Projet 1 : Exploration/Exploitation et Puissance 4

A faire en binôme !
Rendu sur Moodle avant le TME 5
Contacts :
`bouthors@isir.upmc.fr` *(groupe mardi)*
`louis.serrano@sorbonne-universite.fr` *(groupes jeudi)*

Introduction

Le dilemme dit de *l'exploration vs exploitation* est un problème fondamental que l'on retrouve dans plusieurs domaines de l'intelligence artificielle, en particulier en Machine Learning : parmi un certain nombre de choix possibles, vaut-il mieux *exploiter* la connaissance acquise et choisir l'action estimée la plus rentable ou vaut-il mieux continuer à *explorer* d'autres actions afin d'acquérir plus d'informations ? L'exploitation consiste à faire la meilleure décision à partir de toute l'information collectée, l'exploration consiste à obtenir plus d'information. Il est parfois préférable, souvent au début d'un processus, de faire des sacrifices et de ne pas choisir l'option a priori la plus rentable afin d'améliorer le gain à long terme. Mais la question reste de savoir quand arrêter d'explorer, i.e. quand estime-t-on avoir recueilli assez d'informations et que l'exploration n'apportera pas de connaissances supplémentaires.
Un premier exemple d'application est la publicité en ligne, où une régie publicitaire doit choisir une catégorie de pub parmi un certain nombre possibles pour sélectionner une pub à afficher à un utilisateur. L'historique de l'utilisateur est connu, à savoir à quelles catégories appartenaient les pubs présentées dans le passé et celles qui l'ont intéressées ou non. Est-il plus profitable d'afficher une pub de la catégorie qu'il a le plus choisi, quitte à ne pas être sûr que ce soit sa catégorie préférée ? ou d'explorer et de lui présenter une pub d'une autre catégorie qui peut être de plus grand intérêt pour lui ? Le risque d'identifier une catégorie sous-optimale, i.e. qui n'est pas la plus appréciée de l'utilisateur, est d'avoir un rendement plus faible et donc de perdre de l'argent sur le long terme. Mais à chaque fois que l'on explore une catégorie qui n'intéresse pas l'utilisateur, on perd également de l'argent... Le compromis entre la phase exploratoire et la phase d'exploitation est donc cruciale pour optimiser le rendement sur le long terme.
Un autre exemple d'application est l'intelligence artificielle pour les jeux de stratégie. C'est d'ailleurs un concept au cœur des premières IA qui ont révolutionnées l'approche pour le jeu de GO, que l'on retrouve également sous une forme plus complexe dans AlphaGO. La question qui se pose dans ce contexte est de savoir s'il vaut mieux jouer le coup identifié comme le meilleur ou faut-il tenter un coup qui a été peu joué, au risque bien sûr de perdre la partie.
L'objectif de ce projet est d'étudier différents algorithmes du dilemme d'exploration vs exploitation pour des IAs de jeu. Le jeu étudié sera dans un premier temps le puissance 4 qui a l'avantage d'avoir une combinatoire simple (et donc à porter d'une implémentation naïve sans grande puissance computationelle).
La partie 1) est dédiée à l'implémentation et à l'analyse combinatoire du jeu, théorique et pratique. La partie 2 vous demande d'implémenter un algorithme de Monte-Carlo pour la résolution du jeu. La partie 3 étudie les algorithmes classiques d'exploration vs exploitation dans un cadre simple. La partie 4 est consacrée à l'étude des algorithmes avancés pour les jeux combinatoires.
Vous aurez à rendre à l'issue du mini-projet votre code et un rapport expérimetal.

1 Combinatoire du puissance 4

Le jeu du *Puissance 4* est un jeu à 2 joueurs qui se joue sur un plateau de 7 colonnes et 6 lignes. Chaque joueur dispose de 21 jetons d'une couleur donnée (généralement rouge pour l'un, bleu pour l'autre). L'objectif est d'aligner 4 jetons de sa couleur sur le plateau (verticalement, horizontalement ou en diagonal). Tour à tour, chaque joueur place un pion sur une des 7 colonnes et le pion tombe jusqu'à la première position non occupée de la colonne. Le jeu s'arrête dès que 4 jetons de la même couleur sont alignées ou si toutes les cases sont occupées.
Cette partie est dédiée à l'implémentation du jeu et à l'étude combinatoire du jeu.
Conseils pour l'implémentation : ces recommendations ne sont pas obligatoires mais vivement conseillées.

- Pour toute la suite, codez vos fonction de manière générique de manière à ce qu'elles puissent traiter n'importe quelle taille de plateau. Vous pouvez définir des variables globales au début de votre code pour spécifier la taille de plateau étudier.
- Utilisez `numpy` pour coder les tableaux et les opérations mathématiques.
- Utilisez `matplotlib.pyplot` pour les affichages graphiques.
- Codez des classes pour le plateau de jeu et pour les joueurs.
- Utilisez un tableau pour représenter le tableau. Vous pouvez par exemple coder par 0 une case vide, 1 le premier joueur et -1 le deuxième.

1. Implémenter une fonction qui calcule la liste de toutes les quadruplets de cases qui doivent être de la même couleur pour gagner : $\{((0,0),(0,1),(0,2),(0,3)),((0,1),(0,2),(0,3),(0,4)),\ldots\}$. N'oubliez pas les diagonales !
2. Implémenter le moteur du jeu (sous forme de classe de préfèrence). Des indications vous sont données à la fin de l'énoncé pour l'implémentation. Il doit comporter les éléments suivants :
   - un tableau qui représente le plateau du jeu
   - une fonction `reset()` qui permet de réinitialiser le jeu
   - une fonction `has_won()` qui permet de tester la victoire d'un des deux joueurs (en parcourant par exemple la liste des quadruplets et en testant s'ils sont de la même couleur).
   - une fonction `play(x,joueur)` qui permet de placer un jeton dans la colonne x pour le joueur spécifié.
   - une fonction `is_finished()` qui permet de tester si c'est la fin du jeu (plateau plein ou victoire d'un joueur).
   - une fonction `run(joueur1,joueur2)` qui permet de jouer une partie entre le joueur 1 et le joueur 2 : ils jouent à tour de rôle tant que la partie n'est pas finie. Elle renvoie 1 ou -1 selon la victoire du joueur 1 ou 2, et 0 en cas de nul. On supposera que chaque joueur est muni d'une fonction `play(plateau,joueur)` qui renvoie le coup à jouer pour ce joueur en lui précisant s'il est le joueur 1 ou 2.
   - Vous pouvez utiliser ce codage des joueurs dans tous le projet : 1 pour le joueur 1 et -1 pour le joueur 2.
3. Implémenter un joueur qui joue aléatoirement parmi les coups possibles.
4. Etudier la distribution du nombre de coups avant une victoire lorsque les deux joueurs jouent aléatoirement en différenciant selon que ce soit le premier ou le deuxième

joueur qui gagne (la variable aléatoire dénotant le nombre de coups d'une partie). Observez vous une différence ? Est-elle normale ? Cette distribution vous semble-t-elle suivre une loi de probabilité usuelle ? Argumenter.

5. Proposer une expérience pour trouver la probabilité d'une partie nulle lorsque les deux joueurs jouent aléatoirement.

   (Questions ouvertes)

6. Donner une borne théorique à la probabilité d'une partie nulle quand les joueurs jouent aléatoirement et avec toutes les simplifications/hypothèses que vous voulez (par exemple jouer une partie jusqu'à ce que toutes les cases soient occupées même si un des deux joueurs à gagner avant).
7. Donner une borne théorique aux nombres de parties différentes qui peuvent être joués. Proposer une expérience pour approximer ce nombre.
8. Faire varier le nombre de colonnes et de lignes et étudier ce qui change pour les lois étudiées précédement.

2 Algorithme de Monte-Carlo

On appellera *état* dans la suite une configuration donnée du plateau. N.B. : dans le jeu du puissance 4, en considérant que c'est toujours la même couleur qui commence, la configuration du plateau indique également quel joueur doit jouer à ce tour. Toute l'information est donc contenu dans l'état. On appelera *action* le coup qu'un joueur peut jouer (dans notre cas, le choix d'une colonne pour poser un jeton).
Comme étudier dans la section précédente, la combinatoire du jeu de puissance 4 est un peu trop importante pour pouvoir calculer exactement le meilleur coup à jouer - à l'aide d'un arbre du jeu à partir d'un état donné par exemple. L'algorithme de Monte-Carlo est un algorithme probabiliste qui vise à donner une approximation d'un résultat trop complexe à calculer : il s'agit d'échantillonner aléatoirement et de manière uniforme l'espace des possibilités et de rendre comme résultat la moyenne des expériences. Dans le cas d'un jeu tel que le puissance 4, à un état donné, plutôt que de calculer l'arbre de toutes les possibilités pour choisir le meilleur coup, il s'agit de jouer pour chaque action possible un certain nombre de parties au hasard et d'en moyenner le résultat.
L'algorithme détaillé est le suivant :

```
Monte Carlo play(etat,joueur):
    initialiser les récompenses des actions à 0
      (elles représentent la moyenne des victoires par action)
    Pour i de 1 à N:
        choisir une action a au hasard
        Tant que la partie n'est pas finie:
            jouer les deux joueurs au hasard
        mettre à jour la récompense de l'action a en fonction du résultat
    Retourner l'action avec la meilleure probabilité de victoire
```

L'action avec la moyenne de victoire la plus haute est finalement choisie. Implémenter un joueur Monte-Carlo et tester ce joueur contre lui-même et contre le joueur aléatoire. Faites les mêmes analyses que précédement sur la distribution du nombre de coups joués avant une victoire et selon le joueur.

3 Bandits-manchots

Afin de formaliser le problème de l'exploration/exploitation, on prend souvent l'exemple des bandits manchots (ou machine à sous), ce jeu de hasard qu'on retrouve dans tout casino qui se respecte : pour une mise, on a le droit d'actionner un levier qui fait tourner des rouleaux, et en fonction de la combinaison obtenue sur les rouleaux, une récompense est attribué au joueur. Supposons une machine à sous à N leviers dénotés par l'ensemble $\{1, 2, \ldots, N\}$. Chacun de ses leviers est une action possible parmi lesquelles le joueur doit choisir à chaque pas de temps : l'action choisie à l'instant t sera appelée a_t (un entier entre 1 et N). Pour simplifier la modélisation, nous supposerons dans la suite que la récompense associée à chaque levier i suit une distribution de Bernoulli de paramètre μ^i : avec une probabilité μ^i le joueur obtient une récompense de 1, avec une probabilité $1 - \mu^i$ le joueur obtient une récompense de 0. Cette récompense obtenue au temps t lorsque le joueur joue sera notée r_t. En notant a_t l'action jouée au temps t, r_t est donc une variable aléatoire qui suit une loi de Bernoulli de paramètre μ^{a_t}. On suppose de plus que le rendement de chaque levier est stationnaire dans le temps, c'est-à-dire que les μ^i sont constants tout au long de la partie.
Pour le joueur, le gain au bout de T parties est la somme des récompenses qu'il a obtenu pendant les T première parties, soit $G_T = \sum_{t=0}^{T} r_t$ (n.b. la récompense étant aléatoire, le gain G_T est une variable aléatoire, tout comme r_t). Son but est bien sûr de maximiser ce gain. Pour cela, il faut que le joueur identifie le levier au rendement le plus élevé : $i^* = argmax_{i\in\{1,\ldots,N\}}\mu^i$ et le rendement associé : $\mu^* = \mu^{i^*} = max_{i\in\{1,\ldots,n\}}\mu^i$. Si le joueur joue un autre levier que i^*, il aura en moyenne un gain total inférieur au gain maximal qu'il peut espérer. Ce gain maximal à un temps T s'écrit $G_T^* = \sum_{i=1}^{T} r_t^*$, avec r_t^* la récompense aléatoire tirée de la distribution de Bernoulli de paramètre μ^*. On appelle regret au temps T la différence entre le gain maximal espéré et le gain du joueur : $L_T = G^* - \sum_{t=1}^{T} r_t = \sum_{t=1}^{T}(r_t^* - r_t)$. L'objectif est donc de minimiser ce regret.

Nous noterons par la suite :

- $N_T(a)$ le nombre de fois où l'action (le levier) a a été choisi jusqu'au temps T.
- $\hat{\mu}_T^a = \frac{1}{N_T(a)} \sum_{t=1}^{T} r_t \mathbf{1}_{a_t=a}$ la récompense moyenne estimée pour l'action/levier a à partir des essais du joueur.

Nous étudierons dans la suite les algorithmes suivants :

- **l'algorithme aléatoire** qui choisit a_t uniformément parmi toutes les actions possibles. C'est ce qu'on appelle une *baseline*, un algorithme référence que tous les autres algorithmes doivent battre.
- **l'algorithme *greedy*** (ou glouton) : un certain nombre d'itérations sont consacrés au début à l'exploration (on joue uniformément chaque levier) puis par la suite on choisit toujours le levier dont le rendement estimé est maximal : $a_t = argmax_{i\in\{1,\ldots,N\}}\hat{\mu}_t^i$. Cet algorithme fait purement de l'exploitation.
- **l'algorithme ϵ-*greedy*** : après une première phase d'exploration optionelle, à chaque itération : avec une probabilité ϵ on choisit au hasard uniformément parmi les actions possibles, avec une probabilité $1 - \epsilon$ on applique l'algorithme *greedy* : $a_t = argmax_{i\in\{1,\ldots,N\}}\hat{\mu}_t^i$. Cet algorithme explore continuellement.
- **l'algorithme UCB** [1] : l'action choisie est $a_t = argmax_{i\in\{1,\ldots,N\}}\left(\hat{\mu}_t^i + \sqrt{\frac{2\log(t)}{N_t(i)}}\right)$. Le premier terme est identique aux autres algorithmes, il garantit l'exploitation ; le deuxième

1. Upper Confidence Bound

terme lui devient important lorsque le ratio entre le nombre de coups total et le nombre de fois où une action donnée a été choisie devient grand, c'est-à-dire qu'un levier a été peu joué : il garantie l'exploration.

Expériences

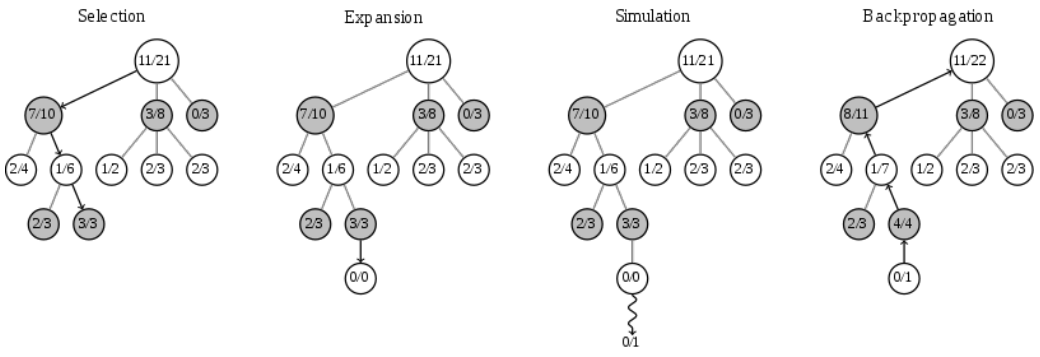
L'ensemble des leviers est représenté par la liste des paramètres de la loi de Bernoulli associée à chaque levier, soit la liste des μ^i, N réels compris entre 0 et 1.

1. Coder une première fonction qui prend en argument une liste de leviers (sous forme d'une liste de réels entre 0 et 1 représentant le rendement de chaque levier) et un entier représentant l'action/levier choisi. La fonction rend le gain binaire correspondant au coup joué : elle fait un tirage de Bernoulli de paramètre l'action choisie et rend le résultat.
2. Coder les quatre algorithmes ci-dessus. Afin d'homogénéiser votre code, il faut considérer qu'ils prennent tous deux arguments, le premier une liste des récompenses moyennes estimées pour chaque levier (la liste des $\hat{\mu}^i$) et le deuxième le nombre de fois où chaque levier a été joué (la liste des $N(i)$) (même si ce dernier paramètre n'aura une utilité que dans le cadre de l'algorithme UCB).

Vous comparerez les différents algorithmes en faisant varier en particulier le nombre de leviers, la distribution des récompenses, les écarts entre les différents paramètres des lois de Bernoulli. Penser à tracer le regret en fonction du temps. Observez vous des formes particulières de regret ? Analyser et commenter vos résultats.

4 Arbre d'exploration et UCT

UCT désigne l'algorithme UCB adapté aux arbres de jeu (communément appelé également Monte Carlo Tree Search). Contrairement à l'algorithme précédent qui explore complétement aléatoirement l'arbre des possibilités, l'idée de l'algorithme est d'explorer en priorité les actions qui ont le plus d'espoir d'amener à une victoire, tout en continuant à explorer d'autres solutions possibles. On retrouve le dilemme d'exploration/exploitation : vaut-il mieux continuer à lancer des simulations aléatoires sur une action qui est pour l'instant la plus performante afin de s'assurer de sa qualité, ou vaut-il mieux explorer d'autres actions possibles ? Pour cela, le principe est d'utiliser l'algorithme UCB à chaque embranchement possible, afin d'équilibrer l'exploration et l'exploitation. Le schéma ci-dessous décrit les différentes étapes de l'algorithme (source `https://en.wikipedia.org/wiki/Monte_Carlo_tree_search`).



La racine correspond à l'état courant du jeu. Comme dans le cas de l'algorithme de Monte-Carlo, on a besoin d'une stratégie par défaut pour explorer rapidement la qualité d'un état (la probabilité de gagner à partir de cet état). Cette stratégie par défaut est la stratégie aléatoire. Au tout début, aucune information n'est disponible, on va simuler pour chaque action une partie avec le joueur aléatoire pour initialiser les nœuds enfants de la racine. Seuls ces nœuds enfants, correspondant chacun à une action, sont pour l'instant créés ; les états visités lors de la simulation du jeu ne sont pas stockés ! Chaque nœud ainsi développé stockera le nombre de fois où ce nœud à mener à une victoire et le nombre de fois où il a été joué (à participer à une simulation, victoire défaite ou match nul). A partir d'un arbre déjà en partie exploré, les différentes étapes sont les suivantes :

1. Sélection : un nœud à explorer est sélectionné dans l'arbre. Pour cela, en partant de la racine, on choisit le nœud suivant en fonction de l'algorithme UCB parmi les enfants du nœud courant, jusqu'à tomber soit sur une feuile de l'arbre (un nœud sans enfant) soit sur un état où une des actions n'a jamais été explorée (pas de nœud fils correspondant à cette action).
2. Expansion : une fois le nœud sélectionné, pour une action jamais effectué à ce nœud, un nœud fils est créé et simulé.
3. Simulation : à partir d'un nœud à simuler, un jeu entre deux joueurs aléatoires est déroulé jusqu'à atteindre un état final (victoire, défaite ou match nul). Les états visités lors de la simulation ne sont pas ajoutés à l'arbre, ils ne sont pas stockés.
4. On rétro-propage le résultat de la simulation à tout le chemin menant au fils simulé : pour chaque nœud sur le chemin, on met à jour en fonction du résultat le nombre de victoires et le nombre de visites.

Le processus est itéré en fonction du nombre de simulation (ou temps) disponible.

Expérimentations
Implémenter l'algorithme UCT. Faire jouer les joueurs aléatoires et Monte-Carlo contre UCT. Essayer de faire varier le comportement exploratoire d'UCT (en ajoutant un facteur multiplicatif devant le deuxième terme permettant de moduler l'exploration/exploitation). Faire varier également le nombre de simulations autorisées pour chaque coup.

(Question Bonus) Faire varier les dimensions du plateau et analyser l'effet sur les probabilités de victoires de chaque joueur, la difficulté de chaque partie et/ou d'autres caractéristiques du jeu.

Fonctions python utiles

- `tab[(x0,x1,x2,x3),(y0,y1,2,3)]` permet de rendre le tableau 1D constitué des cases `[(x0,y0),(x1,y1),(x2,y2),(x3,y3)]`.
- `np.histogram` et plt.hist permettent d'obtenir et deepgreen tracer des histogrammes (normalisés ou non).
- pour connaître la première ligne libre du plateau en colonne `x`, vous pouvez utiliser : `plateau[x,:]==0).argmax()`.
- pour savoir si un quadruplet `p` de cases `[(x0,x1,x2,x3),(y0,y1,y2,y3)]` est gagnant, vous pouvez utliser `np.abs(self.plateau[p].sum())==4`.
- La fonction `tobytes()` d'un tableau permet de transformer le tableau en chaine de caractère. Cela vous sera utile pour sérialiser un état du jeu et le stocker dans un dictionnaire ou autre (afin par exemple de savoir si vous avez déjà rencontré cet état dans une simulation précédente). La fonction inverse est `np.frombuffer()`.