3I005 : Statistique et Informatique

Projet :

Exploration/Exploitation

Par :
Angelo ORTIZ
Celina HANOUTI

Licence d'Informatique
Année 2018/2019

Table des matières

Table des figures

Liste des tableaux

Introduction

En intelligence artificielle, plus précisément en Machine Learning, on est souvent amené, pour un ensemble de choix possibles, à trouver un bon compromis entre l'exploration et l'exploitation. L'exploration consiste a obtenir d'avantage d'informations dans le but de prendre de meilleures decisions dans le future. L'exploitation, quant à elle, consiste à agir de manière optimale en fonction de ce qui est déjà connu. Ce dilemme constitue un problème largement étudié dans le domaine de l'apprentissage par renforcement. En effet, Des algorithmes ont été proposés pour résoudre un certain nombre de problèmes qui illustrent des applications de ce dilemme.

Dans ce projet, nous allons étudier et analyser un certain nombre d'algorithmes basés sur le dilemme de *l'exploration vs exploitation* à travers des IAs de jeu. Dans un premier temps, nous allons effectuer une evaluation experimentale d'algorithmes classiques dans un cadre simplifié, la deuxèime partie est consacrée à l'implementation d'un algorithme de Monte-Carlo et enfin, dans les deux dernieres partie, nous allons étudier des algorithmes plus avancés ainsi qu'un jeu de stratégie combinatoire.

Première partie

Bandits-manchots

1 Description du problème

On condière le problème d'apprentissage suivant : un agent est confronté à plusieurs reprises à un choix parmi N différentes actions, chacune des N actions procure une récompense moyenne μ^i inconnue par l'agent et nous désignons l'action sélectionnée sur le pas de temps t par a_t, la récompense correspondante par r_t et $N_T(a)$ le nombre de fois ou l'action a a été choisi jusqu'au temps T. L'objectif est de maximiser la somme des récompense obtenue au bout des T premieres parties, pour celà, l'agent doit identifier l'action au rendement le plus elevé i^*= $\text{argmax}_{i\in 1,\dots,N}\,\mu^i$. Les valeurs μ^i etant inconnues, il est donc necessaire de faire des estimations qui doivent être le plus représentatives possible des valeurs μ^i, notons $\hat{\mu}^i$ ces estimations. L'estimation associée a une action est la récompense moyenne lorsque cette action est sélectionnée.

$$\hat{\mu}^i = \frac{1}{N_T(a)} \sum_{t=1}^{T} r_t \mathbb{1}_{a_t=a}$$

2 Algorithmes

Si de nombreux algorithmes ont été proposés pour résoudre ce dilemme, on se contentera ici d'en citer que quelques uns :

2.1 Algorithme Aléatoire

l'algorithme aléatoire se contente de choisir l'action uniformément au hasard. On peut donc déjà présager que cet algorithme sera forcément le moins optimal. **En effet, il effectue purement de l'exploitation et est ainsi utilisé comme baseline pour les tests présentés par la suite**

2.2 Algorithme Greedy

L'algorithme Greedy est basé sur une politique de selection très simple qui consiste à sélectionner l'une des actions ayant la valeur estimée la plus élevée : a_t = $\text{argmax}_{i\in 1,\dots,N}\,\hat{\mu}_t^i$. Autrement dit, L'algorithme Greedy ne fait qu'éxploiter les connaissances dont on dispose afin de maximiser la recompense à un instant t.

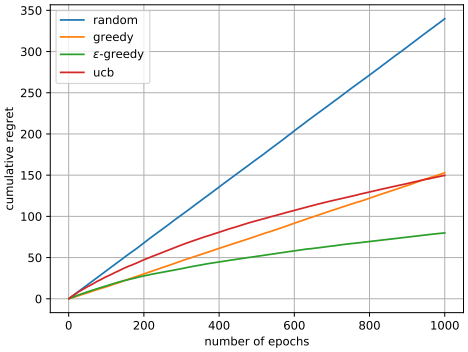
2.3 Algorithme ε-Greedy

Une approche courante pour trouver un compromis exploitation/exploration est l'algorithme ε-Greedy qui consiste a choisir a l'instant t, avec une probabilité 1 - ε, l'action qui maximise le rendement moyen sur les estimations faites jusque la et avec une probabilite ε, une action uniformement au hasard. L'avantage de cet algorithme repose sur le fait que dans la mesure où le nombre d'étape d'apprentissage augmente, chaque action sera choisie un nombre infini de fois assurant ainsi que tous les $\hat{\mu}^i$ convergeront vers les μ^i. On peut conjécturer la puissance de cet algorithme et sa capacité a explorer toutes les actions possibles et de prendre des décisions quasi proches de l'optimum. **Je sais pas si c'est vraiment un nombre infini, mais c'est sûr que les estimateurs se rapprocheront davantage aux gains moyens réels. On pourra aussi dire que la valeur de ε a un fort impact sur l'apprentissage, c'est pourquoi on en a testé différentes valeurs (constantes) et même une forme en fonction (décroissante) du temps ? ? ?**

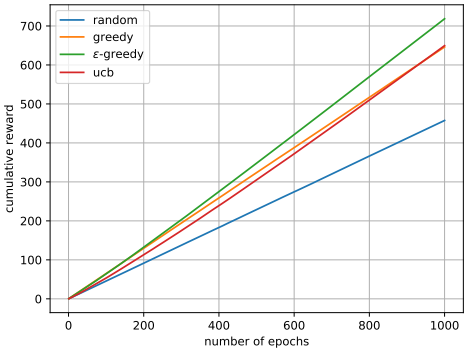
2.4 Algorithme UCB

L'algorithme UCB est une autre stratégie qui permet de trouver une balance entre l'exploration et l'exploitation. L'idée de cet algorithme consiste à se fier à une borne supérieure de confiance, en effet, une incertitude existera toujours sur l'optimalité des estimations faites jusque là. Les deux algorithmes définis précedemment choisissent toujours l'action qui semble meileure à un instant t ou se contentent de choisir des actions aléatoirement sans distinction. L'algorithme UCB prend en compte l'optimalité des estimations mais egalement l'incertitude sur celles-ci. Il selectionne l'action : $a_t = \mathrm{argmax}_{i \in 1,\ldots,N}(\hat{\mu}_t^i + \sqrt{\frac{2\log(t)}{N_T(i)}})$. **Ici, cette incertitude est mesurée par le deuxième terme : plus grand est le déséquilibre entre le(s) (nombre d') actions effectuées, plus on se méfie au résultats. On priviliégiera ainsi les actions le plus défavorisées**

3 Mise en oeuvre et expérimentations



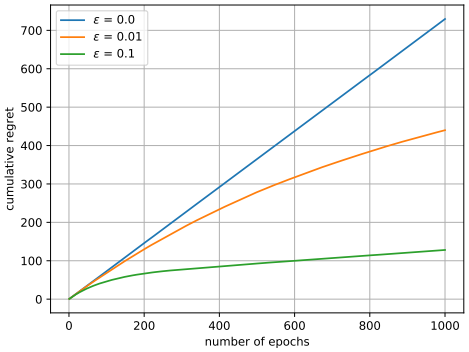
(a) Regret cumulé en fonction du nombre d'étape d'apprentissage



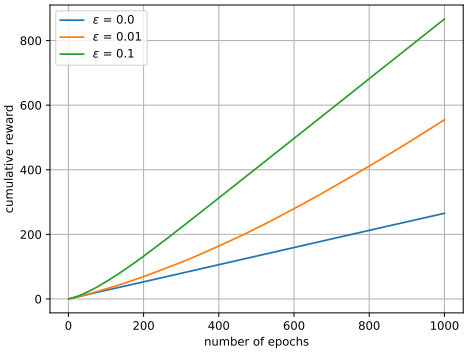
(b) Recompenses cumulés en fonction du nombre d'étape d'apprentissage

FIGURE 1 – Performance des quatres algorithmes sur 10 leviers et en moyennant les résultats sur 400 éxecutions.

La figure 1 montre le gain et le regret cumulé en fonction de t pour les quatre algorithmes décrits dans la section précédente. Nous avons fait le choix de simuler avec une machine à 10 léviers, chacun suit une loi de Bernoulli avec un paramètre choisi uniformément dans [0,1]. TODO : commenter les courbes ci-dessus. Il me semble qu'on est sensé remarquer une amelioration si on change la distribution de gain. En effet, si on a des gains qui suivent une distribution normale plutot qu'une distribution uniforme aléatoire, on remarquera une legere amelioration des performance pour chaque algorithme. Cela est logique, car l'action optimale est mieux "s"paré" des autres actions (ici il serait plus judicieux de favoriser l'exploitation plutot que l'exploration)



(a) Regret cumulé en fonction du nombre d'étape d'apprentissage



(b) Recompenses cumulés en fonction du nombre d'étape d'apprentissage

FIGURE 2 – Performance des algorithmes Greedy et ϵ-Greedy

La figure 2 illustre une comparaison entre la méthode $Greedy$ et deux autres méthodes ϵ-$Greedy$ (ϵ=0.01 et ϵ=0.1). On remarque que le regret cumulé et le nombre de récompenses augmentent avec le nombre de pas d'apprentissage, on peut observer également qu'au tout début, l'algorithme $Greedy$ s'améliore en apprentissage au même rythme que les deux autres algorithmes puis on voit une diminution en performance. En effet, la méthode gloutonne effectue uniquement de l'exploitation, autrement dit, elle se contente de choisir l'action la plus prometteuse à un instant donné sans explorer d'autres actions qui pourrait l'être davantage, les echantillonages initiaux qu'elle a effectué sont avérés être sous optimaux par la suite ce qui explique son déclin de performance. La méthode ϵ-$Greedy$ avec ϵ=0.1 a exploré bien plus que la méthode ϵ=0.01 ce qui explique que la plupart du temps elle identifie l'action optimale assez rapidement. La méthode ϵ-$Greedy$ avec ϵ=0.01 s'améliore en apprentissage plus lentement car elle exploite plus mais à un moment donnée, on est sensé observer un gain de performance de tel sorte qu'elle donne de meilleurs résultats que ϵ-$Greedy$ avec ϵ=0.1.
TODO : il serait intéressant de changer la dispersion des valeurs des récompenses, il me semble que si elles sont trop dispersées c'est à dire avec une variance non négligeable, il nous faudra plus d'exploration pour trouver l'action optimale et dans ce cas ϵ-$Greedy$ sera plus pérformante que Greedy. mais si on a une variance assez faible, l'algorithme Greedy sera plus performant car il trouvera l'action optimale rapidement et ne fera plus d'exploration. à faire : courbe de l'evolution du pourcentage de gain afin qu'on puisse mieux observer le raisonnement précédent

Deuxième partie

Morpion et Monte Carlo

4 Description du problème

Dans cette partie, on considère le célèbre jeu du Tic Tac Toe qui se joue sur une grille 3×3. Les joueurs alternent tour à tour en plaçant un «X» ou un «O» sur une case vide de la grille. Le premier joueur à aligner trois symboles gagne. Nous allons considérer deux types de joueurs, des joueurs *Random* qui se contenteront de choisir leurs prochains coups à jouer aléatoirement et des joueurs qui, eux, utiliseront une méthode de *Monte Carlo*. L'implémentation de ce jeu est basée sur quatre classes : `State` qui représente l'état du jeu, une classe `Agent` qui elle, représente le comportement d'un agent. La méthode `get_action` permet à l'agent de

5 Joueur Aléatoire

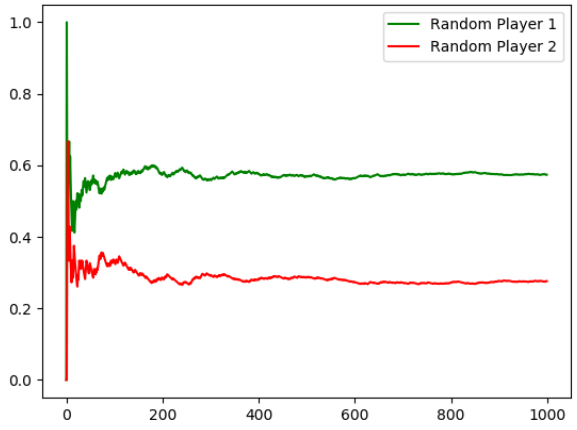


FIGURE 3 – Espérance de gain pour deux joueurs *Random*

La stratégie aléatoire consiste à exploiter que les actions possibles comme information sur le jeu On considère une partie comme étant une expérience à deux issus : victoire ou défaite du Joueur 1. On note X la variable aléatoire qui dénote la victoire du Joueur 1, en cas de victoire, on donne à X la valeur 1, sinon elle prend la valeur 0. Il est évident alors que X suit une loi de Bernoulli, sur la figure ci-dessus, on peut déduire visuellement le paramètre p, en effet, en moyenne, le Joueur 1 a

une probabilité $\mathbb{P}(X=1)$=0.6 de gagner. On en déduit que X suit une loi de Bernoulli de paramètre p=0.6 et de variance $\mathbb{V}(X)= p(1-p)$ = 0.24.

6 Joueur Monte Carlo

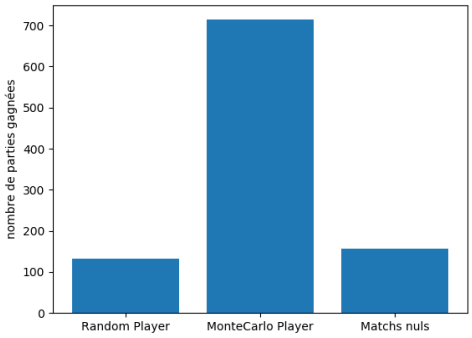
L'idée générale d'une simulation de $Monte\ Carlo$ consiste à jouer un certain nombre de parties avec des choix aléatoires puis d'utiliser les résultats de ces parties pour calculer une bonne action à jouer. Lorsqu'un joueur gagne une de ces parties aléatoires, il devra privilégier les actions qui l'ont mené à cette victoire, dans l'espoir de choisir un coup gagnant lors des prochaines parties et éviter les actions qui ont mené son adversaire à la defaite. Une stratégie $Monte\ Carlo$ échantillonne aléatoirement l'espace de toutes les actions possibles et fait une estimation de l'action qui semble la plus optimale.

6.1 Implémentation

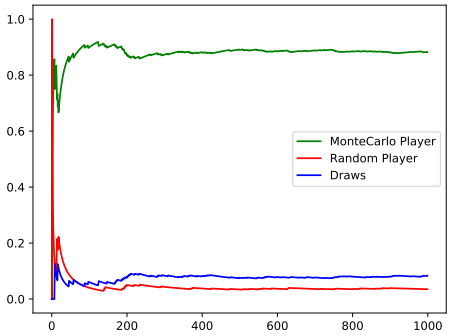
Pour implémenter cet algorithme nous avons créer une classe `MonteCarloAgent` qui hérite de `Agent` et qui implémente la méthode `get_action` dans laquelle on recupère d'abord toutes les actions possibles ensuite, pour N itérations, on choisit une action aléatoirement, on recupère l'état du jeu qui sera joué après cela par deux joueurs aléatoires, enfin, on met à jour la récompense associée à cet action selon le cas où elle a conduit à une victoire ou une défaite. La méthode renvoie l'action avec la plus grande récompense.

6.2 Random VS Monte Carlo

On peut observer sur la figure 4 que le joueur basé sur la méthode de $Monte\ Carlo$ surpasse largement le joueur $Random$ en terme de parties gagnées. Cependant, il existe des situations où gagner est impossible pour le joueur $Monte\ Carlo$, malgré le fait que sa politique de jeu soit plus intelligente que celle du joueur $Random$, par exemple, la défaite est inévitable si c'est le joueur $Random$ qui entame la partie et qu'il marque la case centrale en premier, dans ce cas, beaucoup de voies seront bloquées dans la grille et donc peu importe la stratégie du joueur, cela ne lui évitera pas la défaite.



(a) Statistique sur 1000 parties jouées entre un Joueur *Random* et un Joueur *Monte Carlo*



(b) Espérance de gain pour chaque joueur

FIGURE 4 – Comparaison entre un Joueur *Random* et un Joueur *Monte Carlo*

6.3 Monte Carlo VS Monte Carlo

La figure 5 ci-dessous illustre très bien l'avantage donné au joueur qui entame la partie étant donnée que son espérance de gain est plus elevée que celle du joueur adverse bien qu'ils suivent la meme stratégie de jeu. Cela peut s'expliquer, comme précédemment, par le fait qu'il est presque impossible de gagner pour un joueur lorsqu'il n'entame pas la partie.

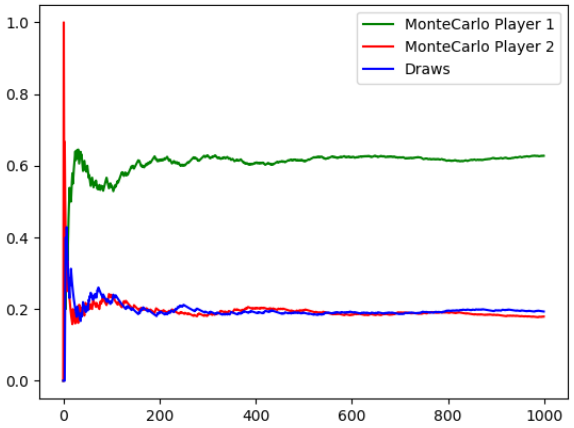


FIGURE 5 – Espérance de gain pour deux joueurs *Monte Carlo*

Troisième partie

Monte Carlo Tree Search

7 Description de l'algorithme

L'algorithme Monte Carlo Tree Search est un algorithme de recherche heuristique qui explore intelligemment l'arbre des actions possibles à partir de l'état courant du jeu et ce, jusqu'à ce que toutes les configurations du jeu soient explorées. Cet algorithme choisit en priorité les branches qui sont le plus susceptibles de mener à une victoire. Il maintient par ailleurs un compromis entre l'exploitation et exploration en utilisant l'algorithme UCB pour choisir la prochaine branche à explorer en priorité.

8 Implémentation

9 Mise en oeuvre