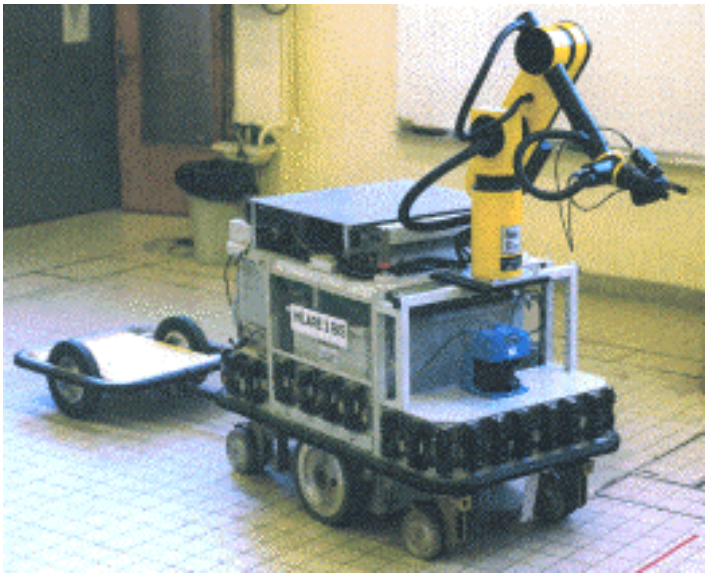
Apprentissage par renforcement

(Reinforcement Learning (RL))

Approche : Monte Carlo

T. AL-ANI
A²SI-ESIEE-PARIS



SOMMAIRE

Partie I, L'apprentissage par renforcement, les méthodes de Monte Carlo

1. *On-Policy Monte Carlo Control*

a) L'algorithme de la '*On-Policy' Monte Carlo Méthode*

Le principe de cet algorithme est de calculer la fonction politique qui, pour chaque couple (état s, action a), optimise le gain obtenue par la fonction de gratification du modèle (Reward (s, a)).

Calcul de la fonction Politique :

- Initialisation de l'algorithme et définition des termes :
  - Soit Q (s, a) la fonction d'évaluation. A l'initialisation Q (s, a) = 0 (arbitraire).
  - Soit Returns (s, a) la fonction mémoire de toutes les gratifications obtenues pour chaque couple (état s, action a). A l'initialisation Returns (s, a) est une liste vide.
  - Soit Reward (s, a) la fonction de gratification pour un couple (état s, action a).
  - Soit π (s, a) une fonction politique arbitraire de type "soft-policy" tel que : $0 \leq \pi (s, a) \leq 1$
  - Soit les termes suivants :
    "a*" représente la ou les actions qui maximisent la fonction d'évaluation Q (s, a)
    "a" les actions qui ne maximisent pas la fonction d'évaluation Q (s, a)
    "ϵ" la pondération des actions 'a' avec $0 \leq \epsilon \leq 1$ (fixé au départ)
- Déroulement de l'algorithme :

On-Policy Monte Carlo Méthode

Répéter à l'infini :

(a) Générer un épisode (partie) en utilisant la politique π (s, a)

(b) Pour chaque paire (s, a) de l'épisode, faire :
R = Gratification de la première occurrence de (s, a)
Ajouter R à la mémoire Returns (s, a)
Q(s, a) = moyenne (Returns (s, a))

(c) Pour chaque état 's' de l'épisode, faire :
$a^* = \text{argmax}_a Q(s, a)$
Pour tout a $\in$ A(s)
$\pi (s, a) = 1 - \epsilon + \epsilon / |A(s)|$ si $a = a^*$
$\pi (s, a) = \epsilon / |A(s)|$ si $a \neq a^*$

b) Application au jeu du Black-jack

❖ Les règles du Black-jack :

Le jeu du Black-jack se joue avec 2 joueurs : la Banque et le Joueur.

Au début de la partie le Joueur tire 2 cartes. La Banque tire aussi 2 cartes mais cache la première au Joueur. Le joueur peut ensuite tirer autant de carte qu'il souhaite, puis s'arrête sans jamais dépasser un score de 21, sinon il perd. Un as peut être égale à 1 ou 11. Une figure est égale à 10. Le score est la somme des cartes du joueur. Lorsque le joueur s'arrête, c'est à la Banque de tirer autant de carte quelle souhaite, puis s'arrête. La partie est finie. Le gagnant est défini selon les règles suivantes :

- Le Joueur gagne la partie si son score est supérieur au score de la Banque ou si le score de la Banque est supérieur à 21.
- Le joueur perd la partie si son score est inférieur au score de la Banque ou si son score est supérieur à 21.
- Si le joueur et la Banque on même score, il y a égalité.

❖ La Fonction de gratification de l'algorithme:

Le jeu du Black-jack peut-être utiliser comme modèle pour un algorithme de Monte Carlo dont on cherche la meilleur politique c'est-à-dire la meilleure façon de jouer pour gagner une partie. Dans cet exemple, générer un épisode équivaut à jouer à une partie de Black-jack. Une des fonctions de gratification possible peut-être :

Fonction de gratification (Reward (s, a)) :

Reward (s, a) = 1 si la partie est gagnée
Reward (s, a) = 0 si la partie continue ou si il y a égalité
Reward (s, a) = -1 si la partie est perdue

avec :
s = score du joueur (le total de ses cartes)
a = entier représentant une action à effectuer
(a = 1 le joueur s'arrête, a = 2 le joueur tire une carte)

On peut qualifié ce modèle de simple car les dimensions des états "s" et des actions "a" sont de dimension 1. Par la suite nous étudierons un modèle plus complexe comportant un couple d'état s. Ce modèle aura une complexité de dimension 2.

- Exemple de partie ou d'épisode générer utilisant ce modèle :

Soit le tirage de cartes suivant:

La banque tire un 8 et 9. Son score est de 17.
Le joueur tire un 3 et Roi. Son score est de 13.

Soit la partie ou épisode généré suivant:

Tour n°1 :
L'état initial "s" de la partie ou de l'épisode = score du joueur = 13.
La politique π (s, a) choisie l'action 2 (tirer une carte).
La nouvelle carte tirée est 2. L'état suivant est donc 13 + 2 = 15. La partie continue.

Tour n°2 :
L'état initial "s" du tour est 15.
La politique π (s, a) choisie ensuite l'action 1 (s'arrêter).
La partie est finie.

- Apprentissage de l'épisode par l'algorithme :

A l'initialisation de l'apprentissage, $\epsilon = 0.1$, et quelque soit s et a, Q (s, a) = 0 et π(s, a) = 0,5.

Pour le premier tour de la partie, le couple (état 13, action 2) permet d'obtenir la gratification qui lui est associé. Le gratification R_1 avec le tirage de la carte 2 est donc Reward (13, 2) = 0 (la partie continue). R_1 est ensuite ajouté à la fonction mémoire (Returns (13, 2) = R_1 = 0). Q (13, 2) et π (13, 2) sont calculés.
Q (13, 2) = 0 (inchangé)
π (13, 2) = 0,95 (augmente) (l'action 2 est une action qui maximise Q(13, a), quelque soit "a")

Pour le second tour de la partie, le joueur a choisi l'action 1 (s'arrêter). Le couple (état 15, action 1) permet d'obtenir la gratification R_2 = Reward (15, 1) = -1 (la partie est perdue). R_2 est ensuite ajouté à la fonction mémoire (Returns (15, 1) = R_2 = -1). Q (15, 1) et π (15, 1) sont calculés.
Q (15, 1) = -1 (diminue)
π (15, 1) = 0,05 (diminue) (l'action 1 ne maximise pas Q(13, a), quelque soit "a")

On peut dire que pour cette première partie, l'algorithme a apprit à choisir une carte dans l'état 13 et à ne plus s'arrêter à l'état 15.

c) Problèmes et conseils d'implémentation sous Scilab

Les dimensions des états et des actions :

Les actions "a" peuvent être des couples d'actions à n dimensions et les états "s" peuvent être des couples d'états à y dimensions. Cela implique l'impossibilité d'utiliser des boucles sur les matrices ne connaissant pas sa dimension selon le modèle choisi. De plus, l'implémentation sous Scilab aurait été très difficile. Pour contourner ce problème nous avons choisi de linéariser les états "s"et les action "a" afin d'obtenir des matrices Q (s, a) et π (s, a) à 2 dimensions. Il a donc été nécessaire d'écrire des fonctions de linéarisation et délinéarisation de matrices et de coordonnés dans le programme.
La fonction mémoire est un tableau à x + y + 1 dimensions

La taille de fonction mémoire Returns (s, a) :

La taille des dimensions de la fonction mémoire doit être dynamique. En effet, on ne connaît pas le nombre de gratification à sauvegarder pour chaque couple (s, a). Cette taille inconnue limite l'algorithme avec l'utilisation des fonctions hypermatrices pour une fonction mémoire. De plus, l'algorithme ralenti avec le nombre d'élément stocké dans cette fonction. Afin de contourner le problème, on peut effectuer une moyenne pour "n" gratifications de (s, a) en conservant la valeur de "n" ainsi que la moyenne des "n-1" gratifications. Ainsi, on calcul la moyenne des n gratifications tel que :

$$Moyenne_{(n)} = (Moyenne_{(n-1)} + R_{(n)}) / n$$

La dissociation du modèle et de l'algorithme :

Lors de l'implémentation sous Scilab, il est important d'écrire le modèle indépendamment de l'algorithme. En effet, il intéressant de programmer cet algorithme de tel manière qu'il soit utilisable pour différents modèles.

2. *Off-Policy Monte Carlo Control*

a) Evaluer une politique tout en suivant une autre

Le principe de cet algorithme est d'améliorer une politique tout en se servant d'une autre pour générer l'épisode. Nous avons donc pour cet algorithme les deux politiques suivantes :

π (s, a) = politique d'évaluation (à améliorer)
π'(s, a) = politique de comportement
Condition: π (s, a) > 0 implique π'(s, a) > 0

L'algorithme est déduit de l'expression V(s) qui est l'évaluation de la politique π par rapport à π '.

$$V(s) = \frac{\sum_{i=1}^{n_s} \frac{p_i(s)}{p'_i(s)} \cdot R_i(s)}{\sum_{i=1}^{n_s} \frac{p_i(s)}{p'_i(s)}}$$

b) L'algorithme de la 'Off-*Policy' Monte Carlo Méthode*

Calcul de la fonction Politique :

- Initialisation de l'algorithme et définition des termes :
  - Soit Q (s, a) la fonction d'évaluation. A l'initialisation Q (s, a) = 0 (arbitraire).
  - Soit N (s, a) le numérateur de la fonction d'évaluation. A l'initialisation N (s, a) = 0.
  - Soit N (s, a) le dénominateur de la fonction d'évaluation. A l'initialisation D (s, a) = 0.
  - Soit Returns (s, a) la fonction mémoire de toutes les gratifications obtenues pour chaque couple (état s, action a). A l'initialisation Returns (s, a) est une liste vide.
  - Soit Reward (s, a) la fonction de gratification pour un couple (état s, action a).
  - Soit π' (s, a) une fonction politique de comportement arbitraire de type "soft-policy"
  - Soit π (s, a) une fonction politique d'évaluation déterministe arbitraire.
  - Soit les termes suivants :
    "τ" le dernier tour de la partie ou épisode pour lequel $a_\tau \neq \pi(s_\tau)$
    "t" la première occurrence de (s, a) dans l'épisode

❖ Déroulement de l'algorithme :

Off-Policy Monte Carlo Méthode

Répéter à l'infini :

(a) Générer un épisode (partie) en sélectionnant une politique π'(s, a)

(b) Calculer τ, dernier état de l'épisode pour lequel :

$a_\tau \neq \pi(s_\tau)$

(c) Pour chaque paire (s, a) de l'épisode après τ, faire :

t ← première occurrence (après τ) de (s, a)

w $\leftarrow \prod_{k=t+1}^{T-1} \frac{1}{\pi'(s_k, a_k)}$

N (s, a) ← N (s, a) + wR_t

D (s, a) ← D (s, a) + w

Q (s, a) ← N (s, a) / D (s, a)

(d) Pour chaque état 's' de l'épisode, faire :

π (s) ← argmax_a Q(s, a)

c) Problèmes et conseil de mise en application

Choix de la politique de comportement :

L'implémentation sous Scilab est facilement réalisable en utilisant les bases de l'implémentation de la 'On-Policy' Monte Carlo Méthode. Cependant on portera une attention sur l'importance du choix de la politique de comportement (soft-policy) qui améliore la vitesse d'apprentissage de la politique à évaluer.

Convergence de l'algorithme :

Cet algorithme a pour principe d'apprendre uniquement sur la fin des épisodes. Sa convergence est extrêmement lente. Il est donc important d'effectuer un apprentissage sur un grand nombre d'épisode.

Partie II, L'implémentation sous Scilab

1. L'utilisation de la RLtoolbox (RLtoolbox.sce) dans l'environnement Scilab

L'utilisation de ces algorithmes s'effectue à l'aide d'une RLtoolbox exécutable à partir du fichier script RLtoolbox.sce (modifier la ligne " path = " du fichier). A l'exécution de la RLtoolbox on peut choisir entre 3 types de méthodes d'apprentissage :

- Dynamic Programming
- Monte Carlo Control
- Temporal-Difference Learning

Après avoir choisi les méthodes de Monte Carlo, on choisi un modèle :

- BlackJack
- BlackJack Simplifié

Enfin, on sélectionne le programme désiré :

▪ Display Policy Pi(s,a)	Affichage la fonction politique évaluée.
▪ Display Policy Evaluation Q(s,a)	Affichage la fonction d'évaluation de la politique.
▪ Set Soft-Policy epsilon	Modifie la valeur de epsilon.
▪ Benchmark Policy	Test le pourcentage de réussite de la politique
▪ On Policy Learning	Apprend à l'aide de l'algorithme "on-policy"
▪ Off Policy Learning	Apprend à l'aide de l'algorithme "off-policy"

En plus, d'être disponible dans une fenêtre, les programmes pour les méthodes de Monte Carlo s'implémentent sous forme de menu dans l'environnement Scilab afin de pouvoir les utiliser manuellement.

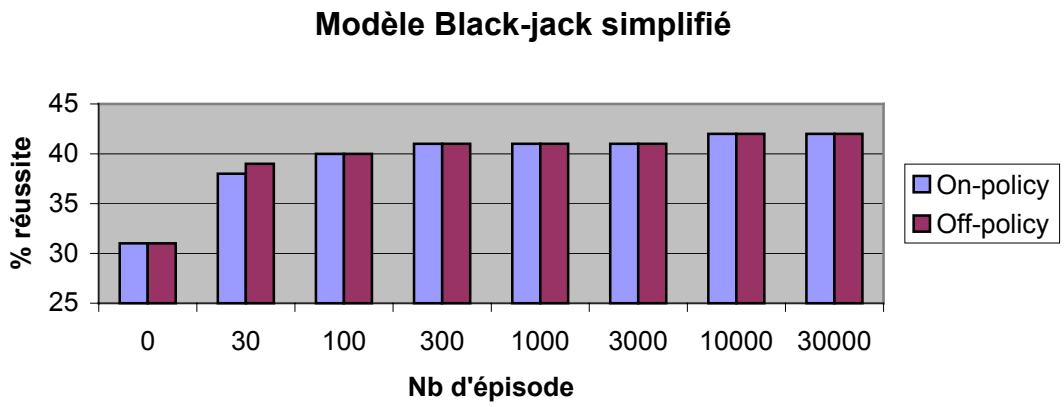
2. Les Tests

Les tests ci-dessous ont été effectués à l'aide de la fonction "benchmark" de la RLtoolbox. Le nombre d'épisode généré pour l'évaluation du pourcentage de réussite de la politique est, pour tous les tests, de 10.000 épisodes.

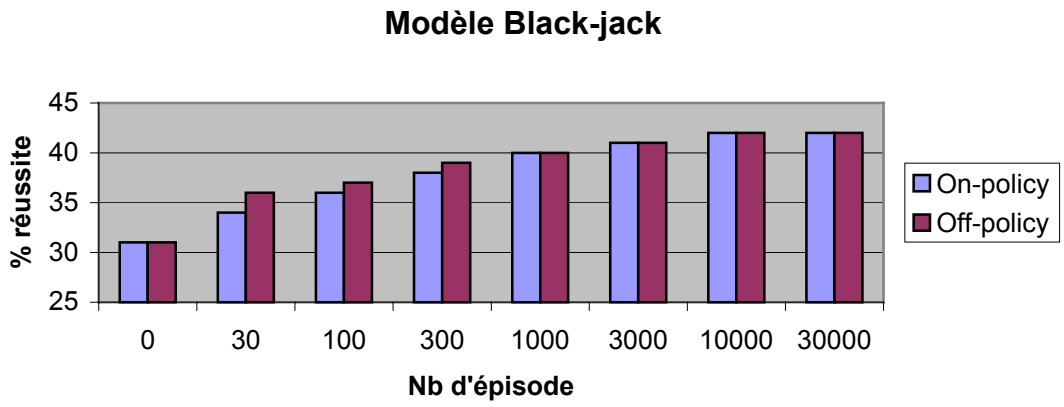
Tableaux des résultats :

METHODES DE MONTE CARLO				
Nb Episodes	Modèle			
	Black-jack		Black-jack simplifié	
	On-policy	Off-policy	On-policy	Off-policy
	% réussite	% réussite	% réussite	% réussite
0	31	31	31	31
30	34	36	38	39
100	36	37	40	40
300	38	39	41	41
1000	40	40	41	41
3000	41	41	41	41
10000	42	42	42	42
30000	42	42	42	42

Comparaison de la 'On-policy' et de la 'Off-policy' pour le modèle Black-jack simplifié :



Comparaison de la 'On-policy' et de la 'Off-policy' pour le modèle Black-jack :



3. Interprétation des résultats

Vitesse d'apprentissage selon le modèle :

La première conclusion évidente est la vitesse d'apprentissage supérieur pour le modèle Black-jack simplifié. En effet, celui-ci obtient un taux de réussite de 41% en seulement 300 épisodes au lieu de 3000 pour le modèle Black-jack. Ce résultat s'obtient indifféremment de la méthode d'apprentissage c'est-à-dire avec la "On-policy Monte Carlo control" ou la "Off-policy Monte Carlo control". On explique cette différence en raison du grand nombre état dans le modèle Black-jack (22 états du joueur × 11 états de la Banque = 242 états) contre seulement 22 pour le modèle simplifié.

Vitesse d'apprentissage selon l'algorithme :

On note une vitesse plus élevée pour un nombre d'épisode réduit pour la "Off-policy". Ainsi, avec seulement 30 épisodes, elle obtient 36% contre 34% sur le modèle Black-jack et 39% contre 38% sur le modèle Black-jack simplifié.

Extrapolation des résultats avec un nombre d'épisode infini :

La modélisation du Black-jack complet laisse à penser que pour un nombre d'itération infini, sa politique sera supérieure à celle du Black-jack simplifié car le joueur connaît une donnée du jeu supplémentaire (la seconde carte de la Banque). Ainsi avec un nombre d'itération de 1.000.000 pour l'apprentissage avec la "On-policy", on obtient un pourcentage de réussite de 44% ce qui supérieur à tous les tests effectués. On suppose que pour le même nombre d'itération avec le modèle simplifié ne pourra pas atteindre ce taux.

Partie III, Directives de développement en vue de la création de nouveaux modèles

1. Présentation de l'architecture du code

Bien que l'objectif principal de ce projet soit la production de programmes fonctionnels, le concept de la toolbox indique un besoin fort de réutilisabilité des programmes. C'est pourquoi le second objectif majeur de ce projet a été pour nous de fournir un outil réellement réutilisable. Nous nous sommes grandement attachés à séparer de manière claire le code source de nos programmes suivant que ce code soit spécifique à un modèle donné ou bien applicable à n'importe quel modèle. Ainsi, pour une plus grande distinction, les fichiers de code sont distribués dans deux répertoires distincts : le premier regroupe le code générique, que nous appellerons *Monte Carlo*, celui-ci à été pensé pour fonctionner avec n'importe quel modèle, et quelle que soit la complexité de ce dernier. Le second regroupe les fichiers dont le code est spécifique ; fichiers que nous appellerons *modèles*.

Ainsi, les fichiers du répertoire **`models`** représentent les différents modèles utilisables avec les algorithmes de Monte Carlo. **Chaque modèle** est implémenté par **deux fichiers** dont le nom est prédéfini par sa déclaration dans l'interface de la RLToolbox.
Exemple : si votre modèle simule un jeu de dames et qu'il soit déclaré sous le nom « draughts », le répertoire **`models`** devra contenir ces deux fichiers :

(1) draughts_mdl_load.sce
(2) draughts_mdl_functions.sci

En effet, dès l'enregistrement du nom « draughts » en tant que modèle dans l'interface de la RLToolbox, cette dernière associera à ce nom les deux noms de fichiers ci-dessus.
Le fichier (1) contiendra la déclaration et l'initialisation des variables d'environnement du modèle. Les noms de ces variables sont fixes car ils sont connus (et utilisés) des fichiers *Monte Carlo*.
Le fichier (2) contiendra l'ensemble des fonctions nécessaires au fonctionnement des algorithmes de Monte Carlo. Une fois encore, le nom de ces fonctions est fixé car les algorithmes de Monte Carlo les appellent depuis le code source des fichiers *Monte Carlo.*

D'autre part, le répertoire **`functions`** contient ces fichiers (*Monte Carlo*). Ils ne sont en aucun cas destinés à une quelconque modification. Ces fichiers contiennent les informations nécessaires (variables d'environnement, fonctions) à l'exécution des algorithmes. Le répertoire **`functions`** contient sept fichiers :

(1) MonteCarlo_Environnement.sce
(2) On_Policy_Learning.sce
(3) Off_Policy_Learning.sce
(4) Mc_Benchmark_Policy.sce
(5) Mc_Control_Functions.sci
(6) Mc_On_Policy_Functions.sci
(7) Mc_Off_Policy_Functions.sci

Les quatre premiers fichiers sont des scripts, les trois derniers contiennent uniquement les fonctions nécessaires à l'exécution de ces scripts.

Le fichier (1) est exécuté après chaque chargement d'un modèle en mémoire. Il utilise les variables d'environnement du modèle pour initialiser celles des algorithmes de Monte Carlo.

Les fichiers (2) (3) et (4) sont les scripts d'exécution des algorithmes. Ces algorithmes sont itératifs, c'est pourquoi il est demandé à l'utilisateur lors de leurs exécutions le nombre d'itération que doit accomplir ces programmes. Les scripts (2) et (3) sont les algorithmes de Monte Carlo, ils tendent à améliorer chacun leur politique – il ne partage pas la même matrice contenant la politique – ; le script (4) permet d'évaluer la capacité d'une politique donnée (il teste la politique du dernier algorithme utilisé) à atteindre son but, il renvoie un résultat sous forme de pourcentage de réussite.

2. Les variables d'environnement du modèle

Bien que peu nombreuses, ces variables suffisent presque à caractériser le modèle. Elles sont au nombre de trois ; trois matrices ***_s***, ***_a***, et ***A*** (leur nom est fixé et doit être conservé). Elles sont déclarées et initialisées dans le fichier *modèle*_mdl_load.sce . ***_s*** et ***_a*** sont des matrices lignes.

Pour comprendre ces matrices, il faut étudier les caractéristiques d'un état et d'une action relativement à un modèle donné. Un état, comme une action, est caractérisé par un certain nombre de paramètres. Dans le modèle du BlackJack par exemple, un état est caractérisé par le score du joueur (1^{er} paramètre) et le score de la banque ($2^{ème}$ paramètre). Ainsi, les matrices ***_s*** et ***_a*** représentent respectivement les dimensions des caractéristiques des états et les dimensions des caractéristiques des actions. Dans notre exemple, le score du joueur est compris entre 1 et 22 (le score 22 représentant tous les scores supérieurs à 21), et le score de la banque entre 1 et 11 (une seule carte dont la valeur va de 1 à 11), donc

_s = [22 11] - 2 paramètres : 22 valeurs pour le premier, 11 pour le second

de même, une action est caractérisée par un paramètre : tirer une carte(1) ou s'arrêter(2), d'où

_a = [2] - 1 paramètre : 2 valeurs

La matrice A(s,a), quant à elle, est une matrice booléenne qui, pour un couple état(E)/action(A), indique 1 si l'est autorisé de prendre l'action A dans l'état E et 0 sinon. Les dimensions de cette matrice sont, pour le Black-Jack, 3 : 2 dimensions pour la caractérisation de l'état, plus 1 pour celle de l'action. Voici un exemple d'initialisation de la matrice A :

A = ones(22, 11, 2); // Exemple d'initialisation = toutes les actions sont autorisées.

Que l'on peut affiner par :

A(21, :,1) = 0 // si le score du joueur est 21, il n'a plus le droit de tirer une carte.

Les informations contenues dans ces trois variables suffisent à initialiser l'ensemble des variables d'environnement des fonctions de Monte Carlo. Toutefois, elles ne caractérisent pas totalement le modèle ; en effet, ces variables ne permettent ni de déterminer un état initial, ni de déterminer l'état s_{i+1} qui succède à un état s_i à la suite d' une action a_i . Ce rôle est pris en charge par les fonctions spécifiques de caractérisation du *modèle.*

3. Les fonctions du modèle

Cette section a pour objectif de présenter les fonctions devant être implémentées pour chaque modèle, et d'expliciter leurs arguments. Il y a 5 fonctions à implémenter dans le fichier *modèle*_mdl_functions.sci . Quatre de ces fonctions sont impératives, la cinquième ne sert que pour le script de benchmark de la politique – les voici :

(1) function state=Mdl_Init_State()

(2) function ns=Mdl_Get_State(Episode)

(3) function Stop=Mdl_Stop(Episode)

(4) function Reward=Mdl_Reward(Episode, turn)

(5) function But=Mdl_Check_Goal(Episode)

Ces fonctions ont quasiment toutes une chose en commun : elle reçoivent ***Episode*** comme argument. ***Episode*** est une liste de matrices. Elle stocke au fur et à mesure qu'ils apparaissent dans l'épisode les couples (s,a). De plus, ***Episode(i)*** renvoie le couple (s_i,a_i) . (s,a) est une matrice ligne formée des deux matrices lignes *etat* et *action*. Dans l'exemple du Black-Jack, imaginons l'état initial suivant :

Score du joueur = 14
Score de la banque = 8

D'où , si on est fidèle aux déclarations des variables d'environnement (ici ***_s***) , on a etat = [14 8]
Mettons que l'action choisie par le joueur soit de s'arrêter : action = [2]
Ainsi, on a ***Episode(1)*** = [[14 8] [2]] = [14 8 2]

Si maintenant nous voulons savoir quel est le score du joueur au début du premier tour :
Score_du_Joueur = ***Episode(1)(1)*** = 14
De même, Score_de_la_banque = ***Episode(1)(2)*** = 8 ; et enfin action = ***Episode(1)(3)*** = 2

Autre astuce, lorsque la fonction ne dispose que de ***Episode*** , il suffit de la commande

```
turn = length(Episode)
Episode(turn)
```

pour récupérer le dernier couple (state, action) de l'épisode.

Objectifs de chaque fonction :

(1) function state=Mdl_Init_State()

Fonction appelée à chaque début d'épisode, elle renvoie la matrice ***etat*** décrivant l'état initial.

(2) function state=Mdl_Get_State(Episode)

Cette fonction est appelée pour connaître l'***etat*** du tour suivant. Ici, ***Episode*** n'est pas complet puisque l'épisode lui-même est en train d'être généré. Cela dit, il est nécessaire que cette fonction dispose de ***Episode***, car même incomplet, cette variable est parfois indispensable.

Exemple : le jeu d'échec ; il arrive parfois que les joueurs rejouent systématiquement une même séquence d'actions : il est alors nécessaire de détecter l'anomalie et de déclarer l'état suivant comme étant PAT (match nul).

(3) function Stop=Mdl_Stop(Episode)

Appelée à chaque tour, cette fonction indique si l'épisode est terminé ou non. De même que pour la fonction Mdl_Get_State, il peut être nécessaire d'identifier une séquence d'états pour déclarer l'épisode terminé. D'autre part, en récupérant la valeur *profondeur = length(Episode)* , on peut limiter l'apprentissage à une profondeur *n*.

Stop est un entier, Si ***Stop*** est différent de 0, l'épisode s'achève.

(4) function Reward=Mdl_Reward(Episode, i)

Fonction de gratification. Elle est appelée lorsque l'épisode est terminé – la liste des couples (s,a) est donc complète. Il s'agit de déterminer la gratification associée au ième couple (s,a) de l'épisode. Cette fonction est très délicate à mettre en œuvre ; en effet, de la qualité des gratifications dépend la qualité de l'apprentissage. Cette fonction renvoie une valeur (par exemple 1 si le joueur a gagné, -1 s'il a perdu, etc.)

(5) function But=Mdl_Check_Goal(Episode)

Détermine si l'épisode est une réussite : si le but a été atteint, la fonction renvoie 1 ; 0 sinon.

4. Les variables sensibles : Pi(s,a) et Q(s,a)

Les fonctions Pi(s,a) et Q(s,a) sont représentées par des matrices. Chaque algorithme, *on-policy* et *off-policy*, travaille sur sa politique et sa fonction d'évaluation. Pour éviter les conflits, la politique Pi(s,a) associée à l'algorithme *on-policy* est nommé sous Scilab ***Pi_on*** et celle associée à avec l'algorithme *off-policy* ***Pi_off*** .

De même les fonctions d'évaluations sont stockées avec des noms différents :
on-policy : matrice ***Returns***
off-policy : matrices ***N***, ***D***, ***Q_off***.

Toutefois, à la fin de chaque apprentissage, les deux algorithmes copient leurs fonctions dans les variables partagées ***Pi*** et ***Q***. Ces dernières sont par exemple utilisées pour l'affichage ou encore pour tester la politique (fonction de bench).

Ainsi, il est inutile de sauvegarder les variables ***Pi*** et ***Q***, mais il est généralement très intéressant de conserver ***Pi_on***, ***Pi_off*** et les autres en vue d'une réutilisation future. Cette précision est cependant donnée à titre purement indicatif, d'une part car l'utilisateur du programme n'a pas besoin de connaître l'existence de ces variables *cachées*, et d'autre part car l'interface possède une fonction de sauvegarde/chargement des politiques, totalement transparente pour l'utilisateur.

Bibliographie

[1]Richard S. Sutton, Andrew G. Barto. « Reinforcement Learning (Adaptive Computation and Machine Learning). Ed. MIT Press, Cambidge, MA, 1998.
[2] http://www-anw.cs.umass.edu/~rich/book/the-book.html
[3] http://www-rocq.inria.fr/scilab/