TD n^o 8 : Apprentissage par renforcement : algorithme ε-greedy

Exercice 1 : (**Trouver le processus gaussien ayant la meilleure moyenne**)

On considère la situation suivante : l'environnement est constitué de N processus gaussiens numéroté de 0 à $N-1$ renvoyant chacun une récompense (reward) suivant une distribution normale dont on ne connait pas ni la moyenne, ni l'écart-type.

On se propose d'utiliser l'algorithme ε-greedy pour trouver quel processus gaussien a la plus grande moyenne. Autrement dit, on joue contre l'environnement en misant sur un entier dans l'intervalle $[0, N-1]$, on reçoit une récompense. le but de l'apprentissage est de trouver quel est le processus gaussien qui optimise la récompense, ou plus exactement l'espérance de la récompense.

1. Création d'une classe `Gaussian` ayant 3 méthodes :
   — `__init__` qui prend 2 arguments et initialise les moyenne et écart-type du processus gaussien,
   — `reward` qui choisit un nombre suivant la distribution normale en question, et
   — `sampling` qui renvoie un échantillon de taille N.
   (`numpi.random.randn()` renvoie un nombre aléatoire suivant la loi normale centrée réduite)
2. Création d'une classe `Model` ayant 2 méthodes.
   — La méthode `__init__` initie le modèle avec une estimation de la moyenne nulle et avec le nombre de fois que l'on a joué ce modèle.
   — La méthode `update` prenant un argument (reward obtenu) et mettant à jour l'estimation de la moyenne (des rewards obtenus lorsqu'on a joué ce modèle)
3. Dans ce cas-là, l'algorithme ε-greedy est vraiment simple. Pour chaque partie du jeu, on choisit un nombre aléatoire uniformément distribué dans $[0, 1]$ (`numpy.random.random()`), et s'il est inférieur à ε, alors (exploration) on tire au hasard un nombre dans $[0, 1, \ldots, N-1]$ (correspondant à l'une des gaussiennes de l'environnement). Sinon (exploitation), on va jouer le nombre de $[0, 1, \ldots, N-1]$ correspondant à la gaussienne dont l'estimation de la moyenne est la meilleure.
   Dans les deux cas, on reçoit un *reward*. Ce reward permet de mettre à jour l'estimation de la moyenne de la gaussienne jouée.
4. Faire varier le nombre de parties pour apprendre, la valeur de ε, le nombre de gaussiennes dans l'environnement.

On pourra s'inspirer du code à trous `ex1-eps-greedy-students.py` disponible à `https://www.i3s.unice.fr/~comet/SUPPORTS/`

Exercice 2 : (**Jeu des allumettes**)

Ce jeu se joue à deux. Les joueurs sont devant un certain nombre d'allumettes (chaque partie commence avec le même nombre d'allumettes). A chaque tour, il faut en enlever 1, 2 ou 3. Celui qui prend la dernière perd la partie.

Nous souhaitons implémenter l'algorithme ε-greedy pour apprendre à jouer à ce jeu-là [1].

1. Création d'une classe `StickGame` ayant 5 méthodes :
   — `__init__` qui initialise le jeu : on stocke dans deux variables (`original_nb` et `nb`) le nombre initial d'allumettes passé en argument.
   — `is_finished` qui teste si le jeu est fini (le nombre d'allumettes restantes est-il inférieur ou égal à 0 ?).
   — `reset` qui réinitialise le nombre d'allumettes sur le plateau au nombre initial choisi, et renvoie ce nombre.
   — `display` qui affiche à l'écran l'état du jeu.
   — `step` qui modifie l'état du jeu en fonction du nombre d'allumettes choisi par un joueur, et qui renvoie un *reward* négatif si le joueur a pris la dernière allumette ou nul dans le cas contraire.
2. Création d'une classe `StickPlayer` ayant 6 méthodes :

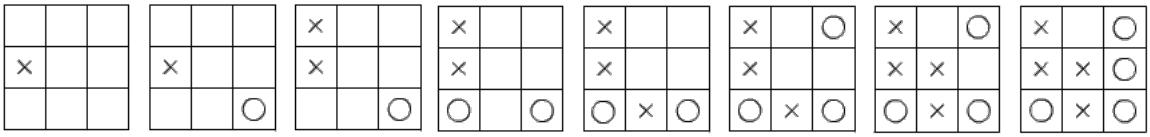
1. En fait, pour ce jeu-là, il y a une stratégie gagnante : pour gagner, il faut laisser à chaque tour un nombre d'allumettes correspondant à un multiple de 4 plus 1, c'est-à-dire : 1, 5, 9, 13, 17, 21, 25, 29, 33, ... Par exemple, s'il reste 25 allumettes et que l'adversaire prend x allumettes, vous en retirez $4-x$ pour qu'il n'en reste plus que 21. Ainsi, dès que vous pouvez laisser $4 \times n + 1$ allumettes, cette stratégie vous permet de gagner.

— `__init__` qui initialise le statut *humain* ou *non humain* du joueur, son statut *entrainable* ou non, la structure de données pour stocker les valeurs d'approximation de $V(state)$, le nombre de parties gagnées et perdues, le paramètre initial ε_0.
— `reset_stat` qui réinitialise le nombre de parties gagnées et perdues, ainsi que la liste des *rewards* obtenus.
— `greedy_step` qui permet d'extraire l'action à choisir qui maximise la valeur de l'état atteint grace à cette action.
— `play` qui implémente l'algorithme ε-greedy. A chaque étape, on choisit un nombre aléatoire uniformément distribué dans $[0, 1]$ (`numpy.random.random()`), et s'il est inférieur à ε, alors (exploration) on tire au hasard un nombre dans $[1, 2, 3]$ (correspondant à l'une des actions possibles). Sinon (exploitation), on va fait appel à `greedy_step` pour avoir l'action à jouer. Evidement, si le joueur est humain, il faut demander à l'utilisateur son choix.
— `add_transition` permet de mémoriser dans une liste la succession des (s, a, r, ns) où s est l'état du jeu, a l'action sélectionnée, r la récompense obtenue, et ns le nouvel état atteint. On peut aussi mémoriser la liste des récompenses obtenues.
— `train` permet la mise à jours des valeurs $V(states)$ lorsqu'une partie est finie. Cette méthode dépend du ratio d'apprentissage.

3. Création d'une fonction `play` qui lance un jeu. Cette fonction prend en paramètre un jeu (classe `StickGame`), deux joueurs de la classe `StickPlayer`, et trois booléens spécifiant si au moins un joueur doit entraîner son modèle, si le joueur qui commence est tirer au sort ou non, et s'il s'agit d'une compétition ou non (en compétition, on n'utilise pas l'exploration, on ne fait que de l'exploitation).
   Cette fonction permet de lancer le jeu : on choisi qui commence, et tant que la partie n'est pas fini, on fait jouer les deux joueurs alternativement,
4. Le programme principal qui déclare deux joueurs artificiels entrainables, un joueur aléatoire non entrainable et un joueur humain. Il faut alors lancer 10000 parties pour entrainer les deux joueurs artificiels. Ce programme pourra alors lancer 1000 parties entre l'un des deux joueurs artificiels entrainés contre le joueur artificiel non entrainable, puis pourra proposer de faire jouer l'utilisateur humain contre l'un des deux joueurs entrainés.

On pourra s'inspirer du code à trous `ex2-stick-students.py` disponible à
`https://www.i3s.unice.fr/~comet/SUPPORTS/`

Exercice 3 : (**Le Morpion**)



En vous inspirant de la structure du code de l'exercice précédent, coder le jeu du Morpion. Il y aura, comme dans le cas précédent, une classe `TicTacToeGame` et une classe `TicTacToePlayer`, une fonction `play` et un programme principal qui lance l'apprentissage entre 2 joueurs artificiels entrainables, puis lance des parties entre un joueur entrainé et un joueur artificiel non entrainé, puis qui propose à l'utilisateur de jouer contre un joueur articiciel entrainé.

Les difficultés ici seront de plusieurs ordres :

1. Comment représenter une position du jeu ? Ici on peut penser à une chaine de caractères de taille fixe où chaque caractère représente une case particulière du plateau de jeu.
2. Le nombre d'états possible est assez grand. Si l'apprentissage de passe pas en revue lors de l'exploration tous les états, le joueur entrainé ne réagira pas correctement lorsqu'il rencontrera cet état.
3. Pour que le joueur artificiel entrainé se comporte bien face à un humain ou face à un joueur artificiel non entrainé, il est nécessaire qu'il se base plus sur l'exploitation. Si le taux d'exploration reste trop haut lors de ces phases de "compétition", il choisira des actions non optimales alors qu'il connaît la bonne stratégie.