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 computationelle). 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, un gain (ou aucun) est attribué au joueur. Supposons une machine à sous à N leviers, l'ensemble $\mathcal{A}$. 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. Lorsque le joueur exécute l'action, il reçoit un gain aléatoire r_t tiré de la distribution de gain associée au levier a_t. Le i-ème levier a une espérance de gain notée μ^i (le gain moyen obtenu au cours d'un grand nombre de parties), inconnu du joueur. On suppose que le rendement de chaque levier est stationnaire dans le temps (les μ^i sont constants).
Le gain au bout de T parties est donc $G_T = \sum_{t=0}^{T} r_t$ et le but est bien sûr de maximiser ce gain. Pour cela, il faut que le joueur identifie la machine au rendement le plus élevé. Soit $\mu^* = max(\{\mu^i\}_{i=1}^N)$, le gain maximal du joueur au temps T est donc $G_T^* = \sum_{i=1}^{T} r_t^*$, avec r_t^* le gain obtenu lorsque l'on joue la machine de rendement maximal au temps t (n.b. : le gain est aléatoire, donc ce gain est une variable aléatoire). On appelle regret au temps T la différence entre le gain maximal espéré et le gain du joueur : $L_T = G^* - \sum_{t=1}^{T} r_t$. L'objectif est donc de minimiser ce regret.
Nous noterons par la suite :

- $N_T(a)$ le nombre de fois où l'action (la machine) a a été choisie jusqu'au temps T.
- $\hat{\mu}_T^a = \frac{1}{N_T(a)} \sum_{t=1}^{T} r_t \mathbf{1}_{a_t=a}$ le gain moyen estimé pour l'action 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 : $a_t = argmax_a \hat{\mu}_t^a$. Cet algorithme est purement d'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* : $a_t = argmax_a \hat{\mu}_t^a$. Cet algorithme explore continuellement.
- **l'algorithme UCB** [1] : l'action choisie est $a_t = argmax_a \hat{\mu_t}^a + \sqrt{\frac{2log(t)}{N_t(a)}}$. 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 garantie l'exploration.

Expériences

Nous allons considérer dans la suite des leviers qui suivent une distribution de Bernoulli : le gain sera de 1 avec une probabilité de μ^i pour le levier i et de 0 sinon. Une machine à sous à n leviers peut donc être représentée par une liste à n réels compris entre 0 et 1, la liste des rendements de chaque levier.

1. Upper Confidence Bound

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 gains par levier et le deuxième le nombre de fois où chaque levier a été joué.
Vous comparerez les différents algorithmes en faisant varier en particulier le nombre de leviers, la distribution des rendements, les écarts entre les différents rendements des leviers. Penser à tracer le regret. 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.
Télécharger le code fourni. Il comprend 4 classes :

- `State` : représente l'état d'un jeu de plateau générique.
- `MorpionState` : représente l'état d'un jeu de morpion
- `Jeu` : permet à partir d'un état et de deux joueurs de faire jouer la partie.
- `Agent` : un joueur abstrait.

Un état correspond a une situation donnée du jeu (dans notre cas, un certain nombre de cases marquées et la donnée du joueur courant). Lorsqu'un des deux joueurs joue un coup, la méthode `next(coup)` permet d'obtenir l'état suivant du jeu. Le mécanisme du jeu vous est entièrement fourni. A chaque état, il est possible de connaître l'ensemble des coups possibles avec la méthode `get_actions()` de l'état.

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'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

A télécharger la semaine prochaine.

4 Bonus : Puissance 4