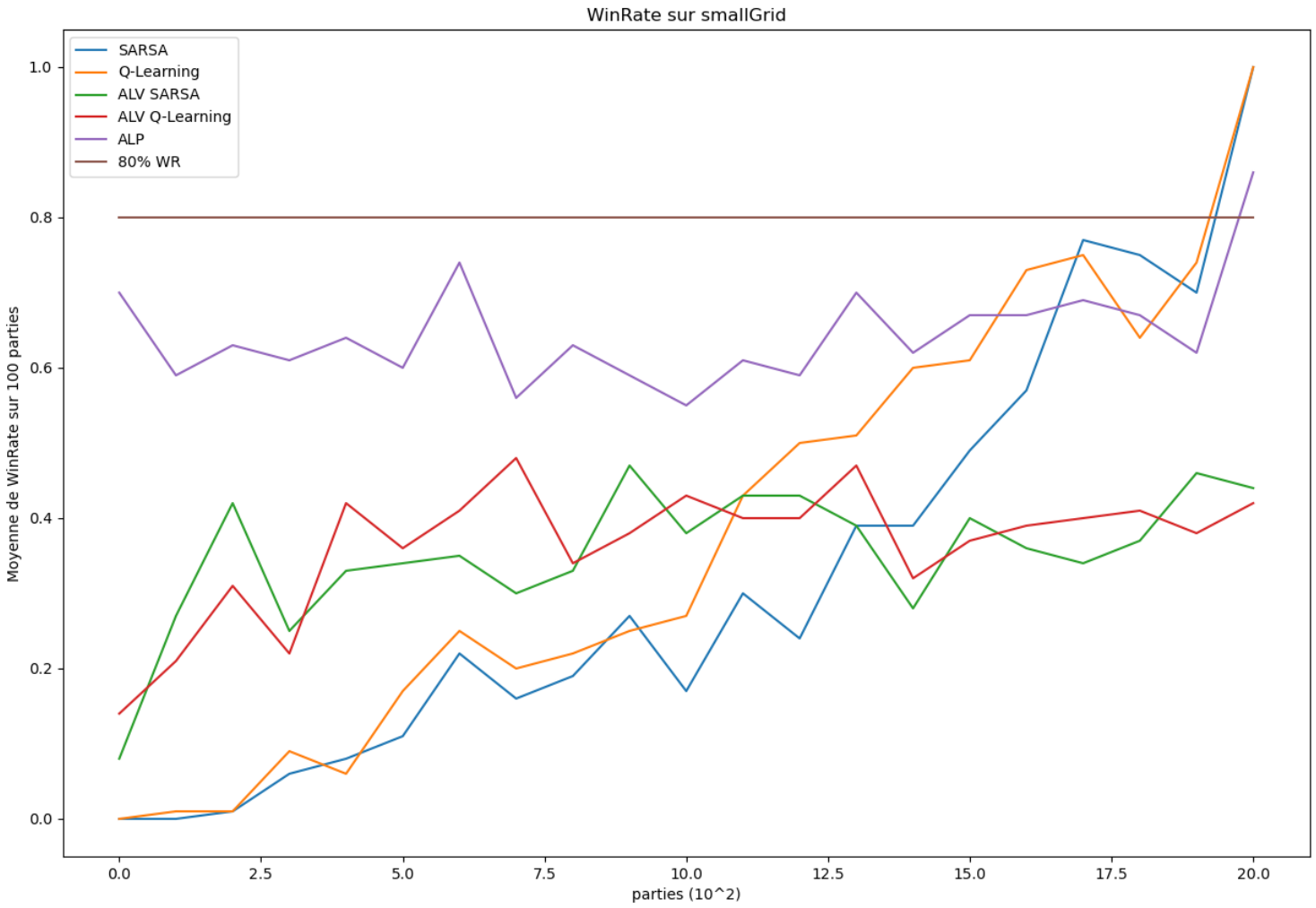
[5] L'article de recherche sur l'algorithme Rainbow

4. Résultats

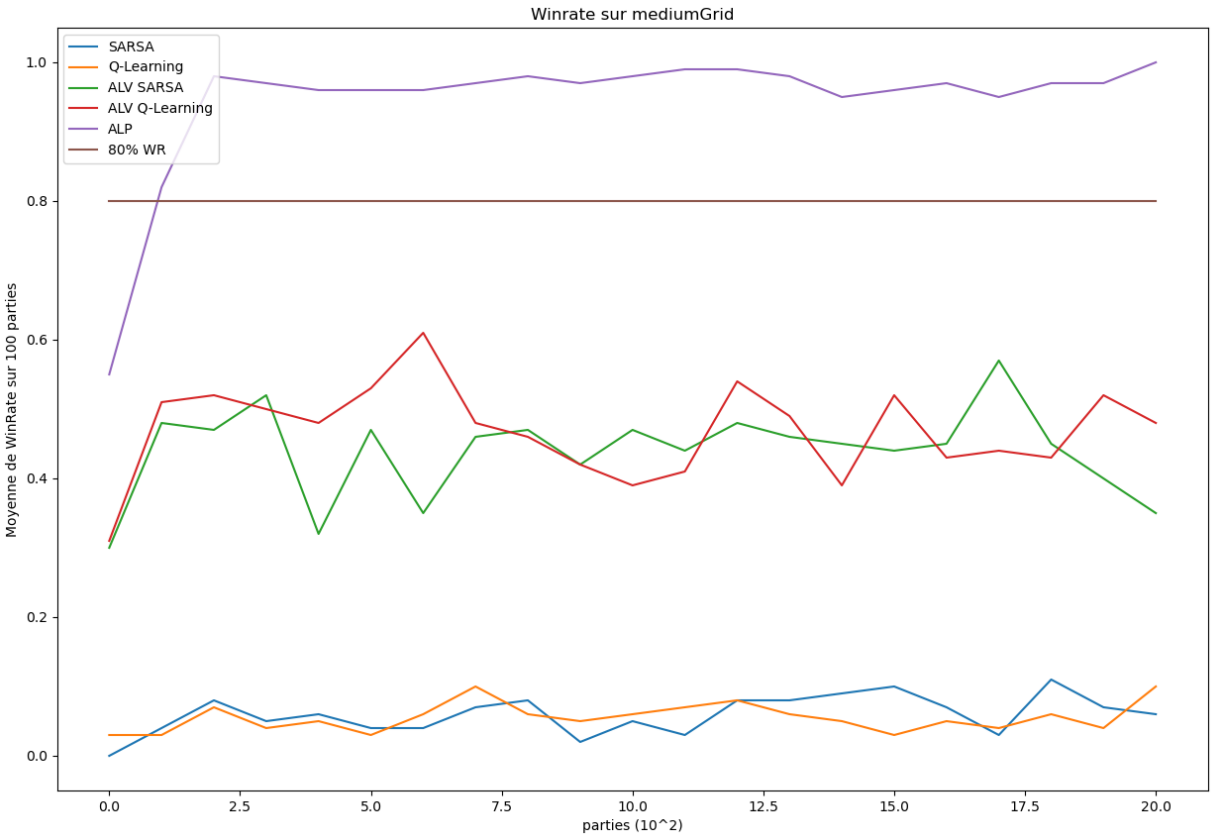
Nous avons implémenté les algorithmes Q-Learning, SARSA, ALP et ALV (SARSA(λ)). Nous avons utilisé 4 grilles de jeu pour nos tests, données en annexe (2 et 3). Pour évaluer les performances, nous nous fions au taux de victoire toutes les 100 parties.

Nous avons gardé les paramètres proposés par le projet (γ = 0.8 et α = 0.2) sauf pour l'ALV où nous avons utilisé ceux utilisé dans leurs expérimentations (γ = 0.99 et α = 0.01).

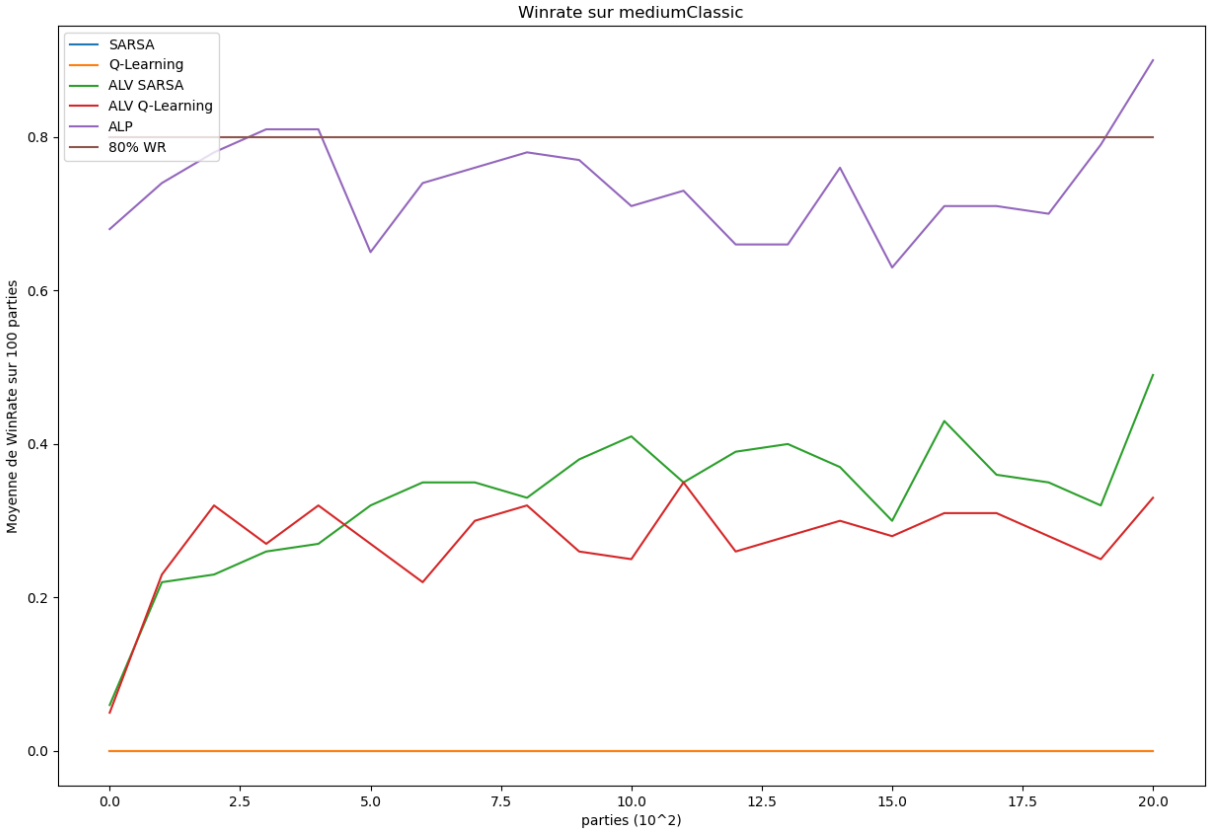
Voici les résultats obtenus pour smallGrid, mediumGrid et mediumClassic :



WR sur 100 parties d'exploitation :
SARSA & Q-L: 1 | ALV SARSA: 0.44 | ALV Q-Learning: 0.42 | ALP: 0.86



WR sur 100 parties d'exploitation :
SARSA: 0.07 | Q-Learning: 0.06 | ALV SARSA: 0.45 | ALV Q-Learning: 0.47
ALP: 0.97



WR sur 100 parties d'exploitation :
SARSA & Q-L: 0 | ALV SARSA: 0.49 | ALV Q-Learning: 0.33 | ALP: 0.9

Nous avons testés d'entraîner un agent avec le Q-Learning et SARSA sur mediumGrid sur 20 000 parties, puis évaluer sa phase d'exploitation sur 1000 parties et nous avons obtenu un taux de victoire de 0.67 pour SARSA et 0.92 pour le Q-Learning. Ce sont des bons scores, mais pour un nombre de parties 10 fois supérieur par rapport aux algorithmes approximatifs.

Nous avons inclus l'ALV avec la méthode Q-Learning dans nos résultats puisque même si SARSA est théoriquement meilleur pour les cas où les situations sont très aléatoires, nous voulions confirmer cela. Les différences entre ALV SARSA et ALV Q-Learning sont assez variables d'un environnement à un autre et il est intéressant d'expérimenter les deux pour prendre le meilleur, qui n'est pas toujours le même. Il est notable que la version Q-Learning s'exécute plus rapidement et donc gagne (ou perd) en moins d'actions (3 fois moins de temps sur mediumGrid). Il pourrait être intéressant d'étudier l'aspect du temps des algorithmes.

À noter que smallGrid et mediumGrid ne respectent pas une structure classique d'un niveau de Pacman. Les résultats sur ces environnements sont à titre indicatif pour le Q-Learning et SARSA, mediumClassic est le vrai test pour vérifier les performances qu'aurait notre agent dans le vrai jeu.

A. Q-Learning et SARSA

Nos résultats pour ces deux algorithmes mettent en évidence le problème de dimensionnalité dont ils souffrent. Bien qu'ils s'en sortent bien pour des grilles petites (mieux que l'ALV), non sans un temps d'entraînement long, ils sont incapables d'apprendre sur des environnements réalistes pour le jeu. Lors de nos essais sur la grille mediumClassic, aucun des deux n'a réussi à gagner la moindre partie.
On peut néanmoins souligner que les résultats pour les environnements possibles sont parfaits, ou tendent à l'être avec assez d'entraînement, puisqu'ils ont expérimenté tous les états possibles avec toutes les transitions possibles pour savoir quelle action est celle optimale.

B. ALP et ALV

Les résultats pour ces deux algorithmes sont satisfaisants mais nécessitent une analyse profonde.

Tout d'abord, l'ALP étudié propose les meilleurs résultats sur les cas étudiés, autant en termes de vitesse d'apprentissage que de performances, mais qui reste un cas d'expérimentation. En effet, les features données à l'agent lui permettent de fuir les fantômes quand il le peut et de se diriger rapidement vers la nourriture. Cependant, elles ne permettent pas d'éviter un cas où Pacman se retrouve piégé entre deux fantômes et son comportement ne ressemble pas à celui d'un vrai joueur, il pourrait même être approché avec des méthodes beaucoup plus simple.
Nous pouvons noter que nous ne disposons pas dans notre environnement de fantômes qui reproduisent le comportement réel des fantômes du jeu original. Chaque fantôme a en réalité un comportement distinct qui vise à piéger Pacman, l'agent entraîné avec cet algorithme aura sans doute beaucoup plus de difficulté pour gagner une partie.
Nous n'avons pas implémenté les features proposés dans l'article 1 puisque aucune d'entre elle ne peux aider notre agent à éviter d'être piégé, la seule réelle faiblesse actuelle, seulement se diriger vers des fantômes mangeables (ce qui ne change pas vraiment l'issu d'une partie) et un potentiel léger gain de vitesse dans l'apprentissage.

L'ALV implémenté nous offre des performances convenables et explore les traces d'éligibilité qui permettent un gain de performances et de vitesse d'apprentissage. Les résultats obtenus sur mediumClassic correspondent aux résultats obtenus par les chercheurs.
Malheureusement, le fait qu'il ait été fait pour un environnement différent nous pénalise dans notre utilisation de celui-ci et toute amélioration qu'on pourrait proposer ne serait pas forcément une amélioration pour l'environnement original.
On peut néanmoins relever que la distance et le fonctionnement des paramètres 5 à 9 qui étudie la menace des fantômes pour une distance de 8 ne permet pas correctement d'éviter d'être piégé. Une amélioration en prenant en compte un paramètre plus complexe, ou simplement des paramètres en plus pourrait permettre de palier à ce problème.

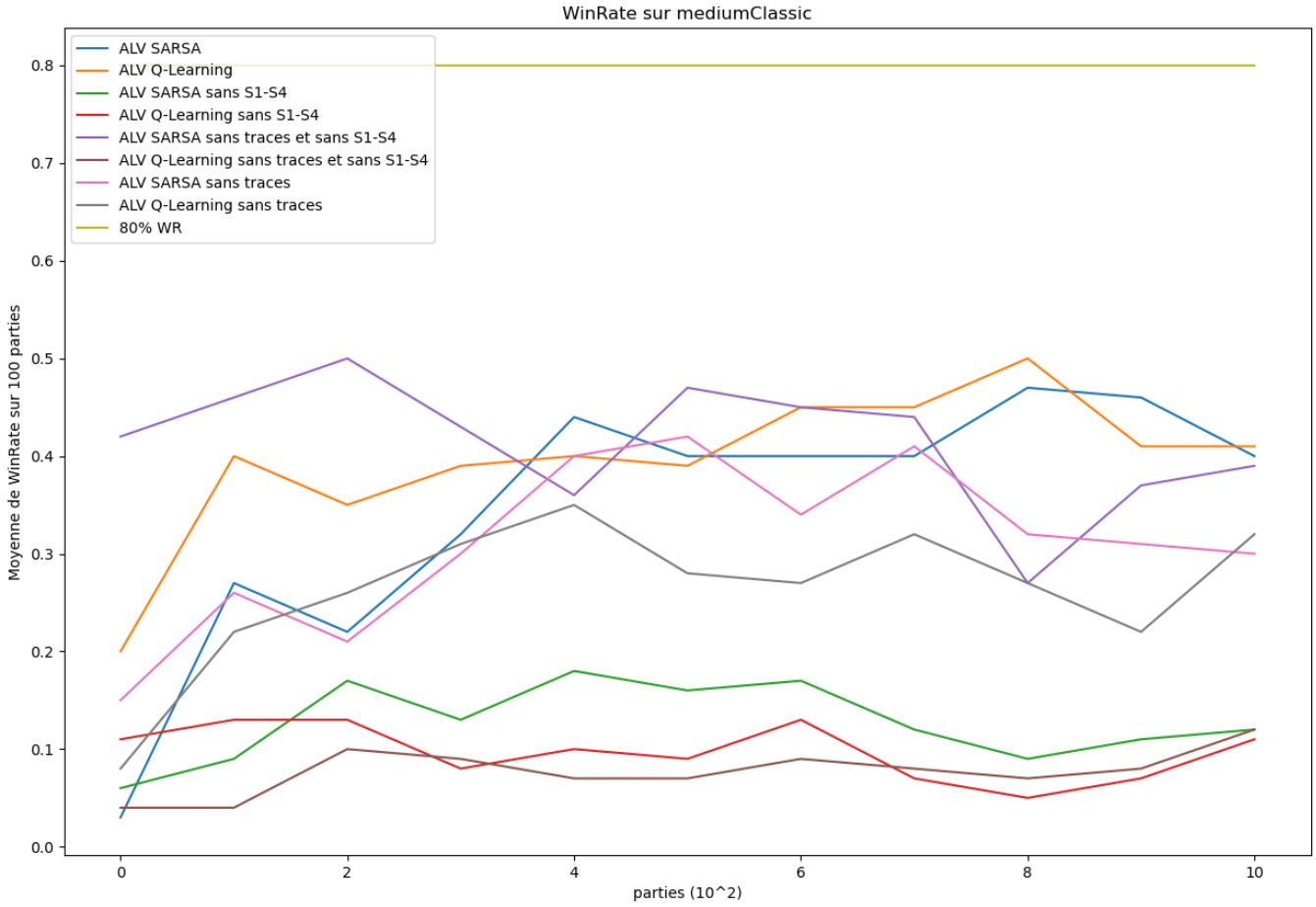
C. Expérimentations

Nous avons cherché à étudier les raisons des paramètres des algorithmes pour essayer de les améliorer. Nous nous sommes concentrés sur l'ALV puisque de nombreux paramètres et améliorations ont été utilisés sans qu'ils n'évoquent réellement la raison de leur choix. Nous avons remarqué que

modifier le taux d'apprentissage pour l'ALP pouvais améliorer les résultats mais nous n'avons pas creuser plus loin pour cet algorithme.

Nous avons effectué nos différents tests sur l'environnement mediumClassic sur 1000 parties d'entraînements et 100 parties de test.

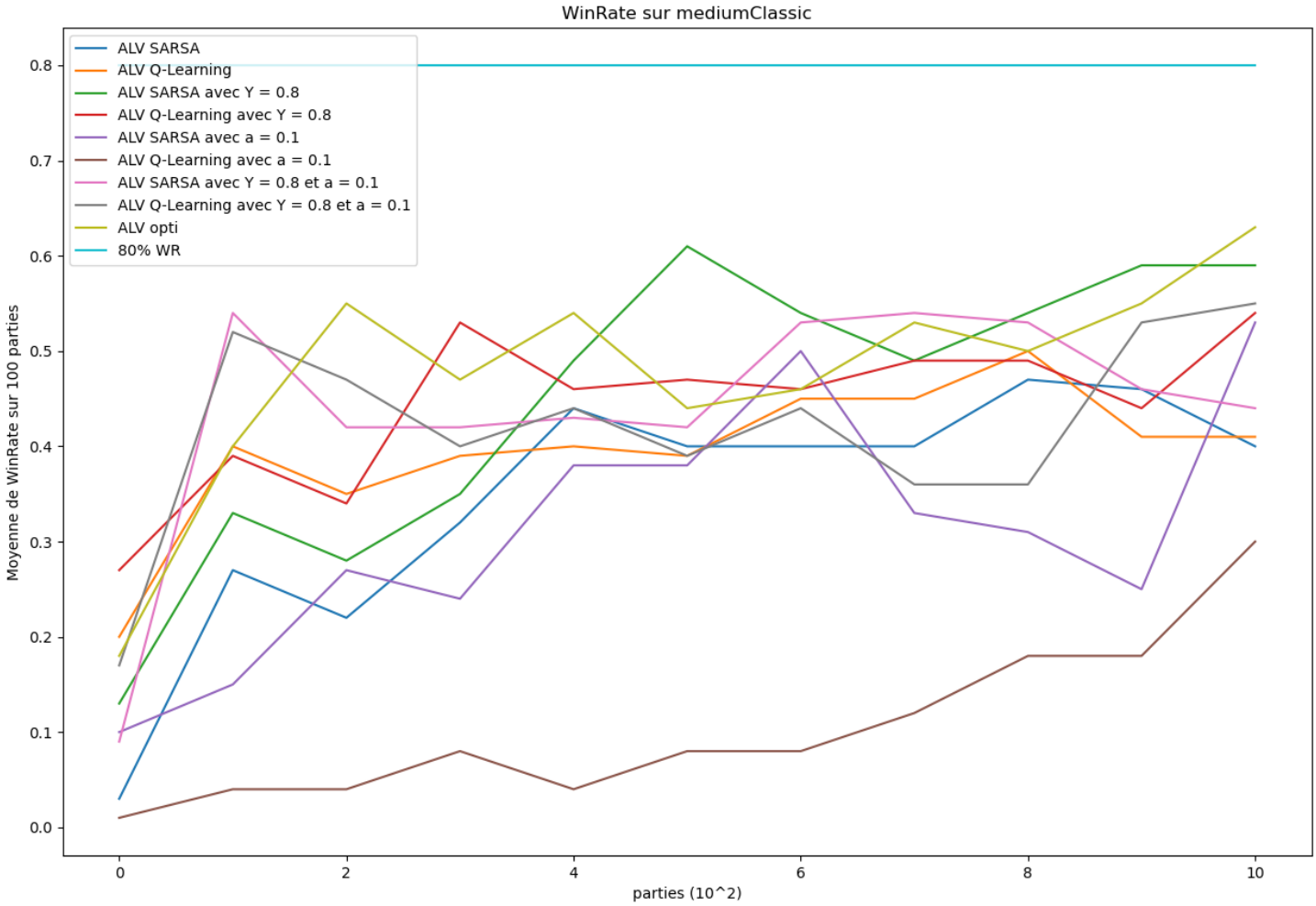
Nous avons d'abord cherché à savoir si les traces d'éligibilité utilisées apportaient réellement quelque chose à l'agent. Après cela, nous nous sommes intéressés aux quatre premiers éléments du vecteur, à savoir l'indication d'un mur sur les côtés de Pacman. Dans l'environnement que les auteurs ont utilisé, l'agent n'avait pas accès aux actions possibles comme dans le nôtre (ni à l'action Stop), c'est pourquoi ils ont donné une récompense négative (-100) au fait de foncer dans un mur. Notre agent étant par défaut orienté sur les actions possibles, nous avons voulu savoir si nous pouvions supprimer ces caractéristiques pour accélérer l'entraînement de celui-ci. Ci-dessous les différents résultats obtenus :



On peut observer qu'une version avec SARSA, sans les traces et sans les paramètres liés aux murs donne des résultats proches de ceux obtenus avec tous les paramètres de l'article, mais que les traces permettent bien d'obtenir

de meilleurs résultats et que les murs, bien que non pénalisant pour notre environnement, améliore les résultats de l'agent.

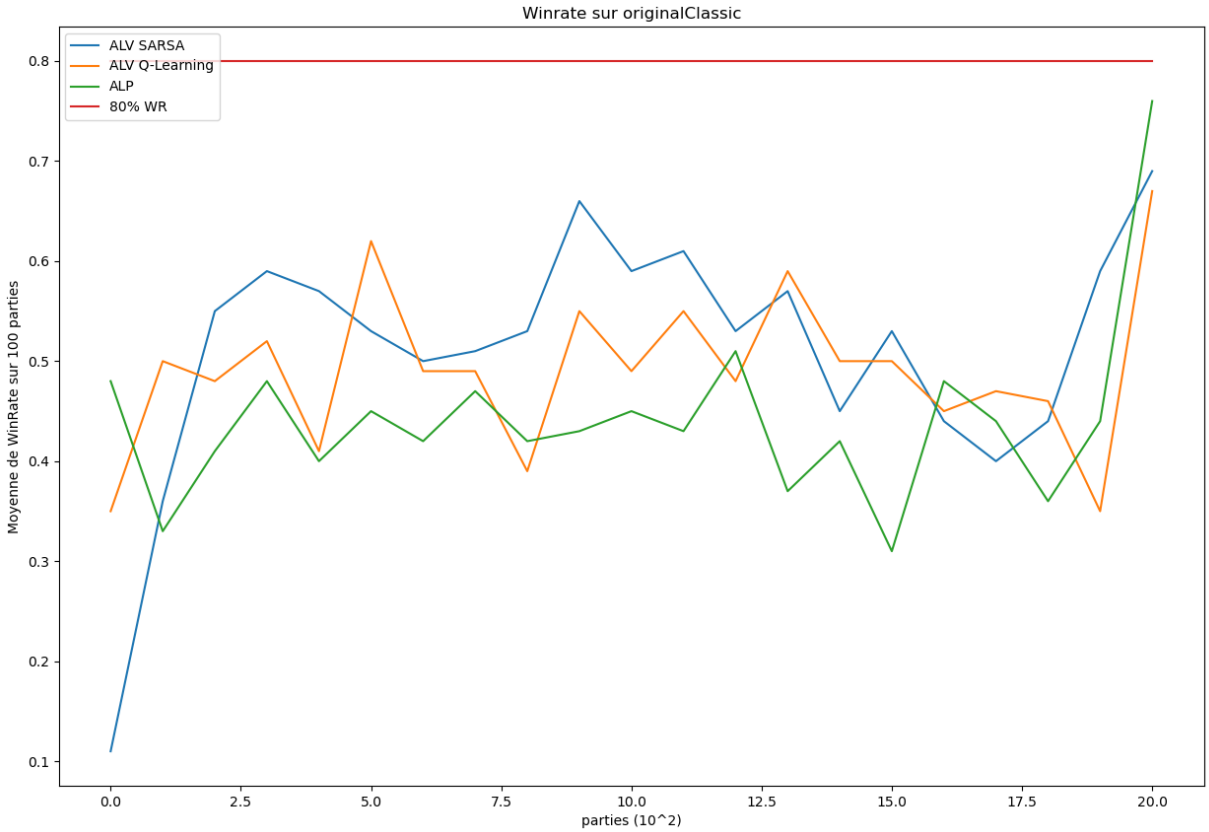
Après cela, nous nous sommes intéressés aux paramètres d'apprentissage qu'ils utilisaient (pour rappel γ = 0.99 et α = 0.01). Ces valeurs jouent un rôle dans les traces d'éligibilité et dans la mise à jour des Q-values.
Nous avons essayé de changer γ par 0.8 et α par 0.1, puis la combinaison des deux.
Nous avons après quelques expérimentations, des paramètres qui donnent un meilleur score et que nous appelerons « ALV optimal » (nous n'avons pas testé toute les combinaisons. Cette configuration utilise γ = 0.8 et α = 0.05.



Notre ALV optimal obtient 63 victoires sur 100 parties et un taux de complétion de 90% en moyenne, contre 40 victoires
Nous avons essayé notre ALV optimal sur une grille réaliste du jeu, ainsi qu'avec l'option « Directional Ghosts » inclus dans notre environnement. Pour ALV et ALP, les résultats avec cette option sont mauvais, ce qui s'explique par le fait qu'aucun des algorithmes ne peut vraiment éviter les

situations de piège et au fait que les fantômes sont beaucoup plus « agressifs » que dans le vrai jeu.

Voici les résultats obtenus sur une grille semblable à une vraie (annexe 4) :



WR sur 100 parties d'exploitation :
ALV SARSA: 0.69 | ALV Q-Learning: 0.67 | ALP: 0.75

On peut voir que pour une grille se rapprochant d'une vraie, les deux types d'algorithme ont des résultats très bons et assez proches.

D. Deep Q-Learning

Nous n'avons pas eu le temps de terminer l'implémentation de cette méthode. Nous pouvons néanmoins comparer les valeurs obtenues dans l'article celles que nous avons pour les techniques implémentées.

En 2000 parties, leur algorithme atteint entre 95% et 100% de taux de victoire sur smallGrid et mediumGrid ce qui est supérieur à ce que les autres ont obtenu. Aucun résultat précis n'est donné pour l'environnement mediumClassic mais ils estiment qu'il faudrait 1 million de parties avec leur méthode pour obtenir des résultats corrects.

5. Conclusion

Nous avons testé différents algorithmes d'apprentissage par renforcement et avons obtenus des premiers résultats intéressants avec certains d'entre eux. Tout d'abord, avec le Q-Learning et SARSA, nous avons pu voir le problème de dimensionnalité qu'ils provoquent.
Nous avons ensuite vu deux méthodes d'Approximate Learning qui permettent de palier ce problème en synthétisant l'environnement en peu d'éléments. Ils sacrifient cependant la vision globale de celui-ci en ne donnant que les informations proches de l'agent et cause pour cela une impossibilité d'éviter des situations qui font obligatoirement perdre l'agent. Ce type d'algorithme possède la contrainte de donner le maximum d'éléments à l'agent tout en restant le plus concis possible pour éviter de retomber dans des problèmes de dimensionnalités.
Nous n'avons pas eu le temps d'étudier les algorithmes utilisant des réseaux de neurones. Nous savons cependant qu'ils permettent une vision globale de l'environnement tout en évitant le problème de dimensionnalité.

Pour résumer algorithmes que nous avons étudiés :

	Points forts	Points faibles
Q-Learning	Exécution rapide, Q-fonction exacte	Ne fonctionne pas sur grand environnement
SARSA	Fonctionne mieux que le Q-Learning sur des environnements aléatoires.	Ne fonctionne pas sur grand environnement, lent
ALV	Très modulaire pour beaucoup d'approximation différentes	Q-fonction moins bien approximée
ALP	Tend rapidement vers une Q-fonction correcte, même pour des grands environnements	Est limité par le nombre de features qu'on peut donner
Deep Q-Learning	Vision globale de l'environnement, beaucoup d'améliorations possibles	Peu efficace sans ces améliorations

Beaucoup de points restent encore à étudier comme les paramètres d'apprentissage, les différentes fonctions de traces d'éligibilité et d'autres techniques qui font l'objet de recherche dans ce genre de problème. L'implémentation du Deep Q-Learning est également un point important que nous aurions aimé étudier puisqu'il s'agit d'une technique au centre de la recherche actuelle (Chat-GPT, Stockfish pour les échecs…)

Ce projet a été pour nous l'occasion de mettre en place des algorithmes d'apprentissage par renforcement qui proposent une approche différente du Machine Learning que nous avons étudié jusqu'alors, sans supervision. Nous avons également pu mettre en pratique des méthodes que nous avions étudié de manière théorique en début d'année en faisant un état de l'art sur l'apprentissage par renforcement appliqué à des problèmes d'énergie.

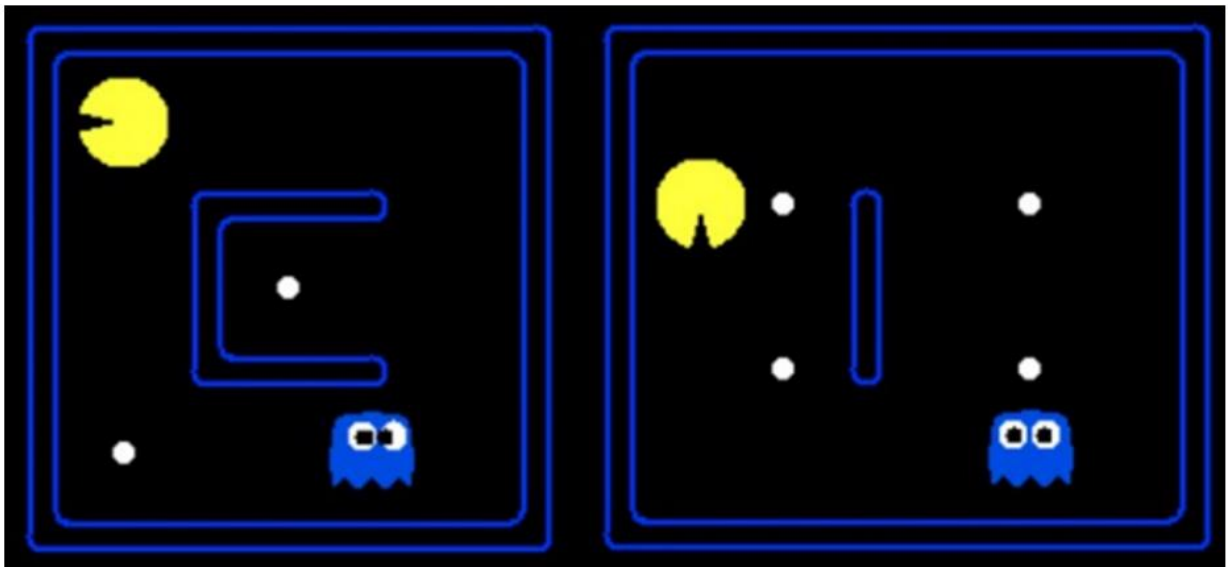
J'ai malheureusement manqué de temps pour exploiter au mieux ce projet mais si, à tout hasard, l'université avait une offre de thèse dans le domaine de l'apprentissage par renforcement à proposer, j'aurais tout le temps nécessaire pour explorer toutes les possibilités de ce genre de problème. :^)

ANNEXE :

1)

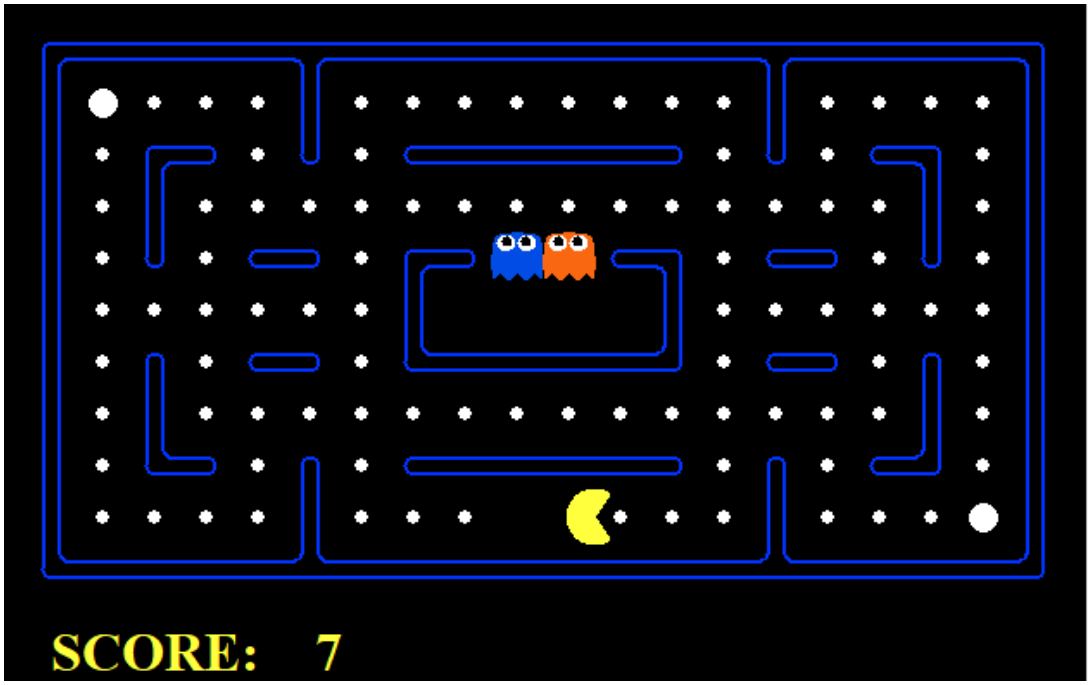
Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Repeat (for each episode):
  $E(s,a) = 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
  Initialize S, A
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $\delta \leftarrow R + \gamma Q(S', A') - Q(S, A)$
    $E(S, A) \leftarrow E(S, A) + 1$ (accumulating traces)
    or $E(S, A) \leftarrow (1 - \alpha)E(S, A) + 1$ (dutch traces)
    or $E(S, A) \leftarrow 1$ (replacing traces)
    For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
      $Q(s,a) \leftarrow Q(s,a) + \alpha \delta E(s,a)$
      $E(s,a) \leftarrow \gamma \lambda E(s,a)$
    $S \leftarrow S'$; $A \leftarrow A'$
  until S is terminal

2)



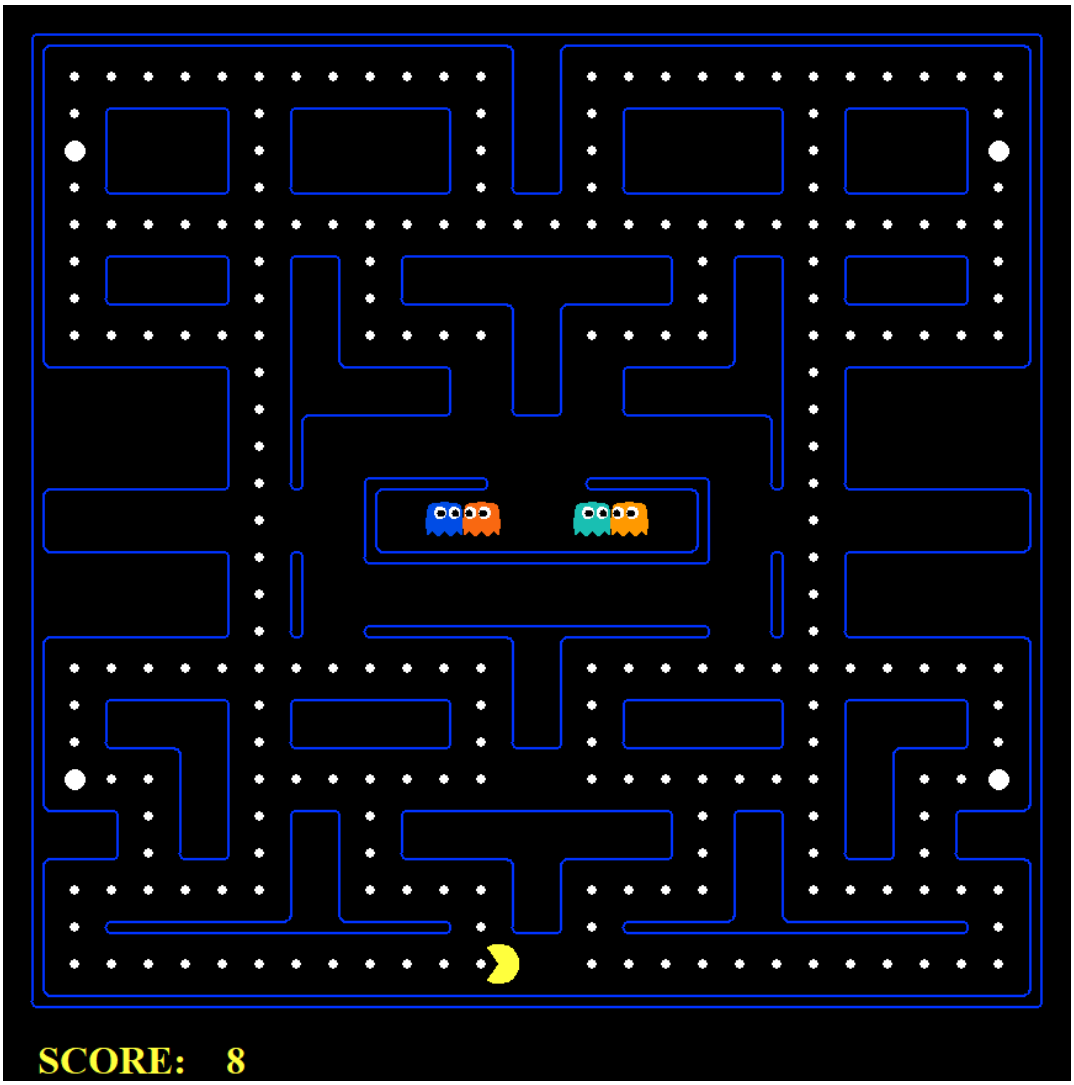
smallGrid à gauche et mediumGrid à droite

3)



mediumClassic

4)



originalClassic