TP bandit à N bras

Sylvain Gault

9 septembre 2024

1 Introduction

Ce TP est à réaliser par trois en Python sous l'environnement de développement de votre choix. Il sera à rendre le Vendredi 13/09/2023 à 13h37 UTC. Il sera à rendre sur Teams dans le devoir *« TP note »*. Vous nommerez vos fichiers du noms des membres du groupe.

Vous rendrez un rapport en **PDF** contenant au moins les réponses aux questions ainsi que toute description nécessaire à la reproduction de vos résultats. **Numérotez les questions** auxquelles vous répondez dans votre rapport et incluez au moins une réponse pour chaque question. Pas de réponse = pas de points. N'oubliez pas de lister **les noms** de tous les membres de votre groupe.

Dans ce TP, vous ferez implémenterez un bandit-manchot à k bras et analyserez les performances de différents algorithmes.

2 Implémenter un bandit manchot

Exercice 1 Le bandit manchot

Le but de cet exercice est d'implémenter un bandit manchot simple.

1. Créez une classe `Bandit` dont le constructeur initialise un attribut `avg` à une valeur aléatoire tirée suivant une distribution gaussienne de moyenne 0 et variance 1.
2. Créez une méthode `play` qui renvoie une valeur aléatoire autour de `self.avg`. Cette valeur devra être tirée aléatoirement avec une distribution gaussienne et une variance de 1.
3. Instanciez votre classe et jouez 1000 fois. Tracez les valeurs sur un graphique sous forme d'histogramme (ou de *« Kernel Density Estimate »*) afin de vérifiez que vous obtenez bien une distribution normale.
4. Instanciez 3 `Bandit` et faites les mêmes graphiques. Vérifiez que vous obtenez bien 3 distributions avec 3 espérances différentes (3 positions de différentes du *« pic »*).

Exercice 2 Le ban-10

Ici, on combine 10 bandit-manchots pour en faire un bandit à 10 bras.

1. Créez une classe `Ban10` qui instancie 10 `Bandit` dans son constructeur et les stocke dans une liste.
2. Rajoutez dans le constructeur un attribut indiquant quelle est le numéro du bandit qui a la plus grande valeur moyenne. Vous pourrez inspecter l'attribut `avg` des `Bandit`s instanciés.

3. Ajoutez une méthode `play` qui prend en argument le numéro du bras à actionner entre 0 et 9 et renvoie la valeur correspondante. Ce sera la récompense associée à l'action donnée.

Exercice 3 Algorithme ε-greedy

Dans cet exercice on implémente l'algorithme de base du k-bandit-machot.

1. Créez une classe `GreedyPlayer` dont le constructeur prend en paramètre n le nombre d'actions (qui sera toujours 10 ici) et `eps` la valeur de ε et les stocke dans des attributs.
2. Initialisez aussi vos tableaux Q et N à 0 dans votre constructeur et stockez-les dans des attrbuts. (Donnez-leur des noms plus pythoniques que Q et N, genre `action_values` et `eval_count`.) Ils contiennent respectivement les estimations et de la valeur associée aux actions et le nombre de fois qu'une action a été choisie.
3. Rajoutez une méthode `get_action` qui ne prend aucun paramètre et qui renverra (dans les questions suivantes) l'action à réaliser sur le `Ban10`. C'est à dire, le numéro de levier à actionner.
4. Dans `get_action` tirez une valeur au hasard entre 0 et 1. Comparez cette valeur à ε afin de définir une variable booléenne `explore`. Elle vaudra `True` si la valeur aléatoire est inférieur à `eps`, `False` sinon.
5. Définissez une méthode `_greedy_action` qui rentourne l'action qui a la plus grande valeur estimée. En cas d'égalité, elle choisira au hasard.
6. Définissez une méthode `_random_action` qui renvoie une action aléatoire.
7. Dans votre méthode `get_action`, choisissez l'action (le bras à tirer) en fonction de `explore`. C'est à dire que si `explore` est `False`, appelez `_greedy_action` sinon, appelez `_random_action`, et renvoyez l'action ainsi récupérée.
8. Ajoutez une méthode `reward` à votre classe `GreedyPlayer`. Celle-ci prendra en argument l'action qui vient d'être évaluée et la récompense obtenue. Elle mettra à jour les valeurs de Q et N pour l'action effectuée et la récompense obtenue.
9. Dans votre code principal créez une instance de `Ban10` et une instance de `GreedyPlayer` pour 10 actions. Appelez `get_action`, évaluez l'action sur le `Ban10` en utilisant sa méthode `play`, et appelez `reward` pour informer votre IA greedy de la récompense obtenue.
10. Mettez le code de la question précédente dans une boucle de 1000 itérations.
11. Testez en affichant pour chaque itération la valeur de la récompense et si oui ou non le choix était optimal.

3 Statistiques

Exercice 4 Graphiques simples

Le but de cet exercice est de faire quelques graphiques afin de montrer l'évolution de la qualité des décisions prises par votre IA.

1. Tracez l'évolution de la récompense au cours des 1000 itérations.
2. Le graphique est très bruité. Créez 2000 instances de `GreedyPlayer` et de `Ban10` dans un tableau. Faites-les toutes jouer étape par étape. C'est à dire que les 2000 instances jouent leur premier coup, puis les 2000 instances jouent leur 2ème coup, etc.

3. Au lieu de tracez la courbe de récompense d'un seul `GreedyPlayer`, tracez la moyenne des 2000.
4. Tracez aussi l'évolution au fil du temps du pourcentage de vos `GreedyPlayer` qui ont fait le choix optimal.
5. Faites de même pour des valeurs d'ε de 0, 0.01, 0.1. Tracez les trois courbes sur le même graphique de récompense moyenne et de pourcentage d'optimalité.
6. Comment interprétez-vous le comportement initial et le comportement asymptotique ?

Exercice 5 Initialisation optimiste

Ici on va comparer l'algo *« ε-greedy »* avec celui qui initialise les estimations des valeurs d'actions à des valeurs élevées.

1. Dérivez la classe `GreedyPlayer` en `OptimistGreedyPlayer`.
2. Définissez un constructeur qui initialise la valeur de Q à 5.
3. Tracez les courbes de moyenne des récompenses et de pourcentage d'optimalité pour comparer l'`OptimistGreedyPlayer` avec le `GreedyPlayer` avec $\varepsilon = 0.1$.
4. Comment interprétez-vous le comportement initial et le comportement asymptotique ?

Exercice 6 Sélection *« Upper-Confidence-Bound »*

Le but de cet exercice est d'implémenter la stratégie UCB de sélection du prochain coup.

1. Dérivez la classe `GreedyPlayer` en `UCBGreedyPlayer`. Elle doit prendre un argument supplémentaire `c` indiquant le degré d'exploration, stockez sa valeur dans un attribut.
2. Surchargez la méthode `_greedy_action` pour sélectionner le meilleur coup selon la formule donnée en cours. Le nombre d'étapes t est donné par le nombre de fois que `reward` a été appelé.
3. Tracez les courbes de moyenne des récompenses et de pourcentage d'optimalité pour comparer l'`UCBGreedyPlayer` avec le `GreedyPlayer` avec $\varepsilon = 0.1$.
4. Comment interprétez-vous le comportement initial et le comportement asymptotique ?

Exercice 7 *(Bonus)* Problèmes non-stationnaires

Ici on va implémenter un système non-stationnaire et tester différentes stratégies.

1. Dérivez la classe `Bandit` en `MovingBandit` et modifiez son comportement pour que la moyenne du bandit bouge doucement à chaque coup. Vous pourrez utiliser la fonction sinus afin de faire osciller la moyenne. Elle devra changer à chaque fois que `play` est appelé. Faites en sorte que la période du sinus soit équivalente à environ 500 coups. Le min et max du sinus seront aléatoires.
2. Dérivez `Ban10` en `MovingBan10` et faites en sorte d'instancier des `MovingBandit` à la place des `Bandit`.
3. Testez différentes valeurs de ε et tracez les graphiques habituels.
4. Testez également l'initialisation optimiste et UCB et tracez les graphiques.

Relisez les modalité de rendu dans l'introduction de ce document et assurez-vous de remplir toutes les conditions. :)