

TME 2-4 : Projet Exploration/Exploitation

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 morpion qui a l'avantage d'avoir une combinatoire simple (et donc à porter d'une implémentation naïve sans grande puissance computationnelle). La partie 1) est dédiée à l'expérimentation des algorithmes classiques d'exploration vs exploitation dans un cadre simple. La partie 2 vous demande d'implémenter un algorithme de Monte-Carlo pour la résolution du jeu du morpion et la partie 3 est consacrée à l'étude des algorithmes avancés pour les jeux combinatoires. Enfin, la partie 4, bonus, étudie le jeu du Puissance 4.

1 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 attribuée au joueur. Supposons une machine à sous à N leviers dénotés par l'ensemble $\{1, 2, \dots, N\}$. 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 supposons 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 (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 premières parties, soit $G_T = \sum_{t=1}^T r_t$ (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 : $i^* = \operatorname{argmax}_{i \in \{1, \dots, N\}} \mu^i$ et le rendement associé : $\mu^* = \mu^{i^*} = \max_{i \in \{1, \dots, N\}} \mu^i$. 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 $G_T^* = \sum_{t=1}^T \mu^*$, 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 : $L_T = G_T^* - G_T = \sum_{t=1}^T (\mu^* - r_t)$. 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 : $a_t = \operatorname{argmax}_{i \in \{1, \dots, N\}} \hat{\mu}_t^i$. Cet algorithme fait purement de l'exploitation.
- **l'algorithme ϵ -greedy** : après une première phase d'exploration optionnelle, 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* : $a_t = \operatorname{argmax}_{i \in \{1, \dots, N\}} \hat{\mu}_t^i$. Cet algorithme explore continuellement.
- **l'algorithme UCB¹** : l'action choisie est $a_t = \operatorname{argmax}_{i \in \{1, \dots, N\}} \left(\hat{\mu}_t^i + \sqrt{\frac{2 \log(t)}{N_t(i)}} \right)$. Le premier terme est identique aux autres algorithmes, il garantit l'exploitation ; le deuxième 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 garantit 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.

Coder une première fonction qui prend en argument une machine sous forme de liste et un entier, l'action/levier choisi, et rend le gain binaire correspondant au coup joué. Coder également 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 été utilisé 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.

2 Morpion et Monte-Carlo

Le jeu du Morpion se joue à 2 joueurs sur un plateau de 3×3 cases. Chaque joueur à son tour marque une case avec son symbole (croix ou cercle) parmi celles qui sont libres. Le but est d'aligner pour un joueur 3 symboles identiques en ligne, en colonne ou en diagonale.

Code fourni Télécharger le code fourni. Il permet de modéliser de manière générique un jeu de plateau quelconque simple type dames, morpion, puissance 4, go ... On utilise une modélisation par état du jeu afin de rendre générique les algorithmes de résolution étudiés. Un état du jeu est défini par l'état de chaque case de la grille du jeu et le joueur qui doit jouer à ce tour. La classe `State` modélise un état générique, elle dispose de :

- deux constantes `NX, NY` qui définissent la taille de la grille ;
- une variable `grid`, tableau `numpy` de taille `(NX, NY)` ; un 0 indique une case libre, un 1 une case jouée par le premier joueur, et un -1 une case jouée par le deuxième joueur ;
- une variable `courant` qui indique le joueur courant, 1 pour le joueur 1, -1 pour le joueur 2 ;
- une méthode `next(coup)` qui permet de renvoyer le nouvel état lorsque le coup est joué. Cette méthode ne change pas l'état, elle renvoie un nouvel état ;
- la méthode `get_actions()` permet de renvoyer les coups possibles pour le joueur courant ;
- la méthode `win()` renvoie 1 si le joueur 1 a gagné, -1 si le joueur 2 a gagné, 0 si match nul ou pas terminé ;
- la méthode `stop()` renvoie `True` si le jeu est fini, `False` si le jeu continue.
- la méthode `hash()` qui permet d'avoir une représentation string (hash) d'un état, et `fromHash(hash)` méthode statique qui prend un hash et renvoie un état.

La classe `Jeu` permet de lancer une partie avec deux joueurs (variables `j1` et `j2`) à partir d'un état initial `init_state`. Une variable `log` permet de logger les coups qui sont joués. La méthode `run(draw, pause)` permet de lancer le jeu, avec ou sans affichage selon la valeur de `draw` et une pause de `pause` entre chaque coup si l'affichage est demandé. Elle retourne à la fin le joueur victorieux et le log. Elle dispose également d'une méthode `replay(log)` pour rejouer un log. Le log est une liste composée de couples (état, coup).

Un joueur est une instance de la classe `Agent`, qui nécessite uniquement la présence d'une méthode `get_action(state)`. Cette méthode est appelée à chaque fois que le joueur doit

joué avec en paramètre l'état courant du jeu (qui indique comme dit ci-dessus à la fois l'état de la grille et le joueur qui doit joué, donc de cette manière le joueur sait s'il est joueur 1 ou 2).

Enfin la classe `MorpionState` implémente l'état pour le jeu du Morpion. Elle hérite bien sûr de `State`, précise que le jeu se joue sur une grille de 3 par 3; la méthode `get_actions()` renvoie la liste des cases à 0; la méthode `win()` recherche si 3 symboles sont alignés et renvoie dans ce cas l'identifiant du joueur qui a gagné; la méthode `stop()` teste si un joueur a gagné ou s'il n'y a plus de coups à jouer.

Joueur aléatoire Dans un premier temps, coder un joueur qui joue de manière aléatoire. Simuler un grand nombre de partie entre deux joueurs aléatoires et tracer l'évolution de la moyenne du nombre de partie gagnée du premier joueur et du deuxième joueur. A quelle loi obéit la variable aléatoire qui dénote la victoire du premier joueur? Quelle est son paramètre, sa variance?

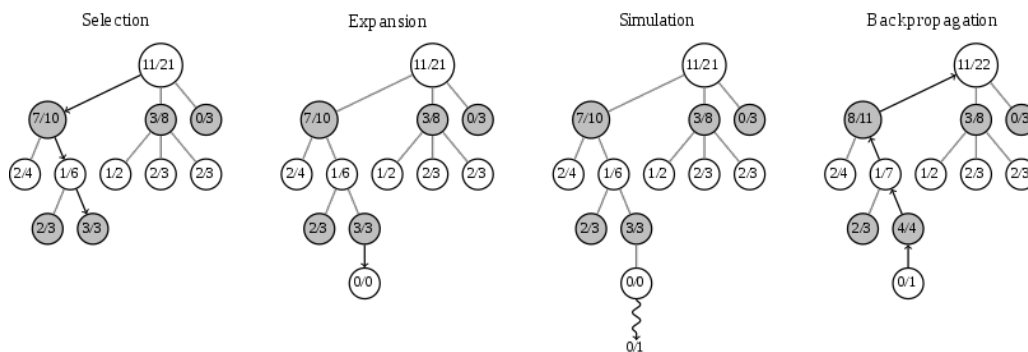
Joueur Monte-Carlo L'algorithme de Monte-Carlo est un algorithme probabiliste qui vise à donner une approximation d'un résultat trop complexe à calculer : il s'agit d'échantillonner aléatoirement et de manière uniforme l'espace des possibilités et de rendre comme résultat la moyenne des expériences. Dans le cas d'un jeu tel que le morpion, 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 get_action(state):
    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. Observer comme précédemment l'espérance de gain dans chaque situation.

3 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 complètement 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 a 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 feuille 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 effectuée 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).

4 Bonus : Puissance 4

Implémenter un puissance 4 et tester vos algorithmes.