Rapport Deep Reinforcement Learning

1 – Dynamic Programming

La programmation dynamique est une méthode utilisée pour résoudre des problèmes en décomposant un problème complexe en sous-problèmes plus simples. Dans le contexte de l'apprentissage par renforcement, la programmation dynamique est utilisée pour résoudre des problèmes de décision markoviens (MDP). Les trois méthodes principales de programmation dynamique, l'évaluation de politique, l'itération de politique et l'itération de valeur, sont toutes implémentées dans ce code.

Un **MDP** est défini par un **ensemble d'états**, un ensemble **d'actions**, une **fonction de transition de probabilité** et une **fonction de récompense**. Ils sont définis pour chaque environnement : Line World, Grid World et Secret Env1.

Le **Line World** est un environnement constitué de 7 états, allant de 0 à 6. Les états 0 et 6 sont des états terminaux avec des récompenses de -1 et 1 respectivement. Les actions possibles sont de se déplacer à gauche (0) ou à droite (1).

La fonction **p_line_world** est une implémentation de **la fonction de transition de probabilité** pour l'environnement "Line World". La fonction renvoie la probabilité que l'agent se retrouve dans l'état s’ et reçoive la récompense r après avoir pris l'action a dans l'état s.

Si l'état actuel est 0 ou 6 (les états terminaux), l'agent ne peut pas prendre d'action, donc la probabilité de transition est toujours 0.
Si l'agent choisit de se déplacer à droite et n'est pas dans l'état 5, il se retrouve dans l'état s+1 et reçoit une récompense de 1, donc la probabilité de transition est 1.
Si l'agent choisit de se déplacer à droite et est dans l'état 5, il se retrouve dans l'état s+1 (l'état terminal 6) et reçoit une récompense de 2, donc la probabilité de transition est 1.
Si l'agent choisit de se déplacer à gauche et n'est pas dans l'état 1, il se retrouve dans l'état s-1 et reçoit une récompense de 1, donc la probabilité de transition est 1.
Si l'agent choisit de se déplacer à gauche et est dans l'état 1, il se retrouve dans l'état s-1 (l'état terminal 0) et reçoit une récompense de 0, donc la probabilité de transition est 1.

Pour toutes les autres combinaisons d'états, d'actions et de récompenses, la probabilité de transition est 0, car elles ne correspondent pas aux règles de l'environnement.

La fonction **pi_random_line_world** établit une stratégie d'action pour l'agent qui est aléatoire de manière uniforme, l'agent ayant une probabilité égale de choisir de se déplacer à gauche ou à droite

La fonction **policy_evaluation** implémente l'algorithme d'évaluation de politique, qui est une méthode pour calculer la fonction de valeur d'une politique donnée pour un processus de décision de Markov (MDP).

L'algorithme d'évaluation de la politique fonctionne en itérant sur chaque état de l'environnement et en calculant la récompense attendue de cet état en tenant compte de la politique actuelle et des récompenses et transitions possibles pour chaque action.
La **même fonction** est utilisée pour le **Line World**, le **Grid World** et l'**environnement 1**.

Elle commence par initialiser une fonction de valeur V, qui est un dictionnaire où chaque état s du MDP a une valeur initiale de 0.

La fonction entre ensuite dans une boucle qui continue jusqu'à ce que la fonction de valeur V ait convergé. La convergence est déterminée par delta, qui mesure le changement maximal de la fonction de valeur d'une itération à l'autre. Si delta est inférieur à un petit nombre theta, on considère que V a convergé et la boucle s'arrête.

Pour chaque état s, la fonction calcule une nouvelle estimation de la valeur de cet état en tenant compte des probabilités de transition vers chaque état suivant s', des récompenses associées à ces transitions, et des valeurs actuelles de ces états suivants. Cette somme est pondérée par les probabilités des actions selon la politique actuelle.

Lorsque la fonction de valeur a convergé, la fonction retourne cette fonction de valeur.

La fonction **policy_evaluation_on_line_world** applique l'algorithme d'évaluation de politique à l'environnement Line World sous une politique aléatoire uniforme. L'objectif est de déterminer la fonction de valeur associée à cette politique spécifique dans cet environnement.

La fonction renvoie la fonction de valeur de cette politique, qui donne la récompense totale attendue pour chaque état lorsqu'on suit la politique donnée :

```
policy_evaluation_on_line_world :
{0: 0.0, 1: -0.666662417853974, 2: -0.33332807119284236, 3: -2.933639832725099e-07, 4: 0.33332755780513823, 5: 0.6666621122647801, 6: 0.0}
```

On peut voir que la valeur augmente à mesure que nous nous déplaçons de l'état 1 à l'état 5. Ce résultat était prévisible car il est plus bénéfique d'être proche de l'état avec la plus grande récompense (l'état 6 terminal dans notre cas).

La fonction **policy_iteration** implémente l'algorithme d'itération de politique afin de trouver la politique optimale dans le MDP en alternant entre l'évaluation de la politique et l'amélioration de la politique jusqu'à ce que la politique ne change plus.
L'évaluation de la politique calcule la fonction de valeur de la politique actuelle et l'amélioration de la politique met à jour la politique en choisissant, pour chaque état, l'action qui donne la plus grande valeur selon la fonction de valeur actuelle.
La **même fonction** est utilisée pour le **Line World**, le **Grid World** et l'**environnement 1**.

La fonction commence par initialiser une politique pi de manière aléatoire. Ensuite, elle utilise l'évaluation de la politique pour calculer la fonction de valeur V de cette politique initiale.

La fonction entre ensuite dans une boucle qui continue jusqu'à ce que la politique pi ne change plus d'une itération à l'autre et devienne stable.
Pour chaque état s dans l'espace des états S, la fonction calcule la valeur de toutes les actions a et met à jour la politique pour cet état en choisissant l'action qui donne la plus grande valeur.
Si la politique a changé pour l'état s, elle définit policy_stable comme False.
Si la politique est stable (i.e. policy_stable est à True), la boucle se termine et la fonction retourne la politique et la fonction de valeur. Sinon, elle met à jour la fonction de valeur V avec la nouvelle politique et recommence la boucle.

La fonction **policy_iteration_on_line_world** applique l'algorithme d'itération de politique à l'environnement Line World. L'objectif de cette fonction est de déterminer la politique optimale et la fonction de valeur associée pour cet environnement spécifique.

```
policy_iteration_on_line_world :
PolicyAndValueFunction(pi={0: {0: 1, 1: 0}, 1: {0: 0, 1: 1}, 2: {0: 0, 1: 1}, 3: {0: 0, 1: 1}, 4: {0: 0, 1: 1}, 5: {0: 0, 1: 1}, 6: {0: 1, 1: 0}}, v={0:
0.0, 1: 0.9999600005999962, 2: 0.9999700002999992, 3: 0.9999800001000001, 4: 0.99999, 5: 1.0, 6: 0.0})
```

Dans ce cas, la politique optimale est de toujours se déplacer à droite (action 1) sauf lorsque l'agent est dans les états terminaux (0 et 6), où il ne peut prendre aucune action.

Pour la fonction de valeur, on voit que les états plus proches de l'état terminal de droite (6) ont une valeur plus élevée, ce qui est logique puisque cet état terminal a une récompense de 1. Les états plus proches du terminal de gauche (0) ont une valeur plus faible, puisque cet état terminal a une récompense de -1. Les états terminaux eux-mêmes ont une valeur de 0, puisqu'il n'y a pas de récompenses futures à partir de ces états (l'agent ne peut pas prendre d'action à partir des états terminaux).

La fonction **value_iteration** implémente l'algorithme d'itération de valeur, qui est une autre méthode que la policy_iteration pour trouver la politique optimale dans un MDP. La **même fonction** est utilisée pour le **Line World**, le **Grid World** et l'**environnement 1**.

La principale différence entre la **value_iteration** et la **policy_iteration** est la façon dont elles mettent à jour la fonction de valeur et la politique. L'itération de politique met à jour la politique après avoir complètement évalué la fonction de valeur pour la politique actuelle, tandis que l'itération de valeur met à jour la fonction de valeur pour chaque état à chaque itération et dérive la politique de la fonction de valeur finale. Par conséquent, l'itération de valeur peut souvent converger plus rapidement que l'itération de politique, car elle ne nécessite pas une évaluation complète de la politique à chaque étape. Cependant, les deux méthodes convergent vers la politique optimale pour un MDP donné.

La fonction **value_iteration_on_line_world** applique l'algorithme d'itération de valeur à l'environnement Line World afin de déterminer la politique optimale et la fonction de valeur associée pour cet environnement spécifique.

```
value_iteration_on_line_world :
PolicyAndValueFunction(pi={0: {0: 1, 1: 0}, 1: {0: 0, 1: 1}, 2: {0: 0, 1: 1}, 3: {0: 0, 1: 1}, 4: {0: 0, 1: 1}, 5: {0: 0, 1: 1}, 6: {0: 1, 1: 0}}, v={0:
0.0, 1: 0.9999600005999962, 2: 0.9999700002999992, 3: 0.9999800001000001, 4: 0.99999, 5: 1.0, 6: 0.0})
```

On obtient exactement le même résultat que précédemment ce qui semble tout à fait correct.

Nous définissons ensuite le **Grid World**, qui est un environnement plus complexe que le Line World. Dans le Grid World, les états sont des tuples qui représentent les coordonnées dans une grille 5x5, et les actions sont des mouvements dans les quatre directions (gauche, droite, haut, bas). Les récompenses sont -1 pour l'état terminal (0,4), et 1 pour l'état terminal (4,4) et 0 pour les autres états.

La fonction **pi_random_grid_world** prend en entrée un état s et une action a, et renvoie la probabilité de choisir l'action a dans l'état s sous la politique aléatoire uniforme.

Si l'état s est un état terminal, alors aucune action ne peut être prise, donc la fonction renvoie 0.0.

Sinon, il y a 4 actions possibles (haut, bas, droite, gauche), et chaque action est choisie avec une probabilité de 1/4. Ainsi, la fonction renvoie 0.25 pour toute action a.

p_grid_world est la fonction de transition de probabilité pour l'environnement Grid World. Cette fonction détermine la probabilité de passer de l'état s à l'état s' en prenant l'action a et en recevant la récompense r.

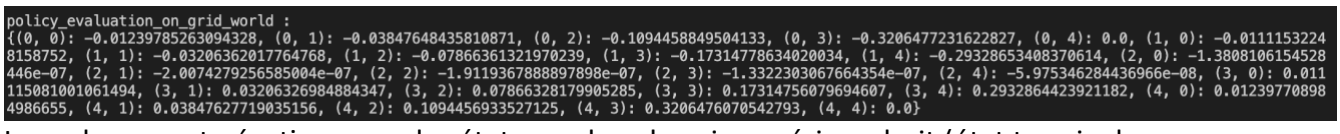
La fonction commence par vérifier que les entrées sont valides. Les états s et s' doivent être des tuples de coordonnées valides dans la grille 5x5, l'action a doit être un nombre entre 0 et 3 correspondant aux directions (0 pour gauche, 1 pour droite, 2 pour haut, 3 pour bas), et r doit être un nombre entre 0 et 2 correspondant aux valeurs de récompense possibles.

Si l'état s est un état terminal (c'est-à-dire (0,4) ou (4,4)), alors aucune action ne peut être prise, donc la fonction retourne 0.0.

Ensuite, la fonction vérifie chaque action possible et retourne 1.0 si l'action mène à l'état s' et produit la récompense r.

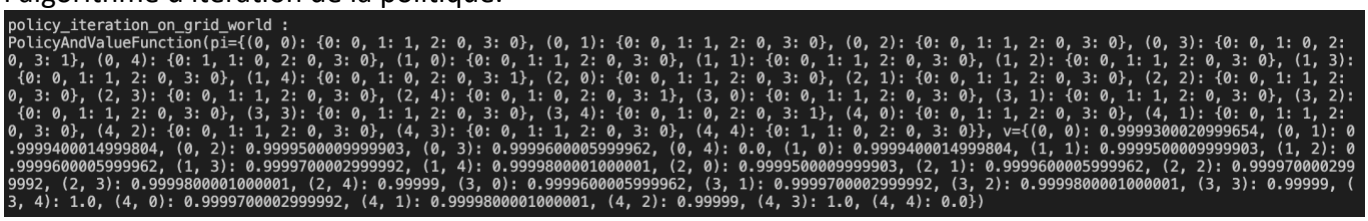
Si aucune des conditions pour les actions et les récompenses n'est remplie, la fonction retourne 0.0, indiquant que la transition de s à s'p avec l'action a et la récompense r est impossible.

policy_evaluation_on_grid_world crée l'environnement Grid World, lance l'algorithme d'évaluation de la politique pour trouver la fonction de valeur de la politique qui choisit uniformément au hasard une action à chaque état (à l'exception des états terminaux où aucune action n'est prise). La fonction de valeur estimée de cette politique est ensuite renvoyée.



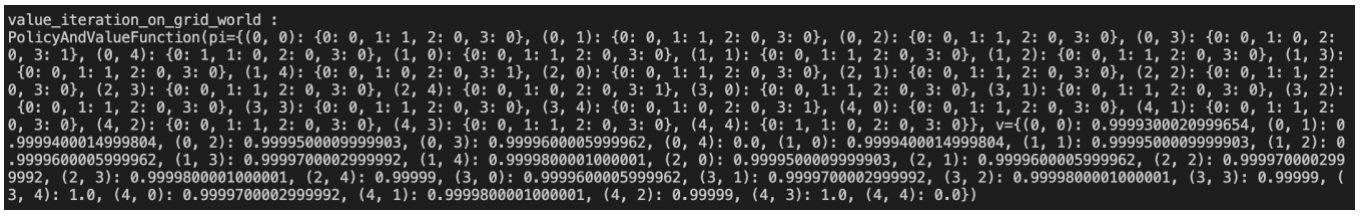
Les valeurs sont négatives pour les états proches du coin supérieur droit (état terminal avec récompense -1) et positives pour les états proches du coin inférieur droit (état terminal avec récompense +1). Cela reflète le fait qu'il est préférable de se diriger vers le coin inférieur droit que vers le coin supérieur droit. L'algorithme semble donc fonctionner correctement.

policy_iteration_on_grid_world crée le même environnement Grid World et lance l'algorithme d'itération de la politique.



La politique est indiquée par le dictionnaire pi, où chaque état est associé à un autre dictionnaire qui indique la probabilité de choisir chaque action à cet état (1 pour l'action choisie, 0 pour les autres). La fonction de valeur v donne la récompense totale attendue à partir de chaque état lorsqu'on suit cette politique optimale. Encore une fois, les valeurs sont plus élevées pour les états proches du coin inférieur droit, reflétant la récompense plus élevée associée à cet état terminal.

Enfin, **value_iteration_on_grid_world** utilise l'algorithme d'itération de la valeur pour trouver la politique optimale et la fonction de valeur associée pour le même environnement Grid World.



Cette fonction a également trouvé la politique optimale et sa fonction de valeur dans l'environnement Grid World, mais en utilisant une méthode différente. Les résultats sont les mêmes que ceux de l'itération de la politique, ce qui est attendu puisque les deux méthodes sont censées converger vers la même politique optimale et la même fonction de valeur.

pi_random_secret_env(s, a) définit une politique uniformément aléatoire pour l'environnement "Secret Env1". Si l'état s est un état terminal, alors aucune action n'est choisie (probabilité de 0). Sinon, toutes les actions sont choisies avec une probabilité égale.

p_secret_env(s, a, s_p, r) renvoie la probabilité de transition de l'état s à l'état s_p en prenant l'action a et en recevant la récompense r.

policy_evaluation_on_secret_env1(), **policy_iteration_on_secret_env1()** et **value_iteration_on_secret_env1()** sont très similaires aux fonctions que nous avons vues précédemment pour le monde en grille et le monde en ligne. Chacune de ces fonctions crée une instance de l'environnement "Secret Env1", extrait les états, les actions et les récompenses de l'environnement, et lance ensuite l'algorithme d'apprentissage par renforcement correspondant (évaluation de la politique, itération de la politique, itération de la valeur).

D'après les résultats obtenus par l'itération de la politique et l'itération de la valeur, la politique optimale pour l'environnement secret 1 est de se rapprocher de l'état 0.

À l'état 0, il faut faire l'action 0 ;
À l'état 1, il faut faire l'action 0.
À l'état 2, il faut faire l'action 1.
À l'état 3, il faut faire l'action 2.
À l'état 4, il faut faire l'action 0.

Si l'agent commence à l'état 0 et suit la politique optimale, il peut s'attendre à obtenir une somme de récompenses de -0.3333.

2 – Monte Carlo

monte_carlo_es_on_tic_tac_toe_solo() utilise l'algorithme de Monte Carlo avec Exploring Starts (MC ES) pour apprendre une politique optimale pour jouer au Tic Tac Toe. Le principe des Exploring Starts est que chaque paire état-action a une chance non nulle d'être la première à être sélectionnée, garantissant ainsi une exploration complète.

L'environnement Tic Tac Toe est initialisé. Les retours somme et les comptes de retours pour chaque paire état-action sont initialisés avec des dictionnaires. Les fonctions de valeur d'action Q et la politique pi sont initialisées avec des valeurs aléatoires.

Pour chaque épisode jusqu'à un nombre prédéfini (70000 dans ce cas), l'environnement est réinitialisé et un état initial et une action sont choisis au hasard. Ensuite, le jeu est joué en suivant la politique actuelle jusqu'à ce qu'il se termine.

Après chaque épisode, les retours pour chaque paire état-action visitée pendant l'épisode sont calculés et utilisés pour mettre à jour Q. Si une paire état-action est visitée plus d'une fois au cours d'un même épisode, seule la première visite est prise en compte (First-Visit MC).

La politique est améliorée en la rendant gourmande par rapport à Q, c'est-à-dire en choisissant toujours l'action qui a la plus grande estimation de la valeur.

La fonction renvoie la politique et la fonction de valeur d'action obtenues.

on_policy_first_visit_monte_carlo_control_on_tic_tac_toe_solo() utilise l'algorithme On-Policy First-Visit Monte Carlo Control pour apprendre une politique optimale pour jouer au Tic Tac Toe. Contrairement à MC ES, il n'est pas nécessaire d'utiliser Exploring Starts car la politique est "soft", c'est-à-dire qu'elle choisit toutes les actions avec une probabilité non nulle.

Comme pour la première fonction, l'environnement est initialisé, ainsi que les retours somme, les comptes de retours, Q et pi.

Pour chaque épisode jusqu'à un nombre prédéfini (50000 dans ce cas), l'environnement est réinitialisé et un état initial est choisi au hasard. Ensuite, le jeu est joué en suivant la politique actuelle jusqu'à ce qu'il se termine.

Après chaque épisode, les retours pour chaque paire état-action visitée pendant l'épisode sont calculés et utilisés pour mettre à jour Q. Si une paire état-action est visitée plus d'une fois au cours d'un même épisode, seule la première visite est prise en compte (First-Visit MC).

La politique est améliorée à chaque étape, pas seulement à la fin de l'épisode. La politique est une politique epsilon-greedy par rapport à Q, ce qui signifie qu'elle choisit généralement l'action avec la plus grande estimation de la valeur, mais choisit parfois une action aléatoire pour garantir une exploration continue.

La fonction renvoie la politique et la fonction de valeur d'action obtenues.
Ces deux méthodes sont des exemples de méthodes de Monte Carlo pour l'apprentissage par renforcement, qui apprennent à partir de séquences complètes d'interaction avec l'environnement, appelées épisodes.

La fonction **create_target_policy(Q)** génère une politique qui pour un état donné sélectionne toujours l'action avec la plus grande estimation de valeur d'après la fonction de valeur d'action Q. En d'autres termes, pour chaque état, cette fonction retourne une politique qui sélectionne avec une probabilité de 1 l'action qui maximise la valeur Q. C'est une politique dite "greedy" (gourmande) car elle choisit toujours l'action qui semble être la meilleure selon l'estimation actuelle de Q.

La fonction **off_policy_monte_carlo_control_on_tic_tac_toe_solo()** utilise la méthode de contrôle hors politique Monte Carlo pour apprendre une politique optimale pour un jeu de Tic Tac Toe en solo.

Un environnement Tic Tac Toe est initialisé. Les fonctions de valeur d'action Q et les comptes C sont initialisés avec des valeurs aléatoires. Une politique cible est créée à l'aide de la fonction create_target_policy(Q).
Pour chaque épisode jusqu'à un nombre prédéfini (80 000 dans ce cas), l'environnement est réinitialisé et un état initial est choisi. Ensuite, le jeu est joué en suivant une politique définie jusqu'à ce qu'il se termine.
Après chaque épisode, les retours pour chaque paire état-action visitée pendant l'épisode sont calculés et utilisés pour mettre à jour Q. Si une paire état-action est visitée plus d'une fois au cours d'un même épisode, seule la première visite est prise en compte (First-Visit MC).
La politique est mise à jour en fonction de l'estimation actuelle de Q. Si l'action prise n'est pas l'action qui maximise la valeur Q pour l'état actuel, la boucle est interrompue et on passe à l'étape suivante de l'épisode. Sinon, le poids W est mis à jour.

Une fois tous les épisodes terminés, la politique finale est déterminée en utilisant la politique cible pour tous les états visités.

monte_carlo_es_on_secret_env2() utilise l'algorithme de Monte Carlo avec Exploring Starts (MC ES) sur un environnement appelé "Env2".

L'environnement "Env2" est initialisé. Les retours somme et les comptes de retours pour chaque paire état-action sont initialisés avec des dictionnaires. Les fonctions de valeur d'action Q et la politique pi sont initialisées avec des valeurs aléatoires.
Pour chaque épisode jusqu'à un nombre prédéfini (10000 dans ce cas), l'environnement est réinitialisé et un état initial et une action sont choisis au hasard. Ensuite, le jeu est joué en suivant la politique actuelle jusqu'à ce qu'il se termine.
Après chaque épisode, les retours pour chaque paire état-action visitée pendant l'épisode sont calculés et utilisés pour mettre à jour Q. Si une paire état-action est visitée plus d'une fois au cours d'un même épisode, seule la première visite est prise en compte (First-Visit MC).
La politique est améliorée en la rendant gourmande par rapport à Q, c'est-à-dire en choisissant toujours l'action qui a la plus grande estimation de la valeur.
Une fois tous les épisodes terminés, la politique finale est déterminée en utilisant la politique pour tous les états visités.
La fonction renvoie la politique et la fonction de valeur d'action obtenues.

on_policy_first_visit_monte_carlo_control_on_secret_env2() Cette fonction utilise l'algorithme On-Policy First-Visit Monte Carlo Control sur un environnement "Env2". Voici les étapes de cette fonction :
L'environnement "Env2" est initialisé. Les retours somme et les comptes de retours pour chaque paire état-action sont initialisés avec des dictionnaires. Les fonctions de valeur d'action Q et la politique pi sont initialisées avec des valeurs aléatoires.
Pour chaque épisode jusqu'à un nombre prédéfini (10000 dans ce cas), l'environnement est réinitialisé et un état initial est choisi. Ensuite, le jeu est joué en suivant une politique epsilon-greedy jusqu'à ce qu'il se termine.
Après chaque épisode, les retours pour chaque paire état-action visitée pendant l'épisode sont calculés et utilisés pour mettre à jour Q. Si une paire état-action est visitée plus d'une fois au cours d'un même épisode, seule la première visite est prise en compte (First-Visit MC).
La politique est mise à jour en fonction de l'estimation actuelle de Q.
Une fois tous les épisodes terminés, la politique finale est déterminée en utilisant la politique pour tous les états visités.
La fonction renvoie la politique et la fonction de valeur d'action obtenues.

create_behaviour_policy(actions) crée une politique de comportement qui est une politique uniforme. Pour un ensemble d'actions donné, elle retourne une politique qui choisit chaque action avec une probabilité égale. Cette politique est utilisée pour générer des comportements, c'est-à-dire pour décider quelles actions prendre lors de la génération d'épisodes.

off_policy_monte_carlo_control_on_secret_env2() utilise la méthode Off-Policy Monte Carlo Control pour apprendre une politique optimale pour un environnement appelé "Env2".

L'environnement "Env2" est initialisé. Les fonctions de valeur d'action Q et C sont initialisées avec des valeurs aléatoires. Une politique de comportement et une politique cible sont créées à l'aide des fonctions create_behaviour_policy(actions) et create_target_policy(Q). Pour chaque épisode jusqu'à un nombre prédéfini (10000 dans ce cas), l'environnement est réinitialisé et un état initial est choisi. Ensuite, le jeu est joué en suivant la politique de comportement jusqu'à ce qu'il se termine.

Après chaque épisode, les retours pour chaque paire état-action visitée pendant l'épisode sont calculés et utilisés pour mettre à jour Q.

La politique cible est mise à jour en fonction de l'estimation actuelle de Q. Si l'action prise n'est pas l'action qui maximise la valeur Q pour l'état actuel, la boucle est interrompue et on passe à l'étape suivante de l'épisode. Sinon, le poids W est mis à jour.
Une fois tous les épisodes terminés, la politique finale est déterminée en utilisant la politique cible pour tous les états visités.
La fonction renvoie la politique et la fonction de valeur d'action obtenues.

Le fichier game.py définit une **implémentation de l'environnement du jeu Tic-Tac-Toe avec lequel des joueurs peuvent interagir**.

Player est une classe qui représente un joueur dans le jeu. Chaque joueur a un **sign** (1 ou 2) qui indique quel symbole il utilise pour marquer ses mouvements sur le plateau. Il a aussi un type ('H', 'R' ou 'A') qui indique s'il s'agit d'un joueur humain, d'un joueur qui choisit des mouvements aléatoirement, ou d'un joueur qui choisit des mouvements en fonction d'une politique donnée. La **méthode play** détermine comment le joueur choisit son mouvement.

TicTacToeEnv est une classe qui représente l'environnement du jeu Tic-Tac-Toe. Elle maintient l'état du jeu, y compris le plateau de jeu et les joueurs. Elle fournit plusieurs méthodes pour interagir avec l'environnement, y compris state_id pour obtenir une représentation unique de l'état actuel, is_game_over pour vérifier si le jeu est terminé, act_with_action_id pour faire un mouvement, score pour obtenir le score actuel, available_actions_ids pour obtenir les mouvements possibles, reset pour réinitialiser l'environnement à son état initial, et convertStateToBoard pour convertir un état en une représentation de plateau.

state_id est une méthode qui convertit l'état actuel du plateau en un identifiant unique. C'est utile pour l'apprentissage par renforcement, où nous avons besoin d'une représentation unique pour chaque état.

is_game_over vérifie si le jeu est terminé. Le jeu est terminé soit quand tous les emplacements du plateau sont occupés, soit quand un joueur a gagné. Un joueur gagne s'il a trois de ses symboles alignés horizontalement, verticalement ou en diagonale.

act_with_action_id est une méthode qui prend un mouvement et l'applique au plateau. Le mouvement est spécifié par un action_id, qui est un nombre de 0 à 8 représentant une position sur le plateau de 3x3.

score est une méthode qui retourne le score actuel du jeu. Le score est calculé en fonction de qui a gagné et du nombre de mouvements effectués.

available_actions_ids retourne une liste de tous les mouvements possibles que le joueur peut faire à l'état actuel.

reset réinitialise l'environnement à son état initial.

convertStateToBoard est une méthode qui convertit un identifiant d'état en une représentation de plateau.

3 - Temporal difference learning:

sarsa_on_tic_tac_toe_solo(): Cette fonction met en œuvre l'algorithme SARSA (State-Action-Reward-State-Action) pour apprendre une politique de jeu pour le Tic-Tac Toe en solo. L'algorithme SARSA est une méthode d'apprentissage par renforcement qui utilise les différences temporelles pour estimer la qualité des actions dans un état donné.

Au début de la fonction, un environnement de jeu Tic-Tac Toe est créé et des paramètres tels qu'epsilon (pour la politique epsilon-gloutonne), le nombre d'épisodes, le taux d'apprentissage et le facteur de remise sont définis. Ensuite, une fonction de valeur d'action est initialisée avec des valeurs aléatoires pour chaque action possible dans chaque état.

La fonction entre ensuite dans une boucle où elle exécute un certain nombre d'épisodes de jeu. Dans chaque épisode, elle choisit une action en utilisant une politique epsilon-gloutonne (qui choisit l'action de la plus grande valeur la plupart du temps, mais choisit parfois une action aléatoire pour encourager l'exploration), puis elle met à jour la fonction de valeur d'action en utilisant la formule de mise à jour SARSA.

Enfin, elle construit la politique finale en choisissant l'action de la plus grande valeur pour chaque état. La fonction renvoie la politique finale et la fonction de valeur d'action.

epsilon_greedy_policy(): Cette fonction génère une politique epsilon-gloutonne basée sur une fonction de valeur d'action donnée. Pour chaque action possible dans l'état actuel, elle attribue une probabilité de (epsilon / nombre d'actions) à cette action. Ensuite, elle ajoute (1 - epsilon) à la probabilité de l'action qui a la plus grande valeur estimée. Ainsi, la politique choisit l'action de la plus grande valeur la plupart du temps, mais elle choisit parfois une action aléatoire pour encourager l'exploration.

q_learning_on_tic_tac_toe_solo(): Cette fonction met en œuvre l'algorithme Q-Learning pour apprendre une politique de jeu pour le Tic-Tac Toe en solo. L'algorithme Q-Learning est une méthode d'apprentissage par renforcement qui utilise les différences temporelles pour estimer la qualité des actions dans un état donné.

Au début de la fonction, un environnement de jeu Tic-Tac Toe est créé et des paramètres tels que le nombre d'épisodes, le facteur de remise, le taux d'exploration et le taux d'apprentissage sont définis. Ensuite, une fonction de valeur d'action est initialisée avec des valeurs nulles pour chaque action possible dans chaque état.

La fonction entre ensuite dans une boucle où elle exécute un certain nombre d'épisodes de jeu. Dans chaque épisode, elle choisit une action en utilisant une politique epsilon-gloutonne (qui choisit l'action de la plus grande valeur la plupart du temps, mais choisit parfois une action aléatoire pour encourager l'exploration), puis elle met à jour la fonction de valeur d'action en utilisant la formule de mise à jour Q-Learning.

Enfin, elle construit la politique finale en choisissant l'action de la plus grande valeur pour chaque état. La fonction renvoie la politique finale et la fonction de valeur d'action.

deep_q_learning_on_tic_tac_toe_solo(): Cette fonction met en œuvre l'algorithme Deep Q-Learning pour apprendre une politique de jeu pour le Tic-Tac Toe en solo. Le Deep Q-Learning est une extension du Q-Learning qui utilise un réseau de neurones pour approximer la fonction de valeur d'action. Cela permet à l'agent de gérer des espaces d'états et d'actions plus grands et plus complexes.

Au début de la fonction, un environnement de jeu Tic Tac Toe est créé et des paramètres tels que le nombre d'épisodes, le facteur de remise, le taux d'exploration et le taux d'apprentissage sont définis. Ensuite, une fonction de valeur d'action est initialisée avec des valeurs nulles pour chaque action possible dans chaque état.

La fonction entre ensuite dans une boucle où elle exécute un certain nombre d'épisodes de jeu. Dans chaque épisode, elle choisit une action en utilisant une politique epsilon-gloutonne (qui choisit l'action de la plus grande valeur la plupart du temps, mais choisit parfois une action aléatoire pour encourager l'exploration), puis elle met à jour la fonction de valeur d'action en utilisant la formule de mise à jour Deep Q-Learning.

Enfin, elle construit la politique finale en choisissant l'action de la plus grande valeur pour chaque état. La fonction renvoie la politique finale et la fonction de valeur d'action.

expected_sarsa_on_tic_tac_toe_solo(): Cette fonction met en œuvre l'algorithme Expected SARSA pour apprendre une politique de jeu pour le Tic-Tac Toe en solo. L'algorithme Expected SARSA est une méthode d'apprentissage par renforcement qui utilise les différences temporelles pour estimer la qualité des actions dans un état donné.

Au début de la fonction, un environnement de jeu Tic-Tac Toe est créé et des paramètres tels qu’epsilon (pour la politique epsilon-gloutonne), le nombre d'épisodes, le taux

d'apprentissage et le facteur de remise sont définis. Ensuite, une fonction de valeur d'action est initialisée avec des valeurs nulles pour chaque action possible dans chaque état.

La fonction entre ensuite dans une boucle où elle exécute un certain nombre d'épisodes de jeu. Dans chaque épisode, elle choisit une action en utilisant une politique epsilon-gloutonne (qui choisit l'action de la plus grande valeur la plupart du temps, mais choisit parfois une action aléatoire pour encourager l'exploration), puis elle met à jour la fonction de valeur d'action en utilisant la formule de mise à jour Expected SARSA.

Enfin, elle construit la politique finale en choisissant l'action de la plus grande valeur pour chaque état. La fonction renvoie la politique finale et la fonction de valeur d'action.

sarsa_on_secret_env3(), q_learning_on_secret_env3(), expected_sarsa_on_secret_env3(): Ces fonctions mettent en œuvre respectivement les algorithmes SARSA, Q-Learning et Expected SARSA pour apprendre une politique de jeu pour un environnement secret appelé "Env3". Cet environnement pourrait être n'importe quel environnement d'apprentissage par renforcement, et ces fonctions apprendraient une politique pour cet environnement en utilisant les algorithmes correspondants.

Résultat :

> **Dans le contexte du jeu de Tic Tac Toe, chaque configuration unique du plateau de jeu est un "état", et chaque mouvement possible que l'agent peut faire est une "action". L'agent apprend une politique qui lui dit quelle action prendre dans chaque état.**

- **deep_q_learning_on_tic_tac_toe_solo():**

La "table Q" est une représentation de la fonction Q, qui donne la qualité attendue (d'où le "Q") de chaque action dans chaque état. Plus précisément, la valeur Q pour une certaine action dans un certain état est la récompense attendue que l'agent recevra s'il prend cette action dans cet état, puis suit sa politique actuelle pour toutes les actions futures.

Chaque entrée dans la table Q est une paire clé-valeur où la clé est un état et la valeur est un autre dictionnaire qui donne la valeur Q pour chaque action possible dans cet état. Par exemple, l'entrée 1115: {0: 0.0, 1: 0.0, 2: 0.99999731564544, 3: 0.0, 4: 0.0, 5: 0.0, 6: 0.0, 7: 0.0, 8: 0.0} signifie que dans l'état 1115, l'action 2 a la valeur Q la plus élevée. Cela signifie que, selon la politique actuelle de l'agent, prendre l'action 2 dans l'état 1115 est l'action qui maximise la récompense attendue.

L'agent met à jour les valeurs Q à chaque fois qu'il joue un jeu. Il commence par initialiser toutes les valeurs Q à zéro, puis il essaie différentes actions, observe les récompenses qu'il reçoit, et met à jour les valeurs Q en conséquence.

- **sarsa_on_tic_tac_toe_solo() :**

Par exemple, prenons le premier élément du dictionnaire : 6299462712503834: {6: 0.30956546315665656}. Ici, 6299462712503834 est un état du jeu. Cet état est

probablement codé d'une certaine manière pour représenter la configuration actuelle du plateau de jeu. Le dictionnaire associé {6: 0.30956546315665656} signifie que pour cet état, l'action 6 a une valeur de 0.30956546315665656. Cela signifie que l'algorithme Sarsa a appris que prendre l'action 6 dans cet état donne en moyenne une récompense de 0.30956546315665656.

Un autre exemple serait 33: {0: 1.6770612975983996, 2: 0.8005842898871746, 4: 0.7899622070804725, 5: 0.15627122571969698, 6: -0.19825637472335203, 7: 0.1762580598432025, 8: -0.9549704511059873}. Ici, l'état est 33 et il y a plusieurs actions possibles avec leurs valeurs respectives. Cela signifie que dans l'état 33, l'action 0 a une valeur de 1.6770612975983996, l'action 2 a une valeur de 0.8005842898871746, et ainsi de suite. L'algorithme Sarsa choisira l'action avec la plus grande valeur, donc dans cet état, il choisira l'action 0.

- **q_learning_on_tic_tac_toe_solo():**

Par exemple, prenons l'entrée 18714: {0: -4.319999999999999, 1: 0.0, 2: 0.0, 3: 0.0, 4: 0.0, 5: 0.0, 6: 0.0, 7: 0.0, 8: 0.0}. Ici, 18714 est un état du jeu. Le dictionnaire associé {0: -4.319999999999999, 1: 0.0, 2: 0.0, 3: 0.0, 4: 0.0, 5: 0.0, 6: 0.0, 7: 0.0, 8: 0.0} signifie que pour cet état, l'action 0 a une valeur Q de -4.319999999999999 et toutes les autres actions ont une valeur Q de 0.0. Cela signifie que l'algorithme Q-learning a appris que prendre l'action 0 dans cet état est probablement une mauvaise idée (car sa valeur Q est négative), tandis que pour les autres actions, il n'a pas d'information particulière (leurs valeurs Q sont 0.0).

C'est ainsi que l'algorithme Q-learning apprend à jouer au jeu de Tic Tac Toe : en expérimentant différentes actions dans différents états et en mettant à jour les valeurs Q en fonction des récompenses reçues. Avec le temps, il devrait être capable de développer une stratégie optimale pour jouer au jeu.

- **expected_sarsa_on_tic_tac_toe_solo():**

La sortie de cette fonction est une instance de la classe PolicyAndActionValueFunction. Cette classe contient deux attributs principaux : pi et q.

pi est la politique que l'agent suit. C'est un dictionnaire où les clés sont les états du jeu et les valeurs sont un autre dictionnaire. Ce dernier dictionnaire a des actions comme clés et la probabilité de choisir cette action dans l'état donné comme valeurs. Par exemple, dans l'état 0, l'action 0 a une probabilité de 0.9111111111111111 d'être choisie, tandis que toutes les autres actions ont une probabilité de 0.011111111111111112.

q est la fonction de valeur d'action. C'est un dictionnaire où les clés sont les états du jeu et les valeurs sont un autre dictionnaire. Ce dernier dictionnaire a des actions comme clés et la valeur estimée de cette action dans l'état donné comme valeurs. Par exemple, dans l'état 0, l'action 0 a une valeur estimée de 2.2588285714285705, tandis que toutes les autres actions ont une valeur estimée de 0.0.