

IA & SMA

TP2. Apprentissage par renforcement

Laëtitia Matignon

Pour ce TP, reprenez le code utilisé dans le TP1.

Les éléments à compléter/rendre dans ce TP2 sont :

- le fichier `RAPPORT.md` à compléter OU un document **au format pdf** dans lequel **vous préciserez vos choix de modélisation** pour :
 - partie 2.2 : les éléments que vous avez utilisés pour la définition d'un état du MDP pour le jeu du Pacman
 - partie 2.3 : les fonctions caractéristiques que vous avez choisies pour la classe `FeatureFunctionPacman`.
 - les classes suivantes sont à compléter :
 - `QLearningAgent.java` du package `agent.rlagent`
 - `StrategyGreedy.java` du package `agent.strategy`
 - `EtatPacmanMDPClassic` du package `pacman.environnementRL`
 - `QLApproxAgent` du package `agent.rlapproxagent`
 - `FeatureFunctionIdentity` du package `agent.rlapproxagent`
 - `FeatureFunctionPacman` du package `agent.rlapproxagent`
-

L'objectif du TP est d'implémenter et de tester dans différents environnements un algorithme d'apprentissage par renforcement : le Q-learning.

Vous trouverez en annexe de cet énoncé de TP le diagramme de classes des éléments utilisés dans ce TP.

1 Agent Q-Learning tabulaire

On s'intéresse ici à l'algorithme du Q-learning tabulaire (cf. transparents de cours), c'est-à-dire utilisant une matrice de Q-valeurs. Cette matrice notée Q est de dimension $|S| \times |A|$ avec des valeurs dans \mathbb{R} .

La classe pour implémenter un agent apprenant avec le Q-learning est nommée `QLearningAgent`. Elle se trouve dans le package `agent.rlagent`. Elle hérite de la classe abstraite `RLAgent`, qui se charge de gérer le lancement des épisodes¹ et l'interaction de l'agent avec l'environnement à chaque action. En particulier, lorsqu'un agent réalise une action a , la fonction `endStep` de `QLearningAgent` est appelée avec les paramètres correspondant à l'interaction de l'agent avec l'environnement. Ces paramètres sont (s, a, s', r) avec s l'état dans lequel l'agent a réalisé l'action a , s' l'état d'arrivée et r la récompense reçue.

1.1 Agent avec exploration manuelle

Tout d'abord, l'agent sera contrôlé manuellement via les flèches du clavier.

Question 1 Compléter la classe `QLearningAgent` pour que l'agent mette à jour sa table des Q-valeurs en fonction des actions réalisées au clavier.

Quelques précisions par rapport à la question précédente :

- pour toute modification d'une Q-valeur, utilisez dans votre code la méthode `setQValeur` (méthode que vous devez compléter), la vue sera ainsi automatiquement notifiée et mise à jour.

Question 2 Pour vérifier votre implémentation, testez votre agent en exécutant la classe `testQLAgentGridworld` du package `simuTP2`. Vous choisirez une stratégie d'exploration manuelle (vous déplacez vous-même l'agent dans le labyrinthe). Vérifier les différents éléments calculés (Q-valeurs, valeurs et politique) par votre agent. Le bouton `ResetAgent` permet de replacer l'agent à son état de départ.

1.2 Agent avec stratégie d'exploration

La stratégie d'exploration utilisée par l'agent est implémentée dans la classe `StrategyGreedy` du package `agent.strategy`.

Question 3 Complétez la classe `StrategyGreedy` de sorte à ce que l'agent utilise une stratégie d'exploration ϵ -greedy (cf. transparents de cours).

Question 4 Testez votre agent en exécutant la classe `testQLAgentGridworld` et en choisissant une stratégie greedy. Les valeurs obtenues après quelques centaines d'épi-

1. Un épisode débute avec l'agent à son état initial et se termine lorsque l'agent atteint un état absorbant ou lorsque le nombre d'actions depuis l'état initial est supérieur au nombre d'actions maximum autorisées par épisode.

sodes et $\epsilon = 0.1$ doivent être du même ordre que les valeurs calculées par votre agent value iteration du TP1.

Quelques précisions par rapport à la question précédente :

- Dans l'interface graphique, lorsque vous lancez un seul épisode ($n = 1$), vous pouvez voir l'agent se déplacer dans l'environnement. Lorsque vous lancez plusieurs épisodes ($n > 1$), l'affichage de l'état courant de votre agent est désactivé. Le nombre de pas (ou nombre d'actions) par épisode s'affiche dans la console.

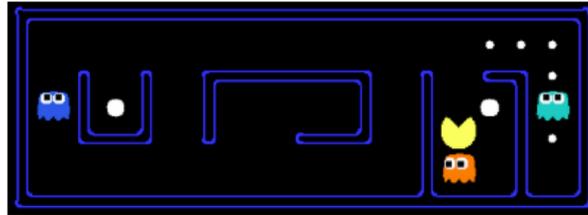
1.3 Nouvel environnement : le robot "*crawler*"

On considère maintenant un robot composé d'un bras en deux parties. La partie haute du bras possède 4 positions possibles, et la partie basse 6 positions. 4 actions sont possibles correspondant au fait de monter ou descendre d'une position l'une des deux parties du bras.

Question 5 *Exécuter la classe `testMoveCrawler` pour contrôler manuellement le robot. La récompense reçue après chaque transition s'affiche dans la console. Quel comportement aura le robot s'il suit sa politique optimale ?*

Question 6 *Vérifier votre réponse à la question précédente en utilisant votre agent *q-learning* pour faire apprendre ce comportement au robot. Pour cela, exécutez la classe `testQLAgentCrawler` du package `simuTP2`.*

2 Maintenant Jouons à Pacman !



On souhaite maintenant avoir un agent qui apprend par renforcement à jouer au jeu de Pacman.

2.1 Jeu Pacman

Les règles du jeu Pacman utilisé dans ce TP sont définies telles que l'on a :

- un seul pacman,
- un ou plusieurs fantômes, qui se déplacent de façon aléatoire
- uniquement des pac-dots (pas de big-dots²)
- la partie est gagnée si le pacman mange tous les dots

2. un big-dot mangé par un pacman entraîne une invincibilité du pacman pendant quelques instants

- la partie est perdue si le pacman est mangé par un fantôme (une seule vie pour le pacman)
- le score est calculé de la manière suivante à chaque itération³ : manger un dot rapporte 10 points, gagner la partie rapporte 500 points, perdre la partie rapporte -500 points. Le score est décrémenté de 2 à chaque itération.

Un état du jeu à chaque instant est défini dans la classe `pacman.elements.StateGamePacman`. Il contient la configuration du labyrinthe (position des murs, positions des fantômes et des pac-dots, ...), l'état du pacman et des fantômes, le nombre de dots mangés, le score,

Vous pouvez tester votre performance au jeu de Pacman (sans y passez le reste du TP ...) en exécutant la classe `PacmanGame` du package `simuProjetPacman`.

2.2 Q-Learning tabulaire pour le jeu Pacman : comment définir les états du MDP ?

On souhaite maintenant utiliser l'algorithme Q-Learning que vous avez codé précédemment pour qu'un agent apprenne à jouer au Pacman. **La classe `QLearningAgent.java` ne doit pas être modifiée ici.**

La classe `QLearningAgent.java` utilise les interfaces `Etat`, `Action` et `Environnement`, qui doivent être implémentées pour chaque environnement dans lequel un agent doit apprendre. Les interfaces `Etat` et `Action` représentent ici les états et actions du MDP. Dans le cas du jeu de Pacman, les éléments suivants sont déjà implémentés :

- l'ensemble des **actions** : l'agent pacman peut faire au plus⁴ 4 actions à chaque itération (NORD,SUD,EST,OUEST); les actions d'un agent pacman sont déterministes, la stochasticité de l'environnement vient des déplacements des fantômes.
- l'environnement est implémenté dans la classe `EnvironnementPacmanMDPClassic`
- la fonction de **récompense** à chaque itération est définie comme la différence de score entre l'itération courante et l'itération précédente.

Seuls les états du MDP doivent être implémentés pour que vous puissiez utiliser votre classe `QLearningAgent.java`. Pour cela, la classe `EtatPacmanMDPClassic` du package `pacman.enviromentRL` implémente l'interface `Etat`. Elle sert à définir les états du MDP dans le cas du jeu Pacman. A chaque étape du jeu, une instance d'`EtatPacmanMDPClassic` est créée à partir de l'état du jeu Pacman courant (`StateGamePacman`). Cette instance est ensuite utilisée comme une instance d'`Etat` par la classe `QLearningAgent.java`.

L'objectif dans cette partie est de compléter la classe `EtatPacmanMDPClassic` pour définir correctement un état MDP pour le jeu du Pacman. Si on utilise directement un état du jeu Pacman (`StateGamePacman`) comme un état du MDP, l'ensemble des états sera trop grand. De plus, certaines informations dans l'état du jeu

3. Une itération du jeu consiste en un déplacement du pacman puis le déplacement des fantômes.

4. Seules les actions qui n'emmènent pas dans un mur sont considérées.

Pacman ne sont pas indispensables pour la décision de l'agent. Il faut donc extraire de l'état du jeu Pacman à chaque instant, les éléments nécessaires à la prise de décision.

1. Complétez la classe `EtatPacmanMDPClassic` du package `pacman.environnementRL`. Une instance de `EtatPacmanMDPClassic` sera créée à chaque étape du jeu. Cette instance sera utilisée comme `Etat` par la classe `QLearningAgent.java`, qui utilise les `Etat` comme des clefs dans sa Map. Il faudra donc veiller à redéfinir correctement la méthode `hashCode()`.
2. Testez votre agent en exécutant la classe `testRLPacman` du package `simuProjetPacman`. Vous pouvez modifier différents paramètres dans cette classe, comme le labyrinthe utilisé, le nombre de parties d'apprentissage, le nombre de parties de test (politique gloutonne), l'affichage du jeu pour les parties de test, ...
3. Vérifiez que l'apprentissage fonctionne correctement en regardant la courbe d'apprentissage (qui trace la somme des récompenses reçues par épisode) et les statistiques sur le pourcentage de parties gagnées en phase de test (politique gloutonne).
4. Testez avec d'autres environnements pour voir les limites de l'approche tabulaire dans ce contexte.

2.3 Q-Learning et généralisation pour le jeu Pacman

Dans cette partie, l'objectif est d'améliorer les capacités de généralisation entre états du Q-Learning, en utilisant une fonction d'approximation pour approximer la fonction Q (cf. Annexe 3).

La fonction Q est approximée comme une combinaison linéaire de M *features* ou fonctions caractéristiques ϕ_i :

$$\begin{aligned} Q_\theta(s, a) &= \sum_{i=1}^M \phi_i(s, a)\theta_i \\ &= \theta_1\phi_1(s, a) + \theta_2\phi_2(s, a) + \dots \end{aligned} \quad (1)$$

avec $\theta_i \in \mathfrak{R}$ les poids, $\phi_i : S \times A \rightarrow \mathfrak{R}$ les fonctions caractéristiques.

Les *features* sont *choisies* (dans ce TP vous devrez les définir) mais les poids sont *appris* de sorte à avoir la meilleure approximation de la fonction Q . Ainsi, à chaque étape (s, a, s', r) , chaque poids est mis à jour selon :

$$\forall k \in [1..M] \theta_k \leftarrow \theta_k + \alpha(r + \gamma \max_b Q_\theta(s', b) - Q_\theta(s, a))\phi_k(s, a) \quad (2)$$

L'interface `FeatureFunction` du package `agent.rlapproxagent` est commune à tout vecteur de fonctions caractéristiques. Elle permet de récupérer le nombre de *features* ainsi que le vecteur de *features* pour un couple (s, a) .

1. Tout d'abord, compléter la classe `QLApproxAgent` du package `agent.rlapproxagent` pour avoir un agent apprenant par Q-Learning, avec une combinaison linéaire de *features* pour approximer la fonction Q . Le constructeur prend en paramètre un vecteur de *features* composé d'un ensemble de ϕ_i .
Remarque : Vous ne devez plus utiliser la table Q , mais une approximation linéaire. En particulier, la méthode `getQValeur` est redéfinie dans la classe `QLApproxAgent`. Il faut vous assurer que les méthodes de la classe mère `QLearningAgent` que vous appelez dans la classe `QLApproxAgent` ne dépendent pas de la table Q et utilisent `getQValeur`.
2. Ensuite, compléter la classe `FeatureFunctionIdentity` qui renvoie un vecteur de *features* simple. Ce vecteur a une taille égale au nombre de couples (s, a) et est composé uniquement de 0 et d'un seul 1. Pour chaque élément (s, a) , on a un seul 1 positionné au même endroit.
3. Compléter la classe `EtatPacmanMDPClassic` du package `pacman.envIRONNEMENTRL` en lui ajoutant une méthode `int getDimensions()` qui renvoie le nombre total d'états. Cette méthode est nécessaire au constructeur de la classe `FeatureFunctionIdentity`.
4. Avec ces *features*, l'agent `QLApproxAgent` devrait fonctionner de manière identique au `PacmanQAgent`. Testez votre `QLApproxAgent` en exécutant la classe `testRlPacman` du package `simuProjetPacman` (**vous devez modifier la méthode `setRLAgent` pour instancier le bon agent**).

Une fonction caractéristique capture une ou plusieurs propriétés importante d'un état. Dans le cas du pacman, on pourra par exemple utiliser les fonctions caractéristiques suivantes :

- $\phi_0(s, a) = 1 \forall s \in S, a \in A$ est le biais
- $\phi_1(s, a)$ est le nombre de fantômes qui peuvent atteindre en un pas la position future du pacman (position atteinte par le pacman lorsqu'il fait a dans s)
- $\phi_2(s, a)$ est la présence d'un pac-dot à la position future du pacman (position atteinte par le pacman lorsqu'il fait a dans s)
- $dist(s, a)$ est la distance au plus proche pac-dot depuis la position future du pacman (position atteinte par le pacman lorsqu'il fait a dans s).

$$\phi_3(s, a) = \frac{dist(s, a)}{mapsize}$$

1. Compléter la classe `FeatureFunctionPacman` qui renvoie un vecteur de *features* pour le jeu du Pacman.
2. Testez votre `QLApproxAgent` en exécutant la classe `testRlPacman` du package `simuProjetPacman` (**vous devez modifier la méthode `setRLAgent` pour instancier le bon agent**).

3 Annexe : Apprentissage par renforcement et généralisation avec fonctions d'approximation linéaires

L'objectif est d'être capable de généraliser les valeurs apprises dans les états rencontrés à de nouvelles situations inconnues.

Dans le cas d'un espace d'état de grande dimension, on est confronté au problème appelé **curse of dimensionality** : la quantité de données d'entraînement nécessaires pour un bon apprentissage (couverture de l'espace d'état) augmente de manière exponentielle avec la dimension de l'espace d'état. Les conséquences sont un coût élevé pour obtenir des données (notamment dans le cas de l'apprentissage supervisé où les données d'entraînement doivent être labellisées), une augmentation du temps de calcul et des besoins en mémoire.

En apprentissage par renforcement, contrairement à la plupart des méthodes d'apprentissage automatique, l'agent apprenant peut « créer » ses propres données d'entrées par l'intermédiaire de ses interactions avec l'environnement. Cependant, avec de grands espaces d'états et d'actions, l'agent doit réaliser un très grand nombre d'expériences pour collecter assez de données.

Une approche classique pour résoudre ce problème est l'utilisation de *features* ou **fonctions caractéristiques**. Une feature contient une information spécifique sur les données. Toutes les features sont réunies dans un **vecteur de features** qui a une dimension plus faible que les données elles-mêmes. L'algorithme d'apprentissage prend alors en entrée ce vecteur de features et apprend la pondération de ces features.

Le choix des features pour une tâche spécifique peut se faire de manière experte (comme dans le TP Pacman). On peut aussi apprendre les features en utilisant les réseaux de neurones.

3.1 Q-Learning avec approximation linéaire et vecteur de caractéristiques

3.1.1 Notations

La fonction Q est approximée comme une combinaison linéaire de *features*/fonctions caractéristiques ϕ_i :

$$\begin{aligned} Q_{\theta}(s, a) &= \sum_{i=1}^M \phi_i(s, a)\theta_i \\ &= \theta_1\phi_1(s, a) + \theta_2\phi_2(s, a) + \dots \\ &= \boldsymbol{\phi}(s, a)^T \boldsymbol{\theta} \end{aligned} \tag{3}$$

avec $\boldsymbol{\phi}$ un vecteur de features qui a pour éléments des fonctions caractéristiques $\phi_i(s, a) : S \times A \rightarrow \mathbb{R}$ et $\boldsymbol{\theta}$ un vecteur de poids de même taille que le vecteur de features.

Un fonction caractéristique doit renvoyer des valeurs similaires pour des paires d'état-action similaires.

NOTE : On utilise généralement comme premier élément dans le vecteur de features une fonction caractéristique $\phi_1 = 1$ qui est le biais.

3.1.2 Fonctions caractéristiques expertes, apprentissage des poids

L'objectif est d'apprendre les poids $\theta_i \in \mathbb{R}$ pour obtenir une bonne approximation de la fonction Q . A chaque échantillon (s, a, s', r) ,

— la prédiction est $\hat{y} = Q_\theta(s, a) = \sum_i \theta_i \phi_i(s, a)$ (\hat{y} dépend de θ)

— la valeur cible est $y = r + \gamma \max_b Q_\theta(s', b)$ (y dépend de θ)

La valeur cible est estimée par amorçage : on utilise l'estimation des états successeurs pour estimer l'état courant.

Cela revient à trouver θ qui minimise une fonction de l'erreur entre la prédiction et la valeur cible sur échantillon de k exemples :

$$\operatorname{argmin}_\theta E(\theta) \text{ avec } E(\theta) = 1/2 \sum_k (y_k - \hat{y}_k)^2$$

Pour cela, on applique une méthode par descente de gradient qui permet de trouver le minimum d'une fonction.

— la descente de gradient est appliquée à une fonction $E(\theta)$ dont on cherche le minimum :

$$E(\theta) = 1/2 (y - \sum_i \theta_i \phi_i(s, a))^2 = 1/2 (y - \theta_1 \phi_1(s, a) - \dots - \theta_m \phi_m(s, a))^2$$

— à partir de θ on suit le vecteur donné par le gradient de la fonction E en (θ) :

$$\vec{\nabla} E(\theta) = \left(\frac{\partial E(\theta_1, \dots, \theta_m)}{\partial \theta_1}, \dots, \frac{\partial E(\theta_1, \dots, \theta_m)}{\partial \theta_k}, \dots \right)^T$$

— le calcul du gradient donne :

$$\frac{\partial E(\theta)}{\partial \theta_k} = (y - \sum_i \theta_i \phi_i(s, a)) \left(\frac{\partial (y - \sum_i \theta_i \phi_i(s, a))}{\partial \theta_k} \right) = (y - \sum_i \theta_i \phi_i(s, a)) (-\phi_k(s, a))$$

$$\vec{\nabla} E(\theta) = -(y - \sum_i \theta_i \phi_i(s, a)) \phi$$

— la mise à jour du vecteur de poids à chaque pas selon (s, a, s', r) est alors :

$$\theta_{t+1} = \theta_t - \alpha \vec{\nabla} E(\theta_t) \quad (4)$$

$$\theta_{t+1} = \theta_t + \alpha (y - \sum_i \theta_i \phi_i(s, a)) \phi \quad (5)$$

$$\theta_{t+1} = \theta_t + \alpha (r + \gamma \max_b Q_{\theta_t}(s', b) - Q_{\theta_t}(s, a)) \phi(s, a) \quad (6)$$

— cela revient à mettre à jour chaque poids k à chaque pas selon (s, a, s', r) :

$$\theta_k \leftarrow \theta_k - \alpha \frac{\partial E(\boldsymbol{\theta})}{\partial \theta_k}$$

$$\theta_k \leftarrow \theta_k + \alpha(y - \hat{y})\phi_k(s, a)$$

$$\theta_k \leftarrow \theta_k + \alpha(r + \gamma \max_b Q(s', b) - Q(s, a))\phi_k(s, a)$$

