République Tunisienne
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université de Tunis El Manar

École Nationale d'Ingénieurs de Tunis

Département TIC

Projet de fin d'année II

Implémentation d'algorithmes d'apprentissage par renforcement pour le problème MAB (Multi Armed Bandit)

Réalisé par :

Mahmoud Aloulou & Souhail Trimech

Classe : **2 A TEL 2**

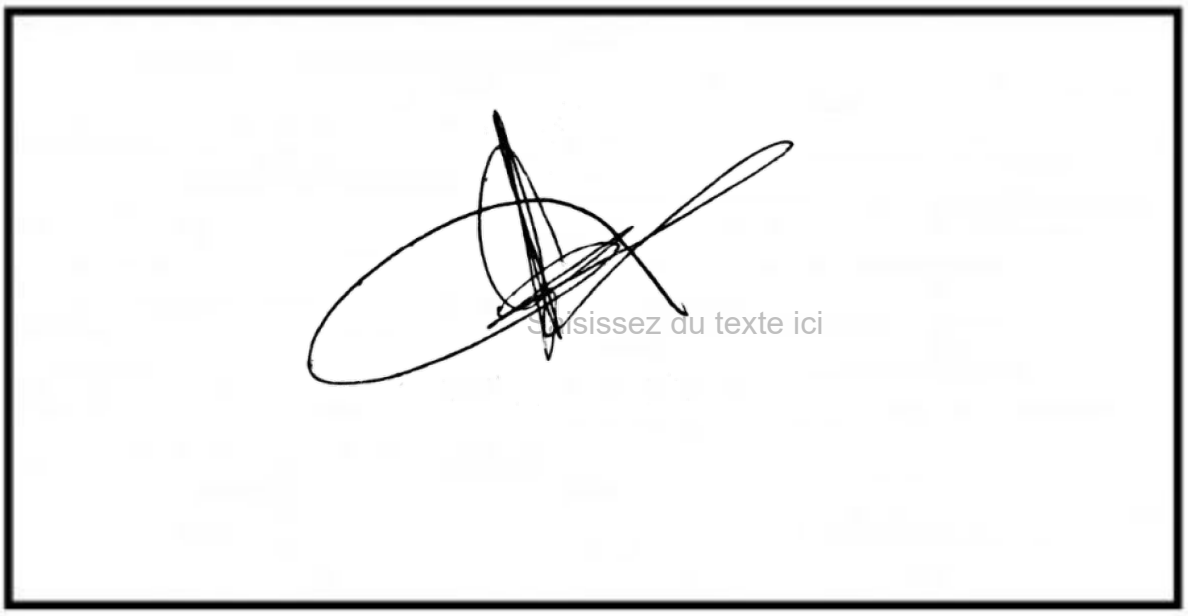
Encadré par :

Mr Mohamed Escheikh

Année universitaire : 2020/2021

Signature

Encadrant : Mr Mohamed Escheikh



Remerciements

Nous tenons tout d'abord à remercier tous les membres du jury qui ont accepté d'évaluer notre projet de fin d'année. Nous remercions également tous ceux qui nous ont aidés dans la réalisation de ce travail et qui nous ont fournis les informations et les renseignements nécessaires pour la réussite de notre projet de fin d'année.
Nous voudrions également exprimer nos sentiments de reconnaissance et de remerciement adressés à notre encadrant Mr Mohamed Escheikh qui nous a guidé et nous a fait bénéficier de son expérience et conseils.
Enfin, nous profitons aussi de cette occasion pour remercier tout le personnel de notre Ecole Nationale d'Ingénieurs de Tunis (ENIT) et spécialement ceux qui font partie du département Technologies de l'Information et de la Communication (TIC) qui ont fait de leur maximum pour nous préserver des bonnes conditions d'études.

Résumé

Le problème MAB pour dire Multi Armed Bandit en anglais ou bandit manchot en français est un cadre théorique important pour étudier le compromis exploration-exploitation dans l'apprentissage par renforcement. Pour un parieur, il s'agit de décider quel bras tirer à partir d'une machine à sous à k bras dans le but de maximiser sa récompense totale après un nombre d'essais. De nos jours, un grand nombre de problèmes d'apprentissage et d'optimisation de notre vie sont modélisées de cette manière dans les différents domaines de santé, finance et autres. Plusieurs algorithmes et méthodes ont été proposées pour résoudre ce problème.
Ce document explique davantage le problème MAB, le dilemme exploration-exploitation et introduit et implémente quelques algorithmes qui résolvent le problème MAB.
Mots clés :
Apprentissage par renforcement, MAB, python, exploration-exploitation.

Table des matières

Table des figures

Liste des acronymes

IA : Intelligence Artificielle

MAB : Multi Armed Bandit

ML : Machine Learning

RL : Reinforcement Learning

UCB : Upper Confidence Bound

Introduction générale

Ce projet intitulé « Implémentation d'algorithmes d'apprentissage par renforcement pour le problème MAB (Multi Armed Bandit) » s'inscrit dans le cadre du module « projet de fin de 2 ème année télécommunications ». Ceci dans le but de nous permettre de s'entrainer à l'écriture d'un rapport, de citer des références et autres : ceci pour se préparer à notre futur projet fin d'études.
Dans ce projet, on introduit tout d'abord dans le chapitre 1 le terme apprentissage par renforcement comme un type d'apprentissage automatique, le dilemme exploration-exploitation et quelques applications du RL. Dans le 2éme chapitre, on introduit le problème MAB et sa relation avec l'apprentissage par renforcement tout en citant des exemples d'applications qui ont été résolues après les avoir modélisées sous forme de problème MAB. Le 3 ème chapitre introduit et explique des algorithmes d'apprentissage pour le problème MAB surtout les algorithmes ϵ-greedy, UCB et Thompson Sampling qu'on a développés et comparés pendant le chapitre 4.

Chapitre 1

Apprentissage par renforcement

1.1 Introduction

Dans ce chapitre, on va tout d'abord définir brièvement l'apprentissage par renforcement. Ensuite, on citera les différents types d'apprentissage tout en détaillant de plus celui de l'apprentissage par renforcement.

1.2 Apprentissage automatique

La définition exacte du terme apprentissage automatique ou Machine Learning en anglais reste non définitive pour plusieurs personnes. On peut évidemment le considérer comme une technologie d'intelligence artificielle permettant aux machines d'apprendre sans être nécessairement implémentées pour ceci. Pour apprendre, ces machines exigent la présence d'un grand nombre de données pour être utilisées durant les phases d'analyse et entrainement ce qui explique la considération de la big data comme essence de l'apprentissage automatique.
Le ML s'agit encore d'une nouvelle science dont le but est de réaliser des prédictions après avoir analysées un grand nombre de données et donc après avoir repérés des patterns et des tendances relatifs aux données[1].
Pour finir, l'apprentissage automatique peut être défini comme une branche de l'IA rassemblant des algorithmes permettant de créer automatiquement des modèles statistiques : c'est-à-dire qu'un système ML apprend à partir de l'expérience et donc sa qualité augmente à chaque fois que l'algorithme ML utilise plus de données[7]. Ce qui précéde n'est pas le cas pour un logiciel ou programme informatique classique qui exécute toujours ce qu'il doit faire en complétant les mêmes instructions avec la même manière[7].

1.3 Les types d'apprentissage

1.3.1 L'apprentissage supervisé

Il s'agit d'un type d'algorithmes de ML ou les données que le modèle de ML utilise pour l'entrainement sont annotées ou étiquetées à l'avance. Le but de ce type d'apprentissage est que le modèle devient capable, après entrainement, de deviner l'annotation convenable à chaque donnée entrée[1, 10]. Par exemple, on peut fournir à un modèle un ensemble de photos de chiens et de loups étiquetées de façon qu'après entrainement le modèle doit différencier entre loups et chiens et donc étiqueter chaque image avec chien ou loup.

1.3.2 L'apprentissage non supervisé

Il s'agit d'un type d'apprentissage ou on entraine le modèle sur des données non étiquetées. Dans ce cas, c'est au modèle d'explorer les données, les comparer et repérer les similarités pour enfin pouvoir rassembler les données qui partagent des caractéristiques communes. On utilise surtout ce type d'apprentissage lorsque on désire mieux apprendre sur un dataset ou jeu de données dont on a l'intention d'utiliser ou même lorsque on désire repérer des comportements relatifs à nos données[1, 10].

1.3.3 L'apprentissage par renforcement

1.3.3.1 Définition et généralités

L'apprentissage par renforcement ou Reinforcement Learning en anglais est un type de Machine Learning qui se base essentiellement sur l'apprentissage suite à l'interaction d'un agent avec son environnement autrement dit suite à un cycle d'expérience : l'agent qui est en train d'apprendre est soit récompensé soit pénalisé à chaque fois qu'il effectue une action. Le but de l'agent et de l'apprentissage par renforcement est alors d'améliorer à chaque itération ses performances ceci en cherchant à maximiser la récompense ou le « Reward ». L'agent doit alors découvrir quelles actions apportent le plus de récompense en suivant un processus d'essai et erreur[10].
La figure ci-dessous représente une idée générale sur le RL et les éléments impliqués dans un modèle de RL dont on parlera de plus dans la section suivante :

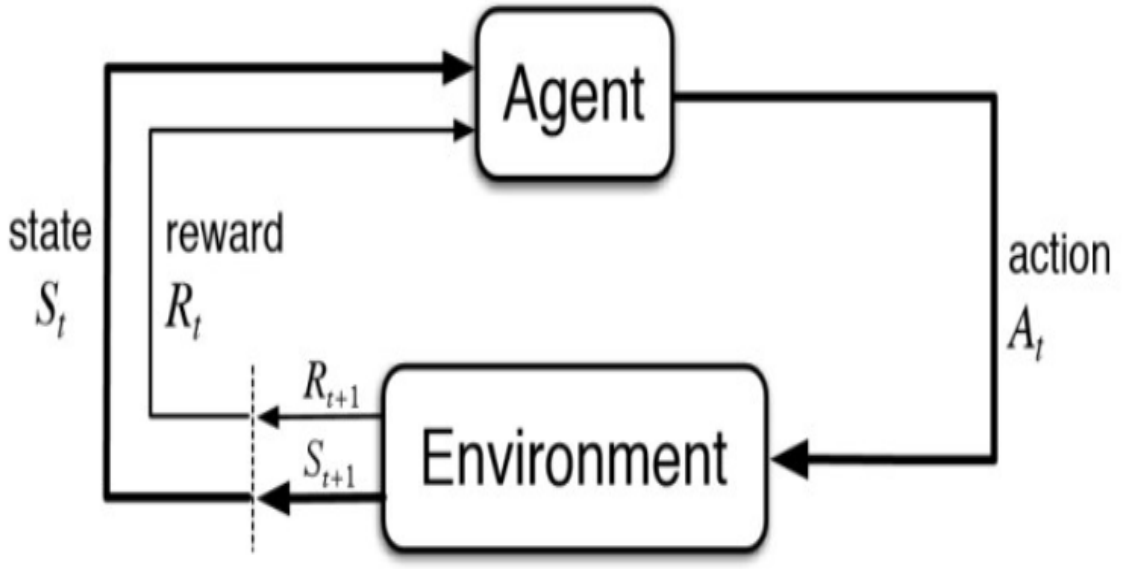


FIGURE 1.1 – Éléments impliqués dans un modèle d'apprentissage par renforcement [2]

L'**environnement** représente l'endroit physique ou l'agent va réaliser ses actions.
Le **State** ou l'état représente la situation actuelle de l'agent.
Le **Reward** ou la récompense représente le retour de l'information suite à une action réalisée.

1.3.3.2 Les éléments de l'apprentissage par renforcement

A part l'agent et l'environnement, on peut repérer 4 éléments fondamentaux d'un système d'apprentissage par renforcement qui sont : une politique, un signal de récompense, une fonction de valeur et un modèle de l'environnement.

• Une **politique** ou policy en anglais est la façon avec laquelle se comporte l'agent dans un état particulier : cette politique peut être résumée dans un tableau ou représentée par une fonction ou encore dépendre du hasard[2, 10].

• Un **signal de récompense** représente l'objectif dans un problème d'apprentissage par renforcement. Suite à chaque état, l'environnement envoie à l'agent un nombre appelé récompense dont l'unique objectif de l'agent est de le maximiser sur le long terme. Le signal de récompense permet de spécifier alors quels sont les bons et les mauvais événements pour l'agent. Dans un système biologique par exemple, on pourrait considérer les récom-

penses comme étant analogues aux expériences de plaisir ou de douleur[10]. Le signal de récompense est la base principale de la modification de la politique ; si une action sélectionnée par la politique est suivie d'une faible récompense, alors la politique peut être modifiée pour sélectionner une autre action dans cette situation à l'avenir[10].

•Alors que le signal de récompense indique ce qui est bon dans un sens immédiat, une **fonction de valeur** spécifie ce qui est bon à long terme. En effet, la valeur d'un état est le montant total de la récompense qu'un agent peut s'attendre à accumuler dans le futur, à partir de cet état. Alors que les récompenses déterminent la désirabilité immédiate des états de l'environnement, les valeurs indiquent la désirabilité à long terme.

Par exemple, un état peut toujours donner une faible récompense immédiate mais avoir une valeur élevée parce qu'il est régulièrement suivi d'autres états qui donnent des récompenses élevées ou l'inverse pourrait être vrai.

Les récompenses sont en quelque sorte primaires, tandis que les valeurs, en tant que prédictions des récompenses, sont secondaires. Cependant, ce sont les valeurs qui nous préoccupent le plus lorsque nous prenons et évaluons des décisions.

Les choix d'action sont faits sur la base de jugements de valeur. Nous recherchons des actions qui apportent des états de plus grande valeur la plus élevée, et non la récompense la plus élevée, car ces actions nous procurent la plus grande récompense à long terme. Malheureusement, il est beaucoup plus difficile de déterminer les valeurs que les récompenses. Les récompenses sont essentiellement données directement par l'environnement, mais les valeurs se modifient en fonction des observations qu'un agent fait au cours de sa vie[10].

•Le **modèle de l'environnement** est quelque chose qui s'inspire du comportement de l'environnement. Par exemple, étant donné un état et une action, le modèle peut prédire l'état suivant et la prochaine récompense. Ces modèles sont utilisés pour la planification, c'est-à-dire ils considèrent les situations futures possibles avant qu'elles ne soient réellement vécues.

1.3.3.3 Le dilemme exploration-exploitation

L'un des choix qu'on rencontre lors d'un problème de RL est le choix entre l'exploitation et l'exploration. En effet, un agent RL en désirant le maximum de récompense doit choisir des actions qu'il a déjà essayés et dont

il a constaté qu'elles apportent une bonne récompense : c'est ça l'exploitation. Cependant, l'agent doit aussi explorer parce qu'il pourrait exister des actions qui apportent plus de récompense que celles qu'il connaisse : ceci dans le but de prendre des meilleures décisions dans le futur. L'agent doit alors essayer les 2 techniques pour enfin favoriser celle qui lui apporte une meilleure récompense. Ce dilemme a été beaucoup étudié par les mathématiciens mais demeure toujours non résolue[1, 10].

1.3.3.4 Applications

L'apprentissage par renforcement a été également utilisé non seulement pour la construction des systèmes d'IA capables de gagner contre les champions du monde des échecs, du jeu chinois Go et de plusieurs autres jeux vidéo mais aussi des robots qui avec interaction apprennent à travailler dans le domaine médical et dans l'industrie [10, 3].

1.4 Conclusion

On a introduit dans ce chapitre une idée générale sur l'apprentissage automatique ou encore plus connu par le Machine Learning, on a parlé ensuite des différents types d'apprentissage tout en donnant plus d'importance à celui par renforcement car c'est ce qui nous concerne le plus durant ce projet.

Chapitre 2

Problème du Multi Armed Bandit

2.1 Introduction

Dans ce chapitre, on va tout d'abord mettre en évidence la relation entre apprentissage par renforcement et MAB abréviation de Multi-Armed Bandit ou problème du bandit manchot. Puis on va formaliser le problème et indiquer l'origine de son appellation pour enfin définir la récompense apportée par une action choisie.

2.2 Relation entre MAB et apprentissage par renforcement

La caractéristique la plus remarquable qui singularise l'apprentissage par renforcement des autres types d'apprentissage est qu'il utilise des anciennes informations d'entrainement qui évaluent les actions déjà prises plutôt que demander l'exécution des actions convenables. C'est ce qui crée le besoin d'une exploration active, d'une recherche explicite par essais et erreurs d'un bon comportement. Le feedback purement évaluatif indique la qualité de l'action prise, mais n'indique pas s'il s'agit de la meilleure ou de la pire action possible. La réaction évaluative est la base des méthodes d'optimisation des fonctions, y compris les méthodes évolutionnaires. Le feedback purement éducatif instructif, par contre, indique l'action correcte à choisir et réaliser indépendamment de l'action réellement prise. Ces deux types de feedback sont tout à fait distincts : la rétroaction évaluative dépend entièrement de l'action entreprise, tandis que la rétroaction instructive est indépendante de l'action entreprise. Il existe également des cas intermédiaires intéressants dans lesquels l'évaluation et l'instruction se mélangent[10, 1]. Dans ce chapitre, nous étudions l'aspect évaluatif de l'apprentissage par renforcement dans un cadre simplifié stationnaire c'est-à-dire. qui n'im-

plique pas d'apprendre à agir dans plus d'une situation. Ce cadre non associatif évite une grande partie de la complexité du problème complet de l'apprentissage par renforcement. L'étude de ce cas nous permettra de voir plus clairement comment la rétroaction évaluative diffère de la rétroaction instructive, tout en pouvant être combinée avec elle. Le problème particulier du feedback évaluatif non associatif que nous allons explorer est une version simple du problème du bandit à n bras[10]. Nous utilisons ce problème pour introduire un certain nombre de méthodes et algorithmes d'apprentissage de base que nous essayerons d'analyser, comprendre et comparer durant ce chapitre et son prédécesseur.

2.3 Formalisation du problème MAB et exemples

Il s'agit d'un problème d'apprentissage par renforcement très simple ou on doit à chaque fois choisir une action parmi un nombre k d'actions possibles pour enfin obtenir une récompense en fonction de notre choix. L'objectif ici est de maximiser la récompense totale après un certain temps c'est-à-dire après 100 actions par exemple[10, 1].
Quant à l'appellation MAB (Multi-Armed Bandit), elle est inspirée de la machine à sous ou bandit à un bras sauf que dans ce problème la machine a k bras ou leviers ou bien on a k machines à un seul bras. De toute façon, le but de l'agent ou de l'utilisateur est presque le même : il s'agit de maximiser les gains en tirant les meilleurs leviers pour gagner le maximum d'argent de la machine à sous[10].
Un exemple d'une application pratique relative au problème MAB est celui d'un site d'actualités qui doit décider quels articles montrer aux visiteurs. En l'absence d'informations sur les visiteurs, tous les résultats des clics sont inconnus. Les questions qui se posent sont quels articles génèrent le plus de clics ? Et dans quel ordre doivent-ils apparaître ? L'objectif du site Web est de maximiser l'engagement, mais il dispose d'un large choix de contenu et manque de données pour l'aider à adopter une stratégie spécifique[8].
Ceci n'est qu' un simple exemple vu que ces dernières années le cadre du MAB a attiré l'attention de divers domaines et applications comme les systèmes de recommandation, la santé, la finance et les télécommunications[3].
En télécommunications, un modèle MAB a été utilisé pour modéliser le problème de la désignation du meilleur réseau sans fil par un dispositif

à technologie d'accès radio multiple(multi-RAT), ceci afin d'améliorer le maximum possible la qualité attendue par l'utilisateur final. Bien que le modèle utilisé soit un élargissement du modèle MAB classique, ce modèle a confirmé la possibilité d'optimiser les performances de la technologie du réseau étendu à longue portée. Les évaluations montrent que de telles méthodes d'apprentissage permettent de gérer le compromis entre la consommation d'énergie et la perte de paquets beaucoup mieux que plusieurs autres algorithmes adaptant les facteurs d'étalement et les puissances d'émission sur la base des valeurs des rapports signal sur interférence et signal sur bruit[3].

2.4 Exploitation vs Exploration

Comme dans tout problème d'apprentissage par renforcement, le problème MAB rencontre le dilemme exploitation-exploration. Choisir une parmi les 2 politiques dépend de la précision des valeurs des estimations des récompenses, du nombre d'essais restants et de plusieurs autres facteurs. De toute façon et d'aprés [10], il est généralement meilleur de balancer entre les 2 politiques que d'utiliser une politique purement exploitatrice : c'est ce qu'on va montrer dans les chapitres suivants.

2.5 Récompense attendue d'une action arbitraire

Considérons maintenant un problème de bandit à k bras ou chacune des k actions a une récompense prévue ou moyenne étant donné que cette action est choisie qu'on note q_* (a). Pour énoncer la formule de cette récompense, on note d'abord t comme le numéro du tir c'est-à-dire la t-éme fois ou l'agent est amené à choisir une action, on note aussi A_t l'action choisie à l'état t et enfin R_t la récompense à l'état t. La récompense prévue lorsqu'une action a est choisie est alors défini par :

$$q_* \text{ (a)}=\text{E}[R_t \mid A_t =\text{a}] \quad .$$

Malheureusement, cette récompense est soit non connue à priori soit connue avec incertitude car sinon la résolution du problème serait simple : on choisirait simplement l'action avec la meilleure valeur de récompense. Dans notre cas, on voudra bien que l'estimation d'une valeur d'une action a soit proche de la valeur attendue de l'action a qu'on vient de définir par q_* (a)[10].

Lors d'un problème MAB, le choix d'une action dépend de la politique exploratrice ou explorative de l'agent :
-Si on choisit l'action qui a déjà apporté la meilleure récompense on est entrain d'exploiter pour maximiser la récompense dans un état. Ce type d'action est appelé greedy en anglais ou gourmand.
-Sinon on choisit l'une des actions incertaines (non greedy) qui peut avoir une meilleure récompense que celle qui possède actuellement la meilleure récompense : on dit qu'on est entrain de faire de l'exploration. Cette politique peut apporter une récompense moins importante à court terme mais plus importante à long terme parce que une fois toutes les actions explorées on peut à ce moment exploiter les actions qui se sont avérées les meilleures[10, 1].

2.6 Conclusion

Ce chapitre nous a permis d'identifier la relation entre apprentissage par renforcement et le problème MAB. Il nous a permis aussi de connaitre l'origine du problème et de citer brièvement quelques applications du problème MAB. Enfin, on a parlé de la récompense engendrée par une action et comment le choix de la politique est important pour une meilleure récompense finale.

Chapitre 3

Quelques algorithmes d'apprentissage pour le problème MAB

3.1 Introduction

Dans ce chapitre on va présenter quelques méthodes d'apprentissage pour le problème de bandit manchot tout en expliquant à chaque fois le principe général de chaque méthode et sa politique de choix d'actions sauf pour l'algorithme de Lai et Robbins.

3.2 Approche gloutonne ou greedy

Appelé aussi méthode de la moyenne simple, elle s'agit d'une méthode ou on fait que de l'exploitation mais pas d'exploration : ainsi la valeur estimée d'une action à l'état t est défini par :

$$Q_t(a) = \frac{somme\ des\ rewards\ fournis\ par\ l'action\ a\ avant\ le\ temps\ t}{nombre\ de\ fois\ l'action\ choisi\ avant\ le\ temps\ t}$$

converge vers q_* (a)[10]. Cette méthode n'est pas nécessairement la meilleure (surtout si la variance de la récompense est élevée) parce que cette approche ne fait que de l'exploitation : d'où l'apparition du méthode ϵ-greedy.

3.3 La méthode ϵ-greedy

L'algorithme $\epsilon - greedy$ est très utilisée parce qu'il est très simple à implémenter. Il s'agit d'un algorithme qui utilise non seulement l'exploitation mais aussi l'exploration. En effet, cette méthode permet à l'agent d'exploiter la plupart de temps en choisissant le bras qui apporte la récompense la

plus élevée avec une probabilité de 1-ϵ et d'explorer avec une probabilité ϵ en choisissant un bras au hasard.
Un avantage de cette méthode est lorsque le nombre de tirs est assez important : toutes les actions possibles sont explorées et tous les q_t (a) qu'on vient de définir convergent vers q_* (a) et ainsi la probabilité de sélectionner la meilleure action converge vers une valeur supérieure à 1-ϵ[1, 5].
En prenant l'exemple suivant : k=10 (10-armed bandit) c'est-à-dire 10 actions possibles, des valeurs d'actions sélectionnées selon une distribution normale de moyenne 0 et variance 1 et les valeurs de récompense selon une distribution normale de moyenne $q_*(A_t)$ et de variance 1 on obtient les graphes suivants qui permettent de comparer l'approche gloutonne avec la méthode $\epsilon - greedy$ pour ϵ=0.1 et ϵ=0.01

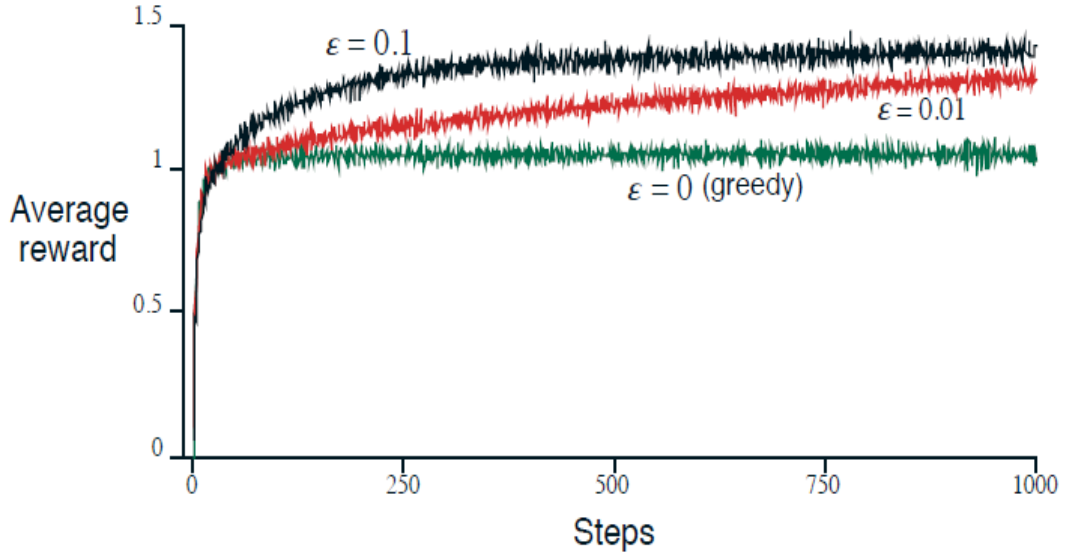


FIGURE 3.1 – Récompense moyenne des différentes méthodes durant 1000 états[10]

Les courbes des figures 3.1 et 3.2 montrent bien que l'approche gloutonne a donné des meilleurs résultats en termes de récompense moyenne et choix de l'action optimal dans les premiers tirs mais pas à long terme comme le cas de la méthode ϵ-greedy qui a mieux fait à long terme parce qu'elle a continué à explorer pour améliorer sa chance à découvrir l'action optimal. Concernant l'approche $\epsilon - greedy$ elle a plus exploré pour ϵ=0.1 que pour ϵ=0.01 vu que ϵ représente le taux d'exploration : ceci explique

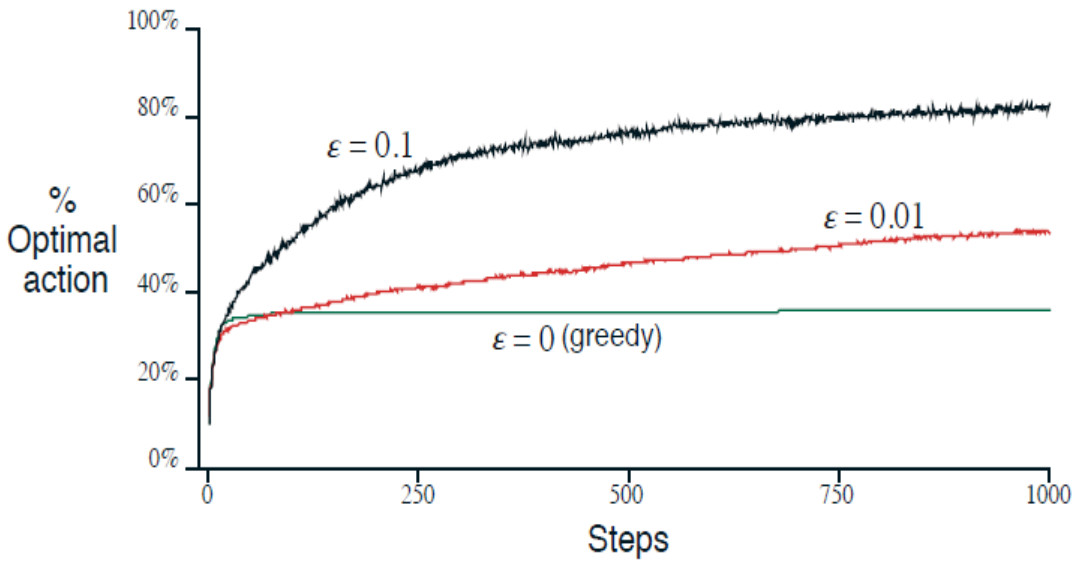


FIGURE 3.2 – Pourcentage du choix du meilleure action pour les différentes méthodes[10]

le fait qu'elle a trouvé l'action optimale plus tôt. L'approche $\epsilon - greedy$ n'est pas toujours meilleure que l'approche gloutonne, en effet si la variance des récompenses était nulle la méthode gloutonne serait capable de détecter la vraie valeur de chaque action après l'avoir essayée une fois et ainsi trouver l'action optimale qu'elle exploitera par la suite sans nécessité d'exploration. Cependant, si la variance de la récompense était 10 au lieu de 1 l'exploration serait très utile et donc l'approche $\epsilon - greedy$ sera plus performante[10].

3.4 Les algorithmes de Lai et Robbins

Herbert Robbins qui était le premier à avoir introduit le problème MAB en 1952 a collaboré avec Leung Lai pour donner des algorithmes d'apprentissage permettant d'obtenir un regret borné par une fonction logarithmique ceci pour des récompenses spécifiques c'est-à-dire pour des distributions de probabilités particulières[5, 8].

Pour commencer, le regret attendu est l'un des outils de mesure les plus performants pour les algorithmes de bandit. Il est défini pour tout tour n par :
r_n= nμ* - E($\sum_{t=1}^{n}$ μ_t) ou μ* est la récompense moyenne du meilleure action ou meilleur bras et μ_t la récompense que l'on obtient après le tour t.

Lai et Robbins à travers leur recherche et calcul avancé, ont réussi à trouver une borne inférieure du regret que tout algorithme de bandit doit encourir pour le qualifier de bon[8]. Pour finir, cette borne est la plus célèbre et elle est nommée la borne de Lai et Robbins et elle est bien démontrée et détaillé dans l'article [6] et elle s'agit principalement de $r_n < O(\log(n))$. Autrement dit, cela signifie que l'action optimale qui apporte la meilleure récompense est jouée exponentiellement plus fréquemment que les autres actions[8].

3.5 L'échantillonnage de Thompson

L'échantillonnage de Thompson ou Thompson Sampling en anglais est la plus ancienne méthode pourtant vue comme la plus performante pour le problème MAB par plusieurs [8]. Elle a été introduite par William R Thompson et elle permet de résoudre le dilemme exploitation-exploration en se basant sur la qualité et l'incertitude.
Dans la suite, on introduit la méthode du Thompson Sampling en critiquant la méthode $\epsilon - greedy$ dans une situation particulière. En effet, on suppose maintenant qu'on est face à un problème de bandits à 3 bras ou à l'état actuel le bras 1 a une récompense moyenne de 0.73 après avoir été tiré 20 fois, le bras 2 a une récompense moyenne de 0.75 après 800 tirages et le bras 3 a une récompense moyenne de 0.8 après 800 tirages. Il est clair donc que le bras 3 est le meilleur actuellement mais si nous devions explorer choisirons-t-on le bras 1 ou 2. Si l'algorithme $\epsilon - greedy$ se met face à cette situation, il choisirait avec une probabilité ϵ équiprobablement l'un des 3 bras. Cependant, choisir le bras 2 apparait comme action exploratrice inefficace vu qu'il a été déjà choisi le même nombre de fois que le bras 3 avec une récompense moyenne inférieure et a été choisi 40 fois autant ou 780 fois de plus que le bras 1 mais n'a pas une récompense moyenne assez grande devant le bras 1 qui a une meilleure récompense moyenne par tour puisqu'il n'a été choisi que 20 fois. Autrement dit, le bras 1 non suffisamment tiré a plus de chances d'être meilleur que le bras 3 que le bras 2. L'algorithme $\epsilon - greedy$, malheureusement, ne favorise pas le bras 1 mais le choisit seulement avec une probabilité de $\epsilon/3$. Quant à l'algorithme de l'échantillonnage de Thompson qui nous intéresse le plus dans cette section, il fait inclure l'incertitude en modélisant le paramètre de Bernoulli du bandit avec une distribution β antérieure. L'avantage de cette méthode

est qu'elle choisit toujours l'action ou le bras dont la récompense attendue est la plus importante, avec la particularité que cette récompense est équilibrée par l'incertitude. Il s'agit effectivement d'une approche bayésienne (le fait de se baser sur des connaissances à priori, les résultats des expériences antérieures et autres facteurs pour mettre à jour les probabilités) du problème MAB. Dans notre configuration de bandit de Bernoulli, chaque action k renvoie une récompense de 1 avec la probabilité θ_k et 0 avec la probabilité 1-θ_k. Au début d'une simulation, chaque θ_k est échantillonné à partir d'une distribution $\theta_k \sim$ Uniform (0,1) avec θ_k constant pour le reste de cette simulation (on va travailler dans le cas stationnaire). L'agent débute avec une croyance préalable de la récompense de chaque bras k avec une distribution beta ou α=β=1. La densité de probabilité antérieure de chaque θ_k est donnée par :

$$p(\Theta_k) = \frac{\Gamma(\alpha_k + \beta_k)}{\Gamma(\alpha_k)\Gamma(\beta_k)} {\Theta_k}^{\alpha_k - 1}(1 - \Theta_k)^{\beta_k - 1}$$

Le choix de l'action est réalisé en échantillonnant en premier lieu la distribution beta puis en choisissant l'action dans la récompense moyenne est la plus élevée :

$$x_t = argmax_k(\hat{\Theta}_k) \ , \ \hat{\Theta}_k \sim \text{beta } (\alpha_k, \beta_k)$$

Finalement, en respectant la règle de Bayes, la distribution postérieure d'une action est mise à jour en fonction de la récompense R_t reçue :

$$(\alpha_k, \beta_k) = (\alpha_k, \beta_k) + (R_t, 1\text{-}R_t)$$

C'est-à-dire que le paramètre α_k est incrémenté de 1 chaque fois que l'action k est choisie sinon c'est le paramètre β_k qui est incrémenté[9].
Pour terminer cette section, cet algorithme a longtemps été utilisé dans plusieurs cas surtout dans le placement des publicités ceci grâce à l'excellente performance de la loi à posteriori qui s'adapte tout seul tout en réussissant à construire des intervalles de confiance adaptés à la géométrie des familles de loi. Il a même été montré que le Thompson Sampling atteint la borne de Lai et Robbins[4, 8].

3.6 L'algorithme UCB

L'algorithme UCB pour dire Upper Confidence Bound en anglais ou borne de confiance supérieure est un algorithme qui se base sur la notion d'optimisme face à l'incertitude[8]. Il est donc considéré comme un algo-

rithme qui encourage l'exploration parce qu'il considère toujours l'existence d'une incertitude sur la précision des estimations des valeurs d'actions. Revenons une autre fois à l'algorithme ϵ-greedy qui est forcé à essayer les actions non greedy (non gourmandes) mais ceci sans différencier entre les actions c'est-à-dire sans préférer ceux qui ont plus de chance d'être meilleur. Cependant, l'algorithme UCB est plus optimiste que la méthode ϵ-greedy puisqu'il sélectionne parmi les actions non greedy celles qui ont plus de chance d'être optimal en tenant compte de toute sorte d'incertitude d'estimation. Le choix de l'action pour cet algorithme se fait de la façon suivante :

$$A_t = \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right]$$

Avec :

- A_t : l'action qu'on va choisir au tour t comme l'on a déjà défini plus tôt
- Ln t : logarithme népérien de t
- N_t(a) : nombre de fois que l'action (a) a été sélectionné avant t
- c>0 : paramètre qui contrôle le degré de l'exploration

Pour N_t(a) =0 le terme entre crochets est maximal et l'action a est choisie : l'algorithme UCB permet alors tout d'abord de tirer les bras qui n'ont pas été encore testés donc les explorer ce qui explique sa caractéristique optimiste.

L'idée de cette politique de choix est que le terme en racine carré mesure l'incertitude ou la variance de l'estimation de la valeur de l'action a : ce terme étant maximal représente une sorte de limite supérieure de la valeur réelle possible de l'action a avec le paramètre c indiquant le niveau de confiance. Cette incertitude est réduite à chaque fois que l'action a est choisie car N_t(a) s'incrémente sachant qu'elle représente le dénominateur du terme[1, 10]. Cependant, cette incertitude augmente à chaque fois que l'action est sélectionné ln t étant une fonction croissante se situant dans le numérateur du terme. Ce qui précède affirme alors que toutes les actions seront tirées mais les actions avec des estimations de valeurs plus petites ou qui ont été choisies un bon nombre de fois seront choisies un peu moins dans le futur ceci parce que la croissance de ln t est moins importante avec l'augmentation de t.

Pour finir, d'après le document [10], l'algorithme UCB pour c=2 a pratique-

ment donné une récompense moyenne meilleure que l'algorithme $\epsilon - greedy$ pour ϵ=0.1 pour un exemple de bandit manchot à 10 bras ce qui affirme la bonne performance de la méthode UCB. Cependant, il est plus difficile que la méthode ϵ-greedy à appliquer pour d'autres problèmes d'apprentissage par renforcement.

3.7 Conclusion

Ce chapitre nous a permis de familiariser avec quelques algorithmes connus et simples du problème MAB puisqu'il existe encore des autres méthodes pour ce problème comme les algorithmes EXP3, POKER, Boltzmann exploration et autres [5, 8, 11] qu'on ne va pas voir dans ce rapport.

Chapitre 4

Réalisation

4.1 Introduction

Après avoir étudié et introduit le problème MAB et quelques algorithmes qui résolvent ce problème, il est maintenant le temps d'implémenter les algorithmes et les tester. C'est pour cela, dans ce chapitre, on va présenter notre environnement de travail matériel et logiciel. Puis, on va présenter les captures du code d'implémentation pour enfin commenter et comparer les différents algorithmes d'apprentissage Thompson Sampling, UCB et ϵ -greedy.

4.2 Environnement de travail

Dans cette partie, on parlera de l'environnement matériel et logiciel qu'on a utilisé dans la réalisation de ce projet.

4.2.1 Environnement matériel

Pour la réalisation de ce projet, on a utilisé un ordinateur portable avec les caractéristiques suivantes :
• **PC portable ASUS**.
• **Système d'exploitation** : Windows 10, 64bits, Processeur x64.
• **Processeur** : Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz.
• **RAM** : 8 Go.

4.2.2 Environnement logiciel et installation

4.2.2.1 Environnement logiciel

Pour implémenter les algorithmes d'apprentissage qu'on a déjà introduit on a utilisé :
•**Anaconda** : il s'agit d'une distribution libre et open source des langages de programmation Python et R appliqué au développement d'applications dédiées à la science des données et à l'apprentissage automatique, qui vise à simplifier la gestion des paquets et de déploiement.
Cette distribution est fonctionnelle sur MacOS, Linux et Microsoft Windows.
•**Python** : Python est un langage de programmation polyvalent qui peut être utilisé pour construire à peu près n'importe quoi. En effet, professionnellement, Python est idéal pour le développement web backend, l'analyse de données, l'intelligence artificielle et le calcul scientifique. De nombreux développeurs ont également utilisé Python pour créer des outils de productivité, des jeux et des applications de bureau. Par exemple, Instagram est construit en Python, Netflix utilise Python pour analyser ses données et optimiser ses recommandations de films et séries télévisées et Google est fortement dépendant de Python puisque ce dernier est parmi le top 3 langages de programmation que les ingénieurs de Google exploitent.
•**Jupyter Notebook** : il s'agit d'un outil populaire parmi les data scientists. Il permet aux équipes de créer et de partager leurs documents, leur code et même des rapports complets, ce qui favorise la productivité et la collaboration. Pour plus expliquer, un « notebook » désigne un document qui contient à la fois du code, des images, des liens et des équations. En raison de cette diversité d'éléments, Jupyter Notebook est l'endroit idéal pour rassembler et analyser le contenu étant donné qu'il peut effectuer une analyse des données en temps réel. Par conséquent, on peut délimiter les différentes parties du code avec les autres éléments pour faciliter la compréhension du code aux autres.

4.2.2.2 Installation et initiation

On a seulement téléchargé Anaconda car il installe automatiquement Python, Jupyter Notebook et d'autres paquets de calcul scientifique et projets de données couramment utilisés. Ce téléchargement a été facile et a été fait via le lien suivant : `https://www.anaconda.com/distribution/`.

Après installation d'Anaconda, on lance le navigateur d'Anaconda et puis on lance Jupyter qui s'agit d'une application web où vous pouvez stocker du code Python, des visualisations, etc. Ceci en cliquant sur « launch » comme le montre la figure 4.1.

FIGURE 4.1 – fenêtre de lancement de Jupyter

Enfin, pour commencer le 1ére notebook, il suffit de cliquer sur « new » puis « python 3 » et commencer à coder,tester et visualiser.

4.3 Implémentation

4.3.1 Mise en situation

Dans la suite, nous allons évaluer l'algorithme d'échantillonage de Thompson en le comparant aux algorithmes ϵ-greedy et UCB en utilisant la notion du regret. Le regret pour le problème du bandit de Bernouilli étant la différence entre la récompense moyenne de l'action optimale et la récompense moyenne de l'action sélectionnée est donné par :

$$\text{regret}_t(\theta) = \max_k \theta_k - \theta_{x_t}$$

avec les paramétres de cette équation étant déjà définis dans la section 5 du dernier chapitre.

4.3.2 Code, expériences et commentaires

4.3.2.1 Code

En premier lieu, nous réalisons les importations nécessaires et le bandit à k bras. La fonction get_reward_regret, qu'on a défini comme le montre la capture de code de la figure 4.2, échantillonne la récompense pour l'action donnée et retourne le regret en se basant sur la véritable meilleure action réellement.

```
import numpy as np
import matplotlib.pyplot as plt
from pdb import set_trace

stationary=True
class Bandit():
  def __init__(self, arm_count):
    """
    Les algos essaient d'apprendre quel bras de bandit est le meilleur pour maximiser la récompense.
   Il le fait en modélisant la distribution des armes de bandit avec un Bêta,
  en supposant que la vraie probabilité de succès d'une arme est distribuée par Bernouilli.
    """
    self.arm_count = arm_count
    self.generate_thetas()
    self.timestep = 0
    global stationary
    self.stationary=stationary

  def generate_thetas(self):
    self.thetas = np.random.uniform(0,1,self.arm_count)

  def get_reward_regret(self, arm): # Renvoie une récompense aléatoire pour l'action du bras.
    """ on Suppose que les actions sont indexées à 0 //// l'argument  arm  est un entier
    """
    self.timestep += 1
    if (self.stationary==False) and (self.timestep%100 == 0) :
      self.generate_thetas()
    # stimuler l'échantillonnage bernoulli
    sim = np.random.uniform(0,1,self.arm_count)
    rewards = (sim<self.thetas).astype(int)
    reward = rewards[arm]
    regret = self.thetas.max() - self.thetas[arm]
    return reward, regret
```

FIGURE 4.2 – Importation des librairies et outils nécessaires et dévelopement de la classe Bandit ainsi que ses méthodes.

Dans la capture de code de la figure 4.3, nous implémentons les deux algorithmes bêta de [9], bien que nous nous concentrions uniquement sur l'algorithme de Thompson Sampling dont on a déjà introduit dans le chapitre précédent. Pour l'algorithme de Bernouilli-greedy, les paramètres de Bernouilli sont les valeurs attendues de la distribution Beta.

Comme nous allons comparer la méthode Thompson Sampling avec les méthodes ϵ-greedy et UCB, nous avons implémenté ces 2 algorithmes en se

```python
class BetaAlgo():
  """ Les algos essaient d'apprendre quel bras de bandit est le meilleur pour maximiser la récompense.
   Il le fait en modélisant la distribution des armes de bandit avec un Bêta,
  en supposant que la vraie probabilité de succès d'une arme est distribuée par Bernouilli. """
  def __init__(self, bandit): # L'argument bandit : la classe de bandit que l'algo essaie de modéliser
    self.bandit = bandit
    self.arm_count = bandit.arm_count
    self.alpha = np.ones(self.arm_count)
    self.beta = np.ones(self.arm_count)

  def get_reward_regret(self, arm):
    reward, regret = self.bandit.get_reward_regret(arm)
    self._update_params(arm, reward)
    return reward, regret

  def _update_params(self, arm, reward):
    self.alpha[arm] += reward
    self.beta[arm] += 1 - reward

class BernGreedy(BetaAlgo):
  def __init__(self, bandit):
    super().__init__(bandit)

  @staticmethod
  def name():
    return 'beta-greedy'

  def get_action(self): # Les paramètres de Bernouilli sont les valeurs attendues du bêta
    theta = self.alpha / (self.alpha + self.beta)
    return theta.argmax()

class BernThompson(BetaAlgo):
  def __init__(self, bandit):
    super().__init__(bandit)

  @staticmethod
  def name():
    return 'thompson'

  def get_action(self): # Les paramètres de Bernouilli sont échantillonnés à partir du bêta
    theta = np.random.beta(self.alpha, self.beta)
    return theta.argmax()
```

FIGURE 4.3 – Implémentation des 2 algorithmes de beta de [9]

basant sur le document [10]. Pour l'algorithme ϵ-greedy on l'a implémenté comme le montre la capture de code de la figure 4.5 : ceci en s'inspirant de la figure 4.4 capturée du document [10] ou Q_n est l'estimation de la valeur d'une action après qu'elle ait été sélectionnée n - 1 fois,R_i est la récompense reçue après la i ème sélection d'une action. Q_n est en d'autres le suivant :

$$Q_n = \frac{R_1 + R_2 + ... + R_{n-1}}{n-1}$$

$$
\begin{aligned}
Q_{n+1} &= \frac{1}{n}\sum_{i=1}^{n} R_i \\
&= \frac{1}{n}\left(R_n + \sum_{i=1}^{n-1} R_i\right) \\
&= \frac{1}{n}\left(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i\right) \\
&= \frac{1}{n}\Big(R_n + (n-1)Q_n\Big) \\
&= \frac{1}{n}\Big(R_n + nQ_n - Q_n\Big) \\
&= Q_n + \frac{1}{n}\Big[R_n - Q_n\Big],
\end{aligned}
$$

FIGURE 4.4 – Mise à jour de Q_n d'une action aprés avoir été choisie la nième fois

Pour l'algorithme UCB, on l'a implémenté comme l'on a déjà introduit dans le dernier chapitre : cette implémentation a été réalisée comme la montre la capture de code de la figure 4.6.

```
epsilon = 0.1
class EpsilonGreedy():

  def __init__(self, bandit):
    global epsilon
    self.epsilon = epsilon
    self.bandit = bandit
    self.arm_count = bandit.arm_count
    self.Q = np.zeros(self.arm_count) # vecteur Q : valeurs des actions
    self.N = np.zeros(self.arm_count) # vecteur N : vecteur du nombre de fois que chaque bras a été tiré

  @staticmethod
  def name():
    return 'epsilon-greedy'

  def get_action(self):
    if np.random.uniform(0,1) > self.epsilon:
      action = self.Q.argmax()
    else:
      action = np.random.randint(0, self.arm_count)
    return action

  def get_reward_regret(self, arm):
    reward, regret = self.bandit.get_reward_regret(arm)
    self._update_params(arm, reward)
    return reward, regret

  def _update_params(self, arm, reward):
    self.N[arm] += 1 # incrémentation du nombre de fois que arm a été tiré
    self.Q[arm] += 1/self.N[arm] * (reward - self.Q[arm]) # mise à jour du valeur d'action Q de arm
```

FIGURE 4.5 – Implémentation de l'algorithme ϵ-greedy

La capture de code de la figure 4.7 montre les fonctions utilisées pour générer le regret moyen pour chaque algorithme en fonction du nombre d'essais. La fonction « simulate » simule l'apprentissage d'un seul algorithme et renvoie les regrets moyens sur un certain nombre d'essais. La fonction « experiment » exécute les simulations sur tous les algorithmes et

```python
ucb_c = 2
class UCB():

  def __init__(self, bandit):
    global ucb_c
    self.ucb_c = ucb_c
    self.bandit = bandit
    self.arm_count = bandit.arm_count
    self.Q = np.zeros(self.arm_count) # valeurs des actions
    self.N = np.zeros(self.arm_count) + 0.0001 # vecteur du nombre de fois que chaque bras a été tiré
    self.timestep = 1
  @staticmethod
  def name(): return 'ucb'
  def get_action(self):
    ln_timestep = np.log(np.full(self.arm_count, self.timestep))
    confidence = self.ucb_c * np.sqrt(ln_timestep/self.N)
    action = np.argmax(self.Q + confidence)
    self.timestep += 1
    return action

  def get_reward_regret(self, arm):
    reward, regret = self.bandit.get_reward_regret(arm)
    self._update_params(arm, reward)
    return reward, regret

  def _update_params(self, arm, reward):
    self.N[arm] += 1  # incrémentation (mise à jour) du nombre de fois que (arm) a été tiré
    self.Q[arm] += 1/self.N[arm] * (reward - self.Q[arm]) # mise à jour du valeur d'action Q de arm
```

FIGURE 4.6 – Implémentation de l'algorithme UCB

trace leurs regrets moyens.

```python
def plot_data(y): # y est un vecteur 1D
  x = np.arange(y.size)
  _ = plt.plot(x, y, 'o')
def multi_plot_data(data, names): # data,names sont des listes de vecteurs
  x = np.arange(data[0].size)
  for i, y in enumerate(data):
    plt.plot(x, y, 'o', markersize=2, label=names[i])
  plt.legend(loc='upper right', prop={'size': 16}, numpoints=10)
  plt.show()
def simulate(simulations, timesteps, arm_count, Algorithm): # Simule l'algorithme sur des époques de 'simulations'
  sum_regrets = np.zeros(timesteps)
  for e in range(simulations):
    bandit = Bandit(arm_count)
    algo = Algorithm(bandit)
    regrets = np.zeros(timesteps)
    for i in range(timesteps):
      action = algo.get_action()
      reward, regret = algo.get_reward_regret(action)
      regrets[i] = regret
    sum_regrets += regrets
  mean_regrets = sum_regrets / simulations
  return mean_regrets
def experiment(arm_count, timesteps=1000, simulations=1000):  #  Configuration standard pour toutes les expériences
 """  Arguments :   timesteps : (int) nombre de pas pour que l'algo apprenne le bandit
                    simulations : (int) nombre d'époques  """
 algos = [EpsilonGreedy, UCB, BernThompson]
 regrets = []
 names = []
 for algo in algos:
    regrets.append(simulate(simulations, timesteps, arm_count, algo))
    names.append(algo.name())
 multi_plot_data(regrets, names)
```

FIGURE 4.7 – Implémentation des fonctions pour la génération des graphes

4.3.2.2 Expériences et commentaires

Expérience1 Dans cette première expérience, nous fixons le nombre de bras à 10, ϵ=0,1 pour l'algorithme ϵ-greedy, et c=2 pour UCB comme l'exemple du document [10] qu'on a mentionné dans la fin de la section relative à l'algorithme UCB. Comme on peut le remarquer dans le graphe de la fi-

gure 4.8, les agents suivant les algorithmes Thompson Sampling et ϵ-greedy convergent rapidement vers une valeur de regret stable après presque seulement 200 pas. Au contraire, le regret de l'agent UCB diminue très lentement par rapport aux deux autres agents : en effet, il continue sa baisse même aux derniers pas. Ceci indique que les agents ϵ-greedy et UCB pourraient profiter d'un ajustement des paramètres contrairement à l'agent qui utilise le Thompson Sampling qui fonctionne bien dès le début.

```
# Expérience 1
arm_count = 10 # nombre de bras/actions
epsilon = 0.1
ucb_c = 2
stationary=True
experiment(arm_count)
```

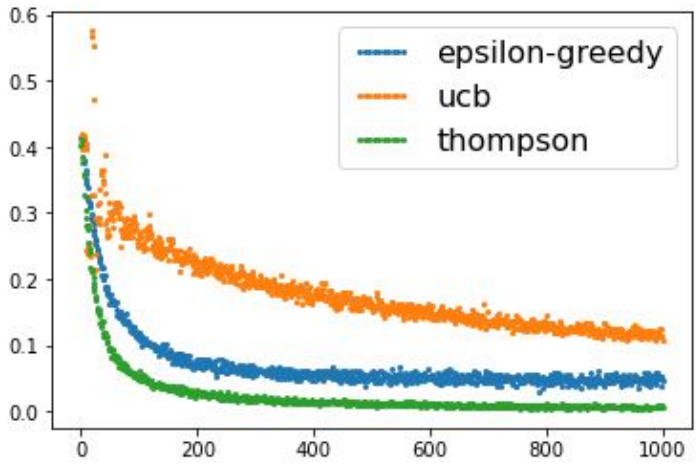


FIGURE 4.8 – regret moyen relatif à l'expérience 1 des 3 méthodes (pour un problème de bandits de **10** bras, ϵ=0,1 et **c**=2) pendant 1000 essais

Expérience2 Dans cette expérience on varie ϵ et c tels que c=0,1 et ϵ=0,03. Après exécution et génération du graphe de la figure 4.9, nous pouvons remarquer que si le paramètre c de l'algorithme UCB est fixé à 0,1, il converge très rapidement vers une valeur de regret faible, après uniquement quelques dizaines de pas et devient maintenant un peu meilleur que ϵ-greedy. Au début, il dépasse même le Thompson Sampling ceci pendant les 400 premiers pas comme le montre le graphe ceci montre que le nombre d'essais est un paramètre utile pour comparer les algorithmes.
Diminuer le c de l'algorithme UCB a principalement diminué la "prime" que l'agent ajoute à l'incertitude. Cela encourage UCB à être plus greedy

ou gourmand, c'est pourquoi il a convergé plus rapidement mais ne surpasse toujours pas le Thompson Sampling.
Pour l'algorithme ϵ-greedy, en diminuant son paramètre à 0,03, il a mieux fait après 1000 pas que lorsque son paramètre est de 0,1, mais ceci seulement de façon marginale. En outre, cela se fait au prix d'une convergence beaucoup plus lente. En effet, nous n'observons pas l'oscillation importante observée avec UCB, même si nous continuons à diminuer ϵ. Il est possible que cette solution de diminuer l'exploration en diminuant ϵ est la meilleure solution parce qu'on a seulement 10 bras ou actions possibles et donc l'exploration peut être considérée comme inutile, ce qui nous pousse à réaliser l'expérience3.

```
# Expérience 2
arm_count = 10 # number of arms in bandit
epsilon = 0.03
ucb_c = 0.1
stationary=True
experiment(arm_count)
```

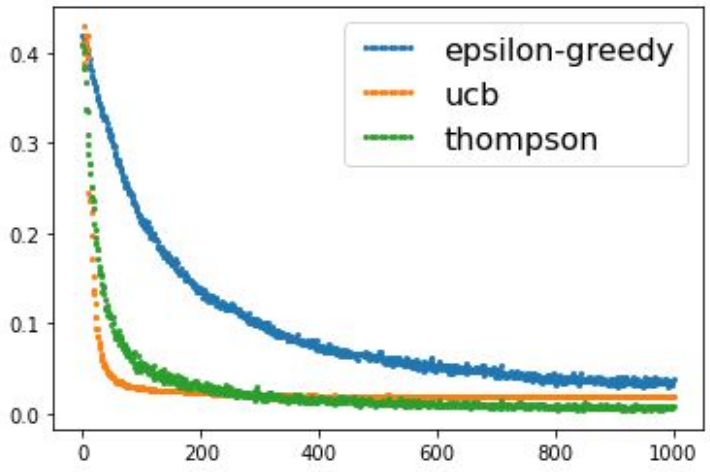


FIGURE 4.9 – regret moyen relatif à l'expérience 2 des 3 méthodes (pour un problème de bandits de **10** bras, ϵ=0,03 et **c**=0,1) pendant 1000 essais

Expérience3 Dans cette 3 -ème expérience, nous augmentons le nombre d'actions de 10 à 200 et on prend ϵ=0,1 et c=2 (comme l'expérience1).
L'agent ϵ-greedy n'évolue presque pas dans son modèle d'apprentissage par rapport à l'expérience 1 parce qu'il va exploiter les meilleures actions la plus part de temps quelque soient le nombre d'actions. L'agent Thompson Sampling, quant à lui, diminue très lentement puisque son regret surpasse l'agent ϵ-greedy à peut prêt à l'étape 700(ceci montre une autre fois que

le nombre d'essais est un facteur important pour choisir le meilleur algorithme. En effet, si on avait droit à seulement 200 essais en jouant à la machine à sous à 200 bras, il serait meilleur d'exploiter majoritairement les actions greedy que d'explorer et gâcher les meilleurs gains), une autre bonne performance pour l'agent Thompson. Cependant, le regret apporté par l'algorithme UCB est instable et divergeant comme le montre le graphe de la figure 4.10 dans la page 27. Ce comportement n'est pas seulement à cause de l'augmentation du nombre de bras ou d'actions, mais au fait que les probabilités de récompense de Bernoulli sont si proches les unes des autres. Pour se rappeler, le regret est calculé comme la différence entre le θ_k optimal et l'action $\hat{\theta}_k$. Comme les vrais θ_k sont échantillonnés à partir d'une distribution uniforme, ils sont uniformément répartis entre 0 et 1. L'algorithme avide c'est à dire greedy n'a besoin d'échantillonner qu'un petit nombre de valeurs et a une forte probabilité que la probabilité de récompense estimée de son action la plus avantageuse soit proche de la vraie probabilité de récompense optimale, ce qui lui garantit un faible regret, même s'il ne réussit jamais à choisir le bras optimal ou l'action optimale. En revanche, les algorithmes Thompson Sampling et UCB, qui s'articulent tous les deux sur l'exploration, accordent une grande dimension et importance à l'exploration des bras(actions) non encore tirés. Puisqu'il y a tant d'actions (200 dans notre cas) et que les récompenses moyennes sont uniformément distribuées entre 0 et 1, le coût de l'exploration des actions ayant une récompense proche de 0 est très élevé. On remarque que l'algorithme de Thompson Sampling est incroyablement et parfaitement stable dans le temps (en le comparant avec l'algorithme UCB toujours en regardant le graphe de la figure 4.10 dans la page 27) malgré le grand nombre d'actions. On peut conclure que l'algorithme Thompson Sampling a une probabilité plus faible que l'UCB de prendre constamment des actions sous-optimales grâce à sa nature d'échantillonnage.

Expérience4 Vu le rendement décevant de l'algorithme UCB dans la dernière expérience, on diminue le paramètre c de 2 à 0.01 pour obtenir le graphe de la figure 4.11 qui montre une augmentation spectaculaire de la stabilité de l'algorithme UCB par rapport à la dernière expérience, qui le permet même à obtenir un regret meilleur que tous les autres algorithmes. ϵ-greedy a aussi meilleur fait que Thompson Sampling pendant les premiers 650 cas et meilleur fait que UCB pendant les premiers 550 cas : ce

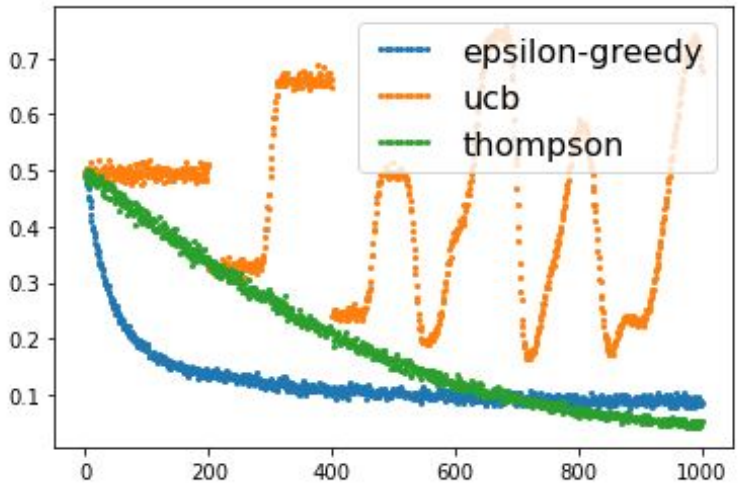


FIGURE 4.10 – regret moyen à l'expérience 3 des 3 méthodes (**200** bras, ϵ=0,1 et **c**=2) pendant 1000 essais

qui montre encore une autre fois l'importance du nombre d'essais pour le choix de l'algorithme. Cependant, l'agent de Thompson semble toujours avoir un regret décroissant vers 1000 pas sans avoir convergé, son regret se situe entre ϵ-greedy et UCB après 1000 pas : on répéte alors la même expérience avec plus de pas.

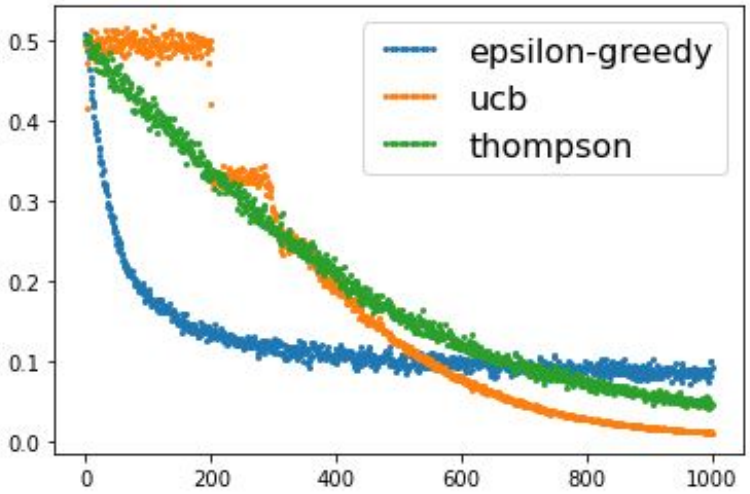


FIGURE 4.11 – regret moyen à l'expérience 4 des 3 méthodes (**200** bras, ϵ=0,1 et **c**=0,01) pendant 1000 essais

Expérience5 On répète l'expérience 4 avec 3000 essais au lieu de 1000 essais pour obtenir le graphe de la figure 4.12 qui confirme que les algorithmes Thompson Sampling et UCB convergent étroitement.

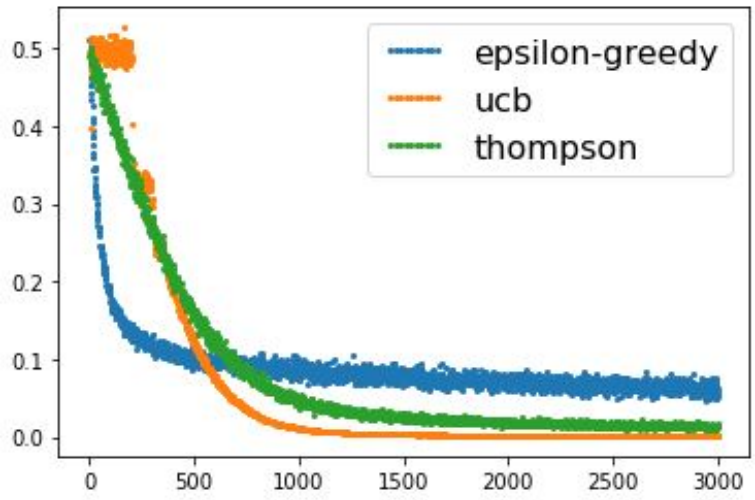


FIGURE 4.12 – regret moyen relatif à l'expérience 5 des 3 méthodes (pour un problème de bandits de **200** bras, ϵ=0,1 et **c**=0,01) pendant 3000 essais

4.3.2.3 Synthése

Les graphes que nous avons pu générer nous ont permis de voir les avantages et les inconvénients des différents 3 algorithmes. L'algorithme de Thompson Sampling a été remarquablement très performant sans qu'il soit nécessaire de régler aucun paramètre. Dans certains cas, UCB a obtenu des résultats proches de ceux de Thompson, mais ceci après avoir ajusté son paramètre c. L'algorithme ϵ-greedy, au contraire, n'a jamais été le plus performant.

4.4 Conclusion

Ce chapitre nous a permis de faire connaissance de la distribution Anaconda et du notebook Jupyter qui nous ont permis d'implémenter les différents algorithmes ϵ-greedy, UCB et Thompson Sampling pour le problème MAB dans le cas stationnaire. Cette implémentation nous a permis de comparer ces 3 algorithmes comme l'on a déjà indiqué dans la sous-sous-section précédente.

Conclusion génerale

Dans ce rapport, nous avons en premier lieu brièvement abordé l'apprentissage automatique pour parler dans la suite de l'apprentissage par renforcement étant donné que le problème MAB est un problème de RL. On a alors défini le RL, les éléments qui constituent un problème de RL et on a cité quelques applications connues du RL. On a aussi parlé de le dilemme exploration-exploitation dont le problème MAB à son état stationnaire étudie facilement grâce à sa simplicité devant les autres problèmes de RL.
On a bien évidemment introduit le problème MAB dans le 2éme chapitre. Dans les chapitres 3 et 4 on a introduit quelques algorithmes d'apprentissage pour ce problème pour implémenter dans la suite le Thompson Sampling, ϵ-greedy et UCB afin de les comparer en fonction du nombre de bras et les paramètres relatifs à chaque algorithme. On a bien conclu à la fin que pour un nombre d'essais élevé, le Thompson Sampling est presque toujours le meilleur pour le problème MAB dans le cas stationnaire qui s'agit d'un problème d'apprentissage simplifié.
Ces dernières années, non seulement plusieurs nouvelles méthodes de résolution du problème MAB sont proposées mais aussi des anciennes se sont plus poussées et améliorées. Ceci pour la simple raison que le problème MAB attire de plus en plus l'attention des diverses applications dans les différents domaines.

Bibliographie

[1] Robin Allesiardo. "Bandits Manchots sur Flux de Données Non Stationnaires". Thèse de doct. Université Paris-Saclay, 2016.

[2] Shweta Bhatt. *5 Things You Need to Know about Reinforcement Learning*. URL : `https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html`. (accessed : 07.03.2021).

[3] Djallel Bouneffouf et Irina Rish. "A survey on practical applications of multi-armed and contextual bandits". In : *arXiv preprint arXiv :1904.10040* (2019).

[4] Emilie Kaufmann, Nathaniel Korda et Rémi Munos. "Thompson sampling : An asymptotically optimal finite-time analysis". In : *International conference on algorithmic learning theory*. Springer. 2012, p. 199-213.

[5] Volodymyr Kuleshov et Doina Precup. "Algorithms for multi-armed bandit problems". In : *arXiv preprint arXiv :1402.6028* (2014).

[6] Tze Leung Lai et Herbert Robbins. "Asymptotically efficient adaptive allocation rules". In : *Advances in applied mathematics* 6.1 (1985), p. 4-22.

[7] lebigdataWebsite. *machine-learning-et-big-data*. URL : `https://www.lebigdata.fr/machine-learning-et-big-data`. (accessed : 02.05.2021).

[8] Jonathan Louëdec et al. "Systemes de recommandations : algorithmes de bandits et évaluation expérimentale". In : *47emes Journees de Statistique de la SFdS (JDS 2015)*. 2015, pp-1.

[9] Daniel Russo et al. "A tutorial on thompson sampling". In : *arXiv preprint arXiv :1707.02038* (2017).

[10] Richard S Sutton et Andrew G Barto. *Reinforcement learning : An introduction*. 2011.

[11] Joannes Vermorel et Mehryar Mohri. "Multi-armed bandit algorithms and empirical evaluation". In : *European conference on machine learning*. Springer. 2005, p. 437-448.