Processus de Décision Markovien

Léa Cohen-Solal
Pénélope Millet
Dirigé par Hugo Gilbert

1 Création d'une classe MDP

Nous avons crée une classe MDP contenant les attributs suivant :

- states : L'ensemble des états du MDP
- actions : L'ensemble des actions possibles du MDP
- transistions : L'ensemble des transistions possibles du MDP avec leur probabilité de réussite.
- features : L'ensemble des descriptions des états (Si la case contient le trésor, une bombe ...)
- initial state : L'état initial
- discount factor : initialisé à 0.9

1.1 Initialisation des transitions

Nous savons que le tuteur ne réussit pas toujours ses actions, nous devons donc créer toutes les transitions possibles.

La fonction 'calculate_next_state' permet, pour chaque état **s**, d'identifier les états **s'** atteints par l'une des actions possibles. Pour chaque transition, il y a une probabilité de réussite de 80%.

La fonction 'calculate_slip_states' détermine les états résultants d'un état **s** et d'une action **a** en cas de glissement. Par exemple, si l'action est 'U' (UP) et que l'agent glisse dans une direction perpendiculaire, nous calculons les états atteints en effectuant les actions 'L' (LEFT) et 'R' (RIGHT).
Comme il y a une probabilité de 20% de glisser, nous supposons que les deux directions sont également probables avec la même probabilité.
Ainsi, pour l'action 'U', il y a 10% de chances d'aller vers la droite 'R' et 10% d'aller vers la gauche 'L'.

1.2 Affichage du MDP

La fonction 'print_caracteristics' permet de visualiser la structure du MDP.

2 Création du MDP de l'énoncé

Voir 'create_mdp()'
Les états sont {H1, H2, ..., H8, G1, ..., G8, ..., A1, ..., A8}.
Les actions sont {'R', 'L', 'U', 'D'} pour Right, Left, Up, Down.
Dans cet MDP, il y a 5 types de cases particulières :
1 - la case du trésor
2 - la case où se trouve une bombe
3 - la case où se trouve un mur
4 - la case avec de l'eau
5 - la case où se trouve une montagne
Une case est donc identifiée par un tuple indiquant si une des caractéristiques est présente sur cette case.
Par exemple, la case A1 contient le trésor (caractéristique 1) et rien d'autre. On a donc A1 : (1,0,0,0,0).
La case G3 contient de l'eau et un mur, on a donc G3 : (0,0,1,1,0).
Nous initialisons aussi l'état initial 'D7' comme montré dans l'énoncé.

3 Création d'un Gridworld aléatoire

Voir generate_random_gridworld(taille, features_number)

La fonction prend en paramètre les dimensions du *gridworld* ainsi que le nombre de cases contenant chaque *feature*. Ce dernier paramètre est défini comme une liste, où chaque élément de la liste correspond au nombre de cases pour sa caractéristique (feature) respective.

Par exemple, l'élément n°3 de la liste contient le nombre de cases possédant la caractéristique n°3.

Pour chaque caractéristique (caractérisant n états), nous récupérons n états aléatoirement et nous lui affectons la caractéristique.Un état peut avoir plusieurs caractéristiques.

4 Policy Iteration

Voir PolicyIteration(reward_function,actual_best_policy)

Cette fonction trouve la meilleure politique en fonction de la fonction de récompense *reward_function*.
On peut commencer avec une politique existante ou une politique aléatoire. L'algorithme fonctionne ainsi :

1. Pour chaque nouvelle politique trouvée, nous mettons à jour les valeurs de chaque état s avec la suite:

$$v_0(s) = 0 \quad \forall s$$

$$v_{t+1}(s) = R(s) + \gamma \sum_{s' \in S} T(s, \pi(s), s') v_t(s')$$

où s' sont tous les états accessibles par l'action $\pi(s)$.
Tant que v_t ne converge pas pour tous les états, on continue.
Cette étape correspond à la **PolicyEvaluation** de l'algorithme.

Une fois que les valeurs sont stables, on met à jour la politique optimale pour chaque état. Pour chaque état 's', la politique optimale pour cet état est l'action a maximise :

$$R(s) + \gamma \sum_{s' \in S} T(s, a, s') v_t(s')$$

2. Une fois la politique mise à jour, on vérifie si elle a changé par rapport à l'itération précédente. Si elle n'a pas changé, cela signifie que la politique est stable et que nous avons trouvé la politique optimale (Nous stoppons l'algorithme).
Sinon, nous revenons à l'étape 1 de l'évaluation de la politique (PolicyEvaluation). Ce processus se poursuit jusqu'à ce que la politique converge.
À la fin, nous obtenons une politique optimale accompagnée d'une valeur optimale.

5 M-state-action

La fonction 'M-state-actions' prend en paramètre une **reward function**, un nombre de pas, un état initial et va simuler le parcours de l'agent en M étapes suivant la meilleure politique.
Sachant que l'agent est "imparfait", il choisira l'action optimale à 95% et une action aléatoire à 5%.
Une fois l'action choisie, l'agent n'est pas sûr à 100% de réussir cette action. Il a 80% de chance de réussir (donc d'atterrir dans l'état de l'action) et 20% de glisser perpendiculairement à cette action.
La fonction répète donc ce processus M fois et renvoie le parcours effectué.

6 Calcul du ratio

Pour pouvoir estimer au mieux la reward function, nous avons crée une classe BayesianFramework. Cette classe prend en atribut :

1. Une distribution prior P(R)
2. Un argument **param_prior**en fonction de la distribution prior
3. Un MDP de la classe MDP

4. Une suite d'observations M-state-action

5. un paramètre α

Comme nous pouvons utiliser plusieurs distribution de la reward function, la fonction **Probability_R(self, reward_function)** permet d'obtenir P(R) en fonction de la distribution qu'elle suit.
Les 3 différentes distributions prises en compte sont Uniforme, Beta et Gaussienne. En fonction des distributions utilisées, les paramètres nécessaires à cette distribution se trouvent dans l'attribut *param_prior* de la classe.

- S'il s'agit d'une distribution uniforme ou Beta, l'attribut **param_prio** est $R_{\max}$, la valeur maximale que peut prendre la fonction de récompense.
- S'il s'agit d'une distribution gaussienne, l'attribut **param_prior** est la variance de la distribution.

6.1 Calcul du prior $P(R)$

Si c'est une distribution Beta, nous ne pouvons pas calculer $P(R)$ s'il y a des valeurs négatives. Nous utilisons donc une normalisation de la fonction de récompense, qui n'affectera pas le résultat de la probabilité. Cette normalisation est de la forme :

$$R_i = \frac{R_i + R_{\max}}{2R_{\max}}$$

de sorte à ne pas avoir de valeur négative.
Pour les distributions Uniformes, on applique la formule de la distribution uniforme :

$$P(R) = \left(\frac{1}{2R_{\max}}\right)^t$$

où t représente le nombre de caractéristiques et $2R_{\max}$ est l'étendue des valeurs de R.
Pour la distribution Gaussienne, la formule correspond à la densité de probabilité suivante :

$$P(R) = \prod_i \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x_i)^2}{2\sigma^2}}$$

où σ represente la variance mis en attribut de la classe.

6.2 Calcul du likelihood P(O|R)

Voir fonction calculate_likelihood(reward_function, states_actions)

Pour calculer $P(O|R)$, nous avons besoin de calculer $Q^*(s,a)$ pour tous les états-actions du gridworld. En sachant que :

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') v^*(s')$$

nous avons besoin de déterminer $v^*(s)$ pour chaque état. On utilise donc la fonction **ValueIteration** qui permet d'obtenir les valeurs optimales pour chaque état en fonction de la reward_function. La fonction **ValueIteration** réalise la suite suivante :

$$v_0(s) = 0 \quad \forall s$$

$$v_{t+1}(s) = \max_{a \in A} \left(R(s) + \gamma \sum_{s' \in S} T(s,a,s') v_t(s') \right)$$

Tant que v ne converge pas, continuer les itérations.
Une fois cette valeurs calculée on peut directement avoir $P(O|R)$ en itérant sur les state-actions comme dans la formule de l'énoncé.

6.3 Calcul du ratio

Nous cherchons à calculer $\frac{P(R1|O)}{P(R2|O)}$

$$P(R1|O) = \frac{P(O|R1)P(R1)}{P(O)}$$

$$P(R2|O) = \frac{P(O|R2)P(R2)}{P(O)}$$

donc

$$\frac{P(R1|O)}{P(R2|O)} = \frac{P(O|R1)P(R1)P(O)}{P(O|R2)P(R2)P(O)} = \frac{P(O|R1)P(R1)}{P(O|R2)P(R2)}$$

Les fonctions **Probability_R** et **calculate_likelihood** permettent d'obtenir cette probabilité. Si P(R1) ou P(R2) est égale à 0 alors nous mettons le ratio à 0.

7 PolicyWalk

L'algorithme de PolicyWalk a pour but d'estimer au mieux la politique optimale et la **reward_function**.
L'algorithme commence par sélectionner aléatoirement une fonction de récompense initiale **R** et calcule la politique optimale associée en utilisant **PolicyIteration**.
Cette étape fournit une première estimation de la politique π_0.
Pour affiner l'estimation de la fonction de récompense, l'algorithme génère une nouvelle fonction de récompense R' en modifiant légèrement R. La génération de ces nouvelles fonctions de récompense est réalisée par la fonction **list_of_neighbours(R, stepsize)**. Cette fonction crée toutes les fonctions de récompense possibles qui ne diffèrent de R que par une seule composante, chaque composante étant ajustée de + ou − **stepsize**.
Parmi les fonctions de récompense générées, une R' est choisie aléatoirement. L'algorithme calcule alors la probabilité que R' soit plus probable que R et met à jour R en conséquence.

8 PolicyWalkModified

Nous avons modifié l'algorithme de PolicyWalk sur la probabilité de mettre à jour la fonction de récompense R avec l'une de ses voisines R'. La probabilité est passée de

$$\min\left(1, \frac{P(R'|O)}{P(R|O)}\right)$$

à

$$\min\left(1, \left(\frac{P(R'|O)}{P(R|O)}\right)^{1/T_i}\right)$$

où T_i représente une température à l'itération i de l'algorithme. Dans l'énoncé, il est dit que cette température doit être initialement élevée et doit décroître au fur et à mesure des itérations. Nous pouvons donc initialiser cette température à 100 et la faire décroître de 0,01 à chaque itération (taux de refroidissement de 0,99). Cependant, dans l'article [1] mentionné dans l'énoncé, nous voyons une formule déjà établie pour T_i telle que :

$$\frac{1}{T_i} = 25 + \frac{i}{50} \implies T_1 = 1/25,02$$

Cette valeur ne semble pas être très élevée mais diminue tout de même au fur et à mesure des itérations. Dans l'algorithme, nous avons utilisé le taux de refroidissement à 0,99 mais nous avons aussi créé la fonction `calculate_Ti(i)` qui renvoie T_i avec la formule de l'article.

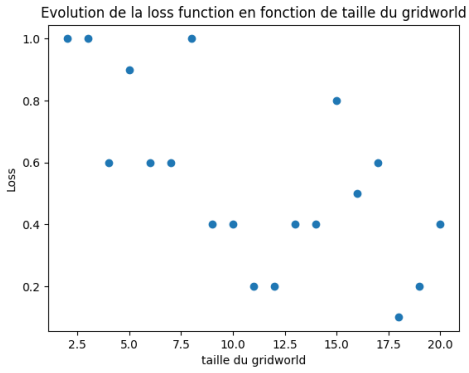
9 Tests

Nous avons testé nos algorithmes en fonction de differents paramètres :

- la taille du gridworld
- le nombre d'itération dans l'algorithme PolicyWalk
- le taux d'apprentissage
- le nombre de features
- le stepsize λ

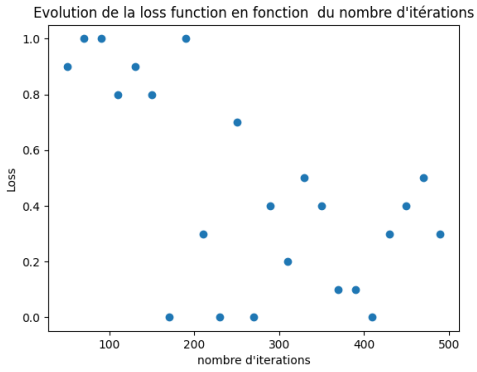
9.1 Tests en fonction de la taille du gridworld

La fonction **perf_from_size(iteration)** permet de comparer les scores obtenus de l'algorithme PolicyWalk en fonction de la taille du gridworld. Cette fonction crée des gridworlds de taille 2x2 à 20x20 et exécute 10 fois l'algorithme PolicyWalk sur ces gridworlds pour obtenir une moyenne des "loss" de l'algorithme.
Le graphique suivant montre l'évolution de la fonction de perte en fonction de la taille du gridworld.



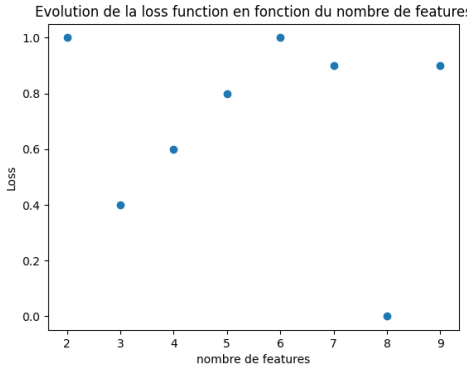
9.2 Tests en fonction du nombre d'itérations de la fonction PolicyWalk

La fonction **perf_from_iteration()** permet de comparer les scores obtenus de l'algorithme PolicyWalk en fonction du nombre d'itérations de l'algorithme. Nous remarquons encore sans surprise que les performances de l'algorithme augmentent proportionnellement au nombre d'itérations de l'algorithme.



9.3 Tests en fonction du nombre de features du MDP

Nous avons maintenant testé les performances de l'algorithme PolicyWalk en fonction du nombre de caractéristiques attribuées au gridworld. La fonction **perf_from_number_features()** teste l'algorithme avec des gridworlds allant de 2 à 10 caractéristiques.



Les résultats ne semblent pas concluants et montrent une petite diminution de la performance de l'algorithme proportionnellement au nombre de caractéristiques.

9.4 Tests en fonction du stepsize

Enfin, nous avons testé, avec la fonction **perf_from_stepsize()**, les performances de l'algorithme PolicyWalk en fonction du stepsize λ entré en paramètre. Nous faisons varier ce λ de $\frac{R_{\max}}{2}$ à $\frac{R_{\max}}{100}$. Ces différents tests ne nous permettent pas de conclure sur l'influence du stepsize sur les performances de l'algorithme PolicyWalk car les résultats sont assez dispersés. Le graphique montre comment évolue la loss en fonction du paramètre x tel que $\frac{R_{\max}}{x}$ sera le stepsize.