

Apprentissage par renforcement :

Méthodes

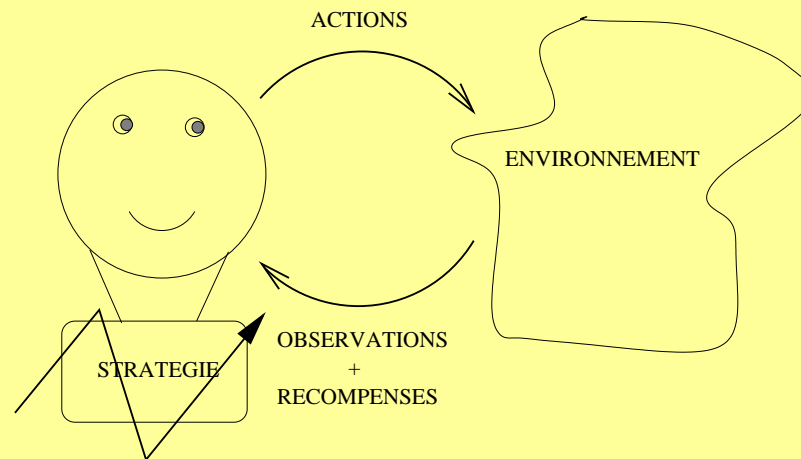
Frédéric Garcia



*Unité de Biométrie et Intelligence Artificielle
BP 27, 31326 Castanet-Tolosan*

Rappel du paradigme de l'AR

Un agent autonome agissant au sein d'un environnement, qui recherche au travers d'expériences itérées un comportement décisionnel optimal.



AR et PDM

L'apprentissage par renforcement étend la problématique et les méthodes des PDM selon plusieurs directions :

- pas de modèle du PDM a priori mais observation de trajectoires contrôlées
- mise à jour locale de la fonction de valeur
- problèmes décisionnels de grande taille avec paramétrisation des fonctions de valeur ou des politiques.

Apprentissage par essais-erreurs

Dans l'état s_n on sélectionne et on applique l'action a_n

On observe l'état résultant s_{n+1} et l'utilité instantanée r_n

$$\begin{array}{ccc} s_n, a_n & \rightsquigarrow & s_{n+1} \\ & \downarrow & \\ & r_n & \end{array}$$

Ces informations sont exploitées pour apprendre une politique optimale

Rappel sur la notion de modèle

Un modèle est ici la donnée des distributions de probabilité $p(s'|s, a)$, et des fonctions de récompense $r(s, a)$

Plusieurs familles d'algorithmes

Comme pour les PDM, plusieurs critères d'optimalité peuvent être considérés (γ -pondéré, moyen, etc.)

Exemples pour le critère γ -pondéré :

	modèle estimé	pas de modèle
itération de la valeur	ARTDP	Q-learning
itération de la politique	<i>actor-critic methods</i>	

Adaptive Real-Time Dynamic Programming

Le modèle est identifié en ligne

$$\begin{aligned} p(s'|s, a) &\leftarrow \frac{n_{s,a}^{s'}}{n_{s,a}} \\ r(s, a) &\leftarrow \mathbf{r}_n \end{aligned}$$

Itération de la valeur avec mise à jour locale

$$V_{n+1}(s_n) \leftarrow \max_a \{ r(s_n, a) + \gamma \sum_{s'} p(s' | s_n, a) V_n(s') \}$$

Choix des transitions

L'état s_n est l'état résultant de la dernière transition (contrainte de simulation ou temps-réel)

Le choix de l'action a_n dans A peut être guidé par la fonction courante V_n

Q-learning

En l'absence de modèle, la seule fonction de valeur V ne permet pas de définir une politique associée

$$\pi(s) = \operatorname{argmax}_a \{r(s, a) + \gamma \sum_{s'} p(s' | s, a) V(s')\}$$

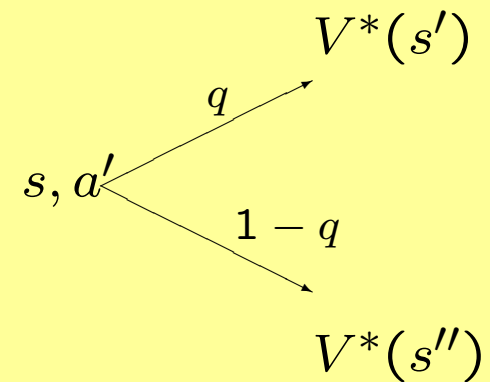
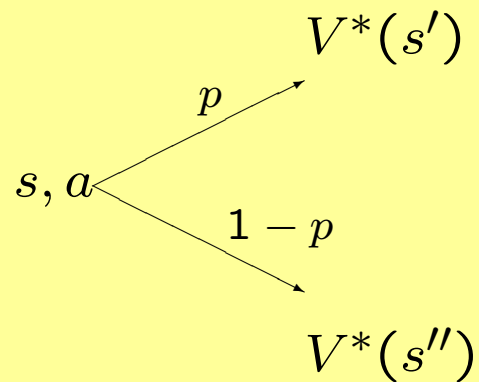
On introduit donc la **fonction de Q-valeur**

$$Q(s, a) = r(s, a) + \gamma \sum_{s'} p(s' | s, a) V(s')$$

On a alors

$$\pi^*(s) = \operatorname{argmax}_a Q^*(s, a) \text{ et } V^*(s) = \max_a Q^*(s, a)$$

La fonction de Q-valeur



$$\begin{aligned} Q^*(s, a) &= r(s, a) + \gamma\{pV^*(s') + (1 - p)V^*(s'')\} \\ Q^*(s, a') &= r(s, a') + \gamma\{qV^*(s') + (1 - q)V^*(s'')\} \\ \pi^*(s) &= \operatorname{argmax}_{a, a'}\{Q^*(s, a), Q^*(s, a')\} \end{aligned}$$

Principe du Q-Learning

On reprend la *value iteration*

$$V_{n+1}(s) \leftarrow \max_a \overbrace{\{r(s, a) + \gamma \sum_{s'} p(s' | s, a) V_n(s')\}}^{Q_n(s, a)}$$
$$\Rightarrow Q_{n+1}(s, a) = r(s, a) + \gamma \sum_{s'} p(s' | s, a) V_{n+1}(s')$$
$$Q_{n+1}(s, a) \leftarrow r(s, a) + \gamma \sum_{s'} p(s' | s, a) \max_{a'} Q_n(s', a')$$

Après observation de la transition (s_n, a_n, s_{n+1}, r_n) , on estime pour le couple (s_n, a_n) le second membre par

$$r_n + \gamma \max_{a'} Q_n(s_{n+1}, a')$$

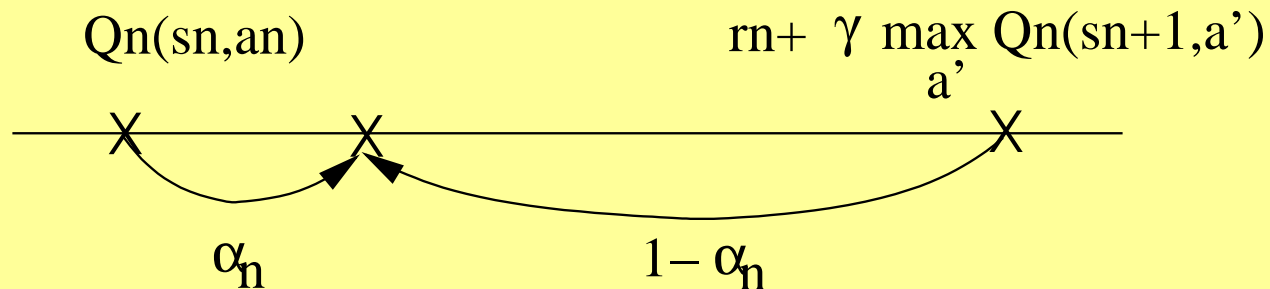
Algorithme du Q-learning

Après chaque expérience, ou transition,

$$(s_n, a_n, s_{n+1}, r_n)$$

On met à jour $Q_n(s_n, a_n)$:

$$Q_{n+1}(s_n, a_n) \leftarrow (1 - \alpha_n)Q_n(s_n, a_n) + \alpha_n(r_n + \gamma \max_{a'} Q_n(s_{n+1}, a'))$$



Convergence du Q-learning

- Pour une bonne décroissance de α_n vers 0 (exemple : $\frac{1}{n}$ ou $\frac{1}{n_{s,a}}$),
- et si on visite infiniment l'espace $S \times A$,
- alors Q-learning **converge** vers Q^* **presque sûrement** (avec une probabilité égale à 1).

Le dilemme Exploration / Exploitation

Le choix de l'état est souvent lié à la dynamique.

Pour l'action, une démarche optimiste consiste à choisir à chaque itération la meilleure action courante

$$a_n = \operatorname{argmax}_a Q_n(s_n, a)$$

Plus raisonnablement, il convient de régulièrement choisir une action aléatoire dans A .

On utilise pour cela des **fonctions d'exploration** dirigées ou non-dirigées

Fonctions d'exploration non-dirigées

- suivre Q_n pendant N_1 transitions, puis tirer uniformément dans A pendant N_2 transitions
- à chaque itération suivre Q_n avec une probabilité τ , ou tirer uniformément dans A avec une probabilité $1 - \tau$
- tirer a_n dans A selon

$$p_T(a) = \frac{e^{-\frac{Q_n(s_n, a)}{T}}}{\sum_{a'} e^{-\frac{Q_n(s_n, a')}{T}}}$$

avec $T \rightarrow \infty$

Fonctions d'exploration dirigées

On utilise l'information accumulée au cours de la recherche, autre que Q_n .

En pratique, on ajoute à Q_n un bonus d'exploration et on choisit la fonction qui maximise cette somme.

Exemple de bonus

- la *recency based method*: $Q_n(s, a) + \varepsilon \sqrt{T_{s,a}}$
 $T_{s,a}$ est le nombre d'itérations depuis le dernier choix de a dans s
- la *uncertainty estimation method*: $Q_n(s, a) + \varepsilon / n_{s,a}$

Retour sur le parking (p et r inconnus)

Après chaque transition $(i, G, F, -i)$

$$Q_{n+1}(i, G) \leftarrow Q_n(i, G) + \alpha_n(-i - Q_n(i, G))$$

Après chaque transition $(0, G, F, -C)$

$$Q_{n+1}(0, G) \leftarrow Q_n(0, G) + \alpha_n(-C - Q_n(0, G))$$

Après chaque transition $(i, \bar{G}, (i-1, L), 0)$

$$Q_{n+1}(i, \bar{G}) \leftarrow Q_n(i, \bar{G}) + \alpha_n(\max(Q_n(i-1, G), Q_n(i-1, \bar{G})) - Q_n(i, \bar{G}))$$

Après chaque transition $(i, \bar{G}, (i-1, \bar{L}), 0)$

$$Q_{n+1}(i, \bar{G}) \leftarrow Q_n(i, \bar{G}) + \alpha_n(Q_n(i-1, \bar{G}) - Q_n(i, \bar{G}))$$

Approche par itération de la politique

Dans le Q-learning, la politique apprise est directement liée à la fonction Q apprise.

Il est aussi possible de gérer explicitement une suite de politiques π_n et une suite de fonctions de valeur V_n . On suit alors le même principe que dans l'algorithme de *policy iteration* \Rightarrow *actor-critic methods*.

Principe des *actor-critic methods*

On maintient π_n (*action network*) et V_n (*critic network*)

Après chaque transition (s_n, a_n, s_{n+1}, r_n)

- V_n est mise à jour
- π_n est améliorée

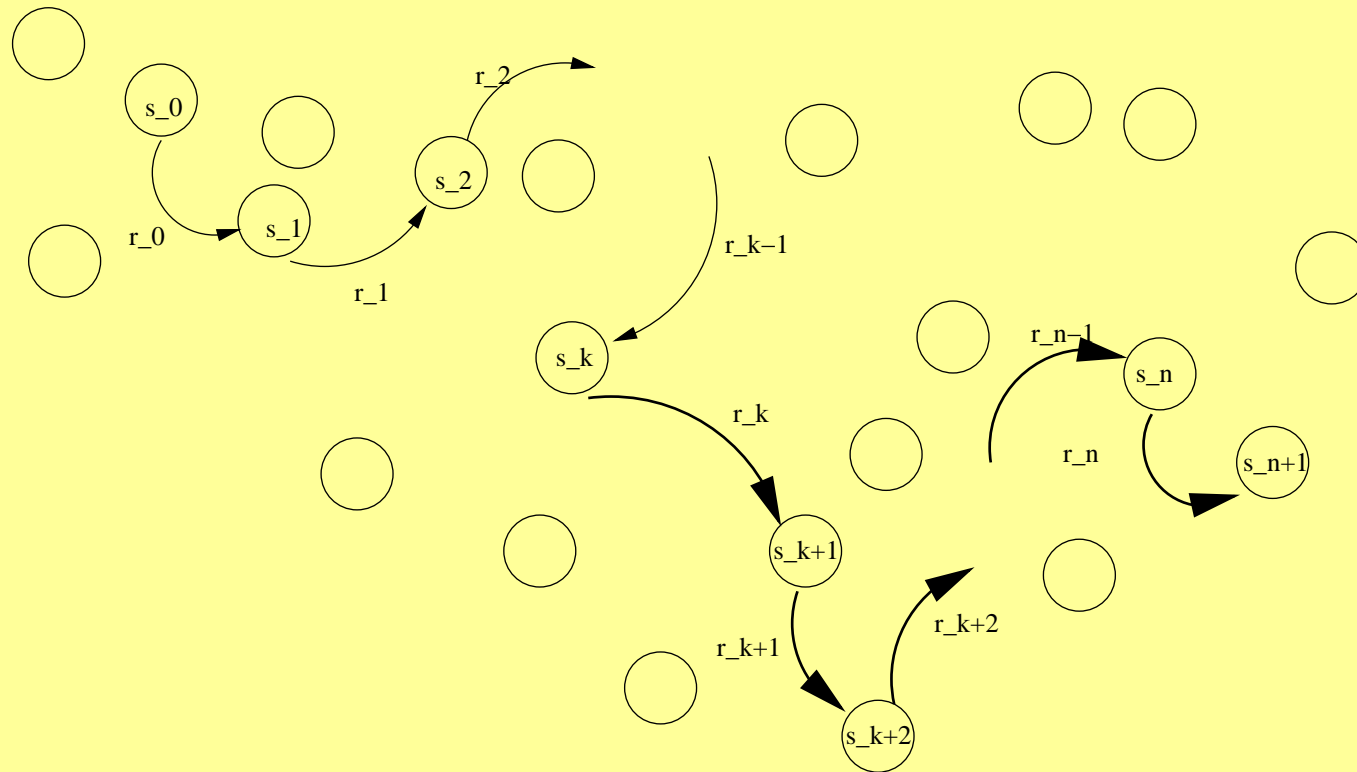
Evaluation approchée de la politique π_n

La phase d'évaluation de la politique courante π_n est entreprise à l'aide de la simulation.

Différentes méthodes permettent d'approcher V_{π_n}

- maximum de vraisemblance : calcul de V_{π_n} sur la base des estimées $p()$ et $r()$; peu efficace
- Monte Carlo
- Programmation Dynamique
- TD(λ)

Chaînes de Markov valuées



$$\langle S, p, r \rangle \Rightarrow V$$

Méthode de Monte Carlo

Dans le cadre $\gamma = 1$ avec état absorbant

Pour une politique π fixée

On observe (s_0, \dots, s_N) et (r_0, \dots, r_{N-1})

On met à jour les N valeurs $V(s_k)$ par

$$V(s_k) \leftarrow V(s_k) + \alpha(r_k + r_{k+1} + \dots + r_{N-1} - V(s_k))$$

Algorithmes off-line / on-line

L'approche off-line nécessite d'attendre la fin de chaque trajectoire pour mettre à jour V

On peut aussi modifier V après chaque transition (s_k, s_{k+1}, r_k) , en utilisant les **différences temporelles** :

$$r_k + r_{k+1} + \cdots + r_{N-1} - V(s_k) = d_k + d_{k+1} + \cdots + d_{N-1}$$

avec

$$d_k = r_k + V(s_{k+1}) - V(s_k)$$

D'où la règle

$$V(s_l) \leftarrow V(s_l) + \alpha d_k, \quad l = 0, \dots, k$$

La méthode de la programmation dynamique

On peut aussi chercher à résoudre stochastiquement le système d'équations linéaires définissant V :

$$V(s) = r(s, \pi(s)) + \sum_{s'} p(s' | s, \pi(s)) V(s')$$

Après chaque transition (s_k, s_{k+1}, r_k)

$$\begin{aligned} V(s_k) &\leftarrow V(s_k) + \alpha(r_k + V(s_{k+1}) - V(s_k)) \\ &\leftarrow V(s_k) + \alpha d_k \end{aligned}$$

La méthode des différences temporelles (TD(λ))

TD(λ) est un compromis entre les deux précédentes méthodes

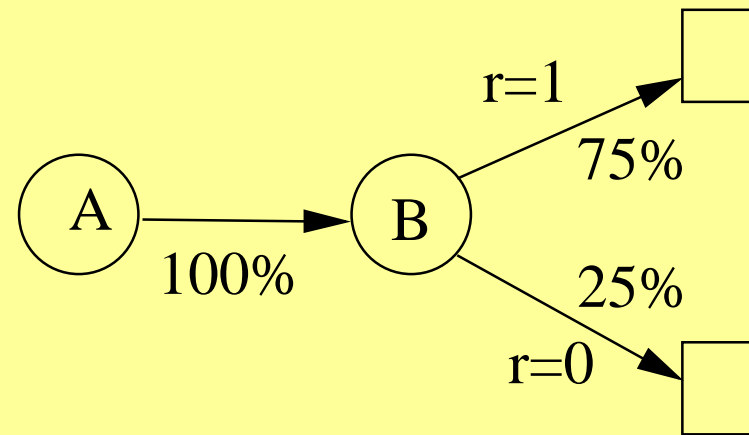
Pour une trajectoire observée (s_0, \dots, s_N) et (r_0, \dots, r_{N-1}) la fonction de valeur courante V est mise à jour selon

$$V(s_k) \leftarrow V(s_k) + \alpha \sum_{m=k}^{m=N-1} \lambda^{m-k} d_m, \quad k = 0, \dots, N - 1$$

- $\lambda = 0$: méthode de la programmation dynamique
- $\lambda = 1$: méthode de Monte Carlo

La convergence presque sûre est assurée.

Comparaison entre Monte Carlo et Programmation Dynamique (TD(0))



(A, 0, B, 0)

(B,1)

(B,1)

(B,1)

(B,1)

(B,1)

(B,1)

(B,0)

Que vaut $V(B)$? et $V(A)$?

Comparaison entre Monte Carlo et Programmation Dynamique (TD(0))

- $V(B) \approx \frac{3}{4}$
- $V(A) \approx 0$ pour Monte Carlo
- $V(A) \approx \frac{3}{4}$ pour TD(0)

Monte Carlo minimise l'erreur quadratique sur les observations.

TD(0) maximise la vraisemblance de l'estimation.

TD(λ) *on-line*

Après chaque transition (s_k, s_{k+1}, r_k) ,

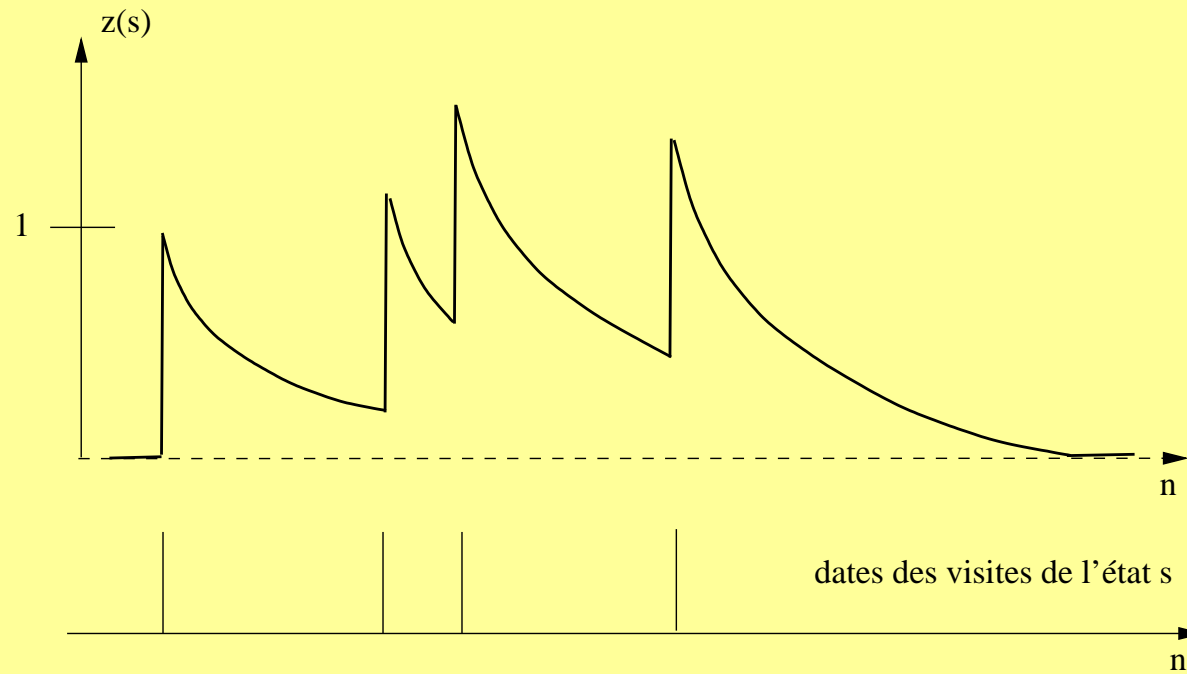
$$V(s_l) \leftarrow V(s_l) + \alpha \lambda^{k-l} d_k, \quad l = 0, \dots, k$$

On utilise surtout une forme similaire de cet algorithme utilisant la notion de *trace d'éligibilité*.

$$V(s) \leftarrow V(s) + \alpha z_k(s) d_k \quad \forall s \in S$$

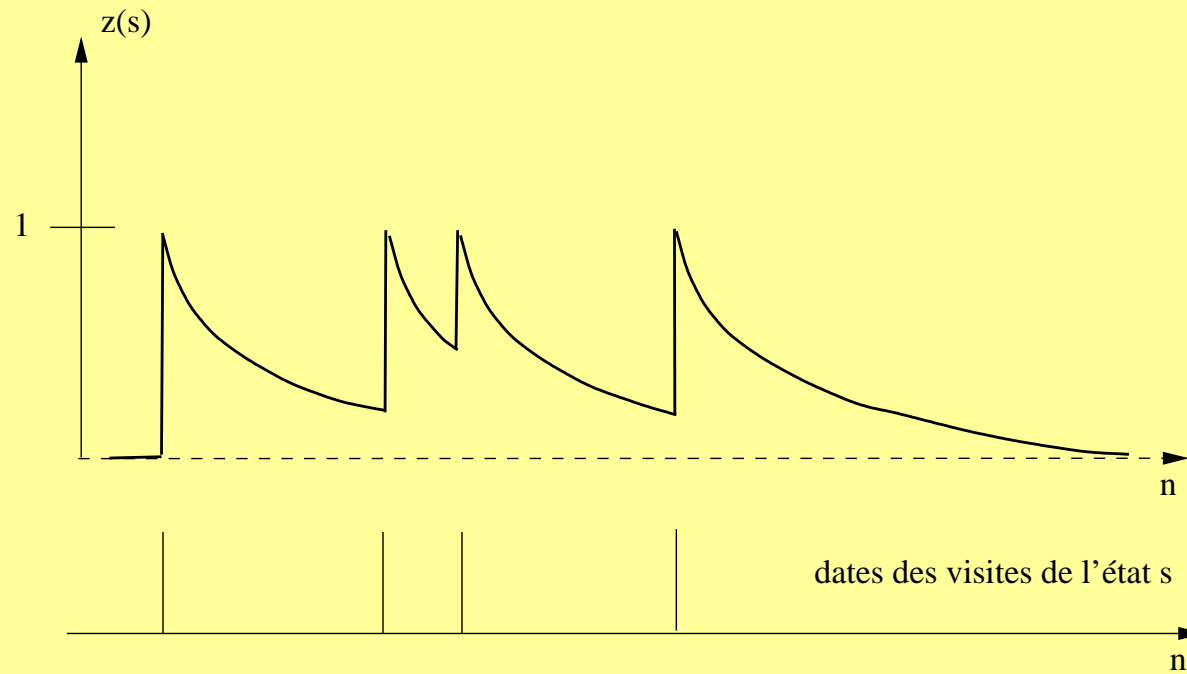
Trace d'éligibilité accumulative

$$z_0(s) = 0, \quad \forall s \in S$$
$$z_n(s) = \begin{cases} \gamma \lambda z_{n-1}(s) & \text{si } s \neq s_n \\ \gamma \lambda z_{n-1}(s) + 1 & \text{si } s = s_n \end{cases}$$



Trace d'éligibilité avec réinitialisation

$$z_0(s) = 0, \quad \forall s \in S$$
$$z_n(s) = \begin{cases} \gamma \lambda z_{n-1}(s) & \text{si } s \neq s_n \\ 1 & \text{si } s = s_n \end{cases}$$



TD(λ) et Q-learning

Si TD(λ) est utilisé dans les méthodes de type *policy iteration* pour évaluer une politique, il peut aussi être utilisé pour améliorer le Q-learning

Après chaque expérience, ou transition (s_n, a_n, s_{n+1}, r_n)

$$z_n(s, a) = \begin{cases} 0 & \text{si } a_n \neq \operatorname{argmax}_a Q_n(s_n, a) \\ \gamma \lambda z_{n-1}(s, a) & \text{si } a_n = \operatorname{argmax}_a Q_n(s_n, a) \end{cases}$$

$$z_n(s_n, a_n) \leftarrow z_n(s_n, a_n) + 1$$

$$Q_{n+1}(s, a) \leftarrow Q_n(s, a) + \alpha_n z_n(s, a) (r_n + \gamma \max_{a'} Q_n(s_{n+1}, a') - Q_n(s_n, a_n))$$

Représentation de la fonction de valeur

Pour des problèmes discrets de faible taille, il est possible de représenter V^π ou V^* par un tableau état:valeur (*look-up table*). L'apprentissage de V porte alors directement sur ses composantes.

Pour des problèmes de grande dimension ou à domaines continus, on utilise une représentation paramétrée

$$V = V_\xi$$

où ξ est un vecteur de paramètres de faible taille.

L'apprentissage porte alors sur ξ

$$\xi_{n+1} = \xi_n + \alpha \cdot \Delta(\xi_n, s_n, a_n, s_{n+1}, r_n)$$

Caractéristiques d'une architecture

la capacité d'approximation : on cherche à minimiser l'erreur intrinsèque $\epsilon = \min_{\xi} \|V - V_{\xi}\|$;

la capacité de généralisation : on recherche des ξ de faible dimension minimisant $\epsilon \Rightarrow$ les valeurs de plusieurs états seront simultanément modifiées avec la présentation d'un seul exemple ;

la simplicité de l'évaluation : selon la paramétrisation retenue, le calcul pour un état $s \in S$ de $V_{\xi}(s)$ peut être plus ou moins coûteux ;

l'efficacité de l'apprentissage : selon ξ , la convergence n'est plus toujours assurée, et les algorithmes peuvent devenir très complexes.

Différentes architectures d'approximation

- représentations différentiables
 - linéaires
 - réseaux neuronaux
 - ...
- représentations non différentiables
 - arbres de régression,
 - ...

Représentations différentiables

Il est naturel ici d'utiliser des méthodes de type **descente de gradient stochastique** pour apprendre le vecteur de paramètres ξ

$$\xi_{n+1} = \xi_n + \alpha_n (R_n - V_{\xi_n}(s_n)) \nabla_{\xi_n} V_{\xi_n}(s_n)$$

où R_n est l'estimation directe tirée de l'expérience de la valeur de V en s_n (cf. Monte Carlo, TD(0) et TD(λ)).

Dans le cas général, seul Monte Carlo assure que cette règle de mise à jour converge vers un optimum local pour

$$\epsilon(\xi) = \sum_{s \in \mathcal{S}} (V(s) - V_{\xi}(s))^2$$

Q-learning et représentations paramétrées

La fonction de valeur $Q(s, a)$ peut elle aussi être paramétrée.

Exemple de règle d'apprentissage :

$$\xi_{n+1} = \xi_n + \alpha_n (r_n + \gamma \max_a Q_n(s_{n+1}, a) - Q_n(s_n, a_n)) \nabla_{\xi_n} Q_{\xi_n}(s_n, a_n)$$

Plus trop de résultats de convergence ...

Les représentations linéaires

$$V_{\xi}(s) = \xi(1)\psi_1(s) + \dots + \xi(K)\psi_K(s)$$

où les $\psi_i(\cdot)$ sont K fonctions de S dans \mathbb{R}

$$\nabla_{\xi_n} V_{\xi_n}(s_n) = (\psi_1(s_n), \dots, \psi_K(s_n))^T$$

entraîne

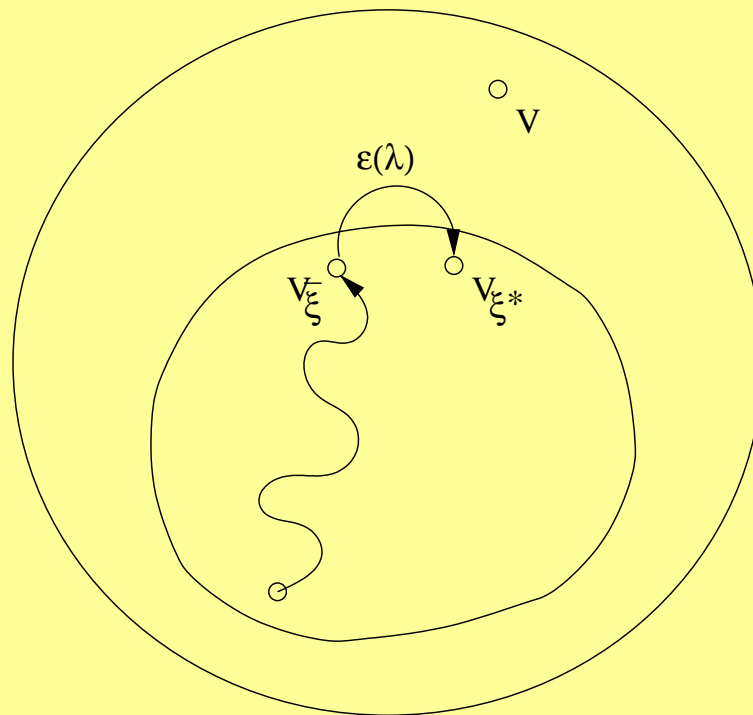
$$\xi_{n+1}(i) = \xi_n(i) + \alpha_n(R_n - V_{\xi_n}(s_n))\psi_i(s_n), \quad \forall i = 1, \dots, K$$

Il existe un optimum unique ξ^* pour l'erreur quadratique, et TD(λ) converge nécessairement (mais non forcément vers ξ^*)

TD(λ) et représentations linéaires

TD(λ) converge vers $\bar{\xi}$, à proximité de l'unique optimum ξ^* pour la norme quadratique $\sum_s (V(s) - V_\xi(s))^2$.

L'égalité $\bar{\xi} = \xi^*$ est atteinte pour $\lambda \rightarrow 1$.



Les fonctions de voisinage

On définit K régions \mathcal{R}_i qui forment ensemble une couverture de S , et on pose pour $i = 1, \dots, K$ $\psi_i(\cdot) = \mathbb{1}_{\mathcal{R}_i}(\cdot)$ la fonction indicatrice de \mathcal{R}_i

$$\forall s \in S \quad \psi_i(s) = \begin{cases} 1 & \text{si } s \in \mathcal{R}_i \\ 0 & \text{sinon} \end{cases}$$

Exemple : *state aggregation*, où les \mathcal{R}_i forment une partition de S .

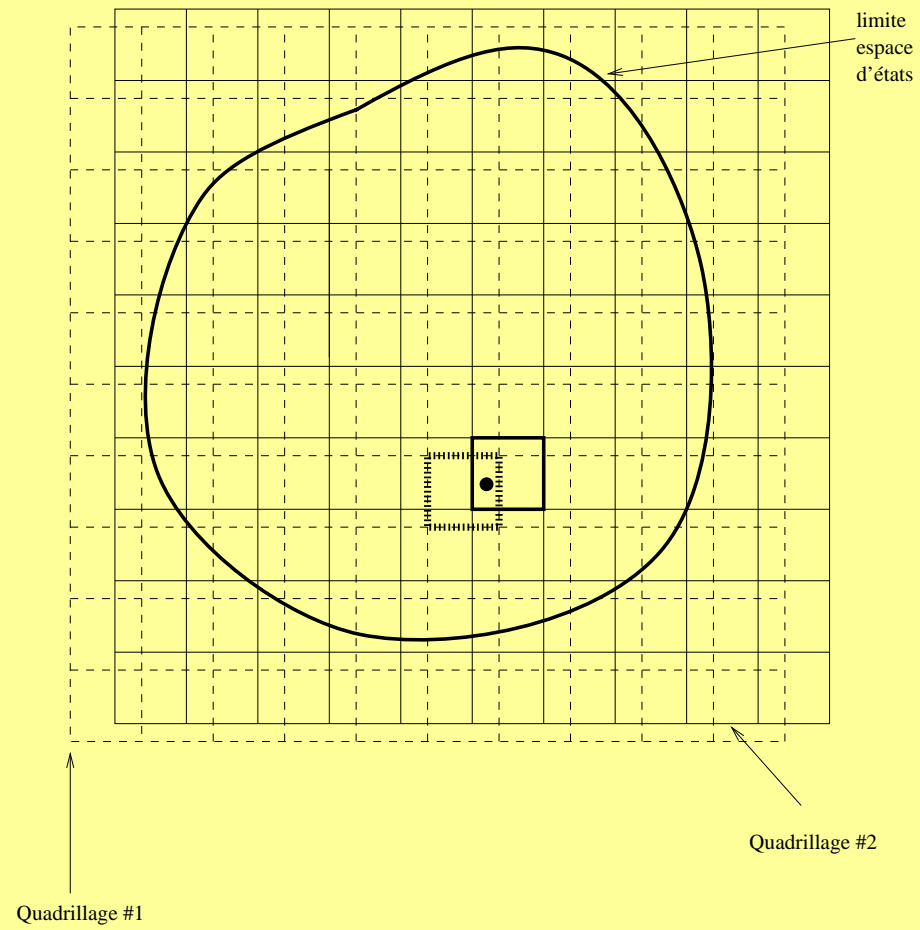
La méthode CMAC (*cerebellar model articulator controller*)

Particulièrement utilisée pour des espaces continus.

Principe : définir C partitions (ou grilles) de S , décalées géométriquement les unes par rapport aux autres

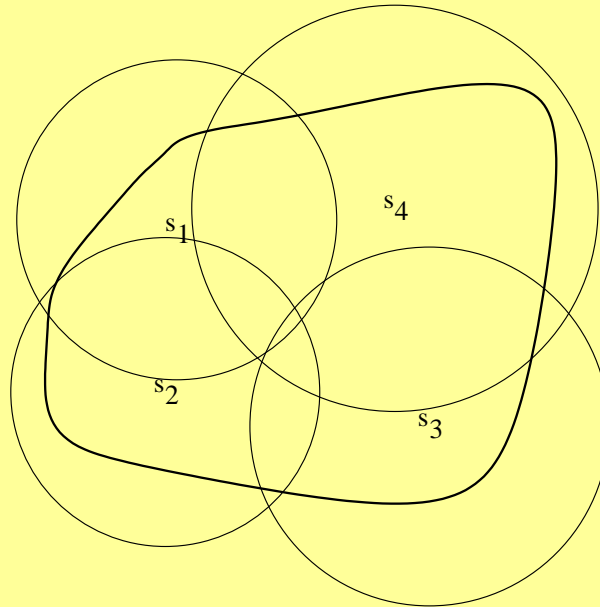
À chaque région de chaque grille, on associe un poids $\xi(i)$. La valeur d'un état s de S est la somme des C poids des régions de chaque grille qui le contiennent.

La méthode CMAC



Les méthodes à base d'états représentatifs

On place K points s_i dans S , centres d'une région



La forme de la région est fixe, les recouvrements sont possibles

Les méthodes à base d'états représentatifs

On peut aussi associer une valeur réelle dans l'intervalle $[0, 1]$ fonction de la distance au centre.

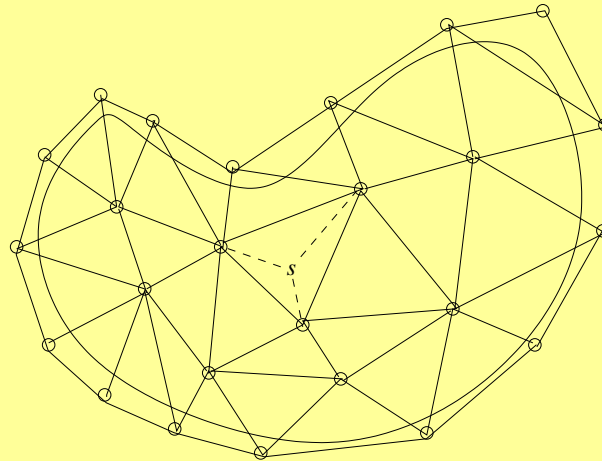
Exemple : les *radial basis functions*, ou RBF

$$\forall s \in S \quad \psi_i(s) = \exp\left(-\frac{\|s - s_i\|}{2\sigma_i^2}\right)$$

On peut alors apprendre aussi les meilleurs paramètres s_i et σ_i (architecture non-linéaire)

Triangularisation de l'espace d'états

On triangularise $S \subset \mathbb{R}^n$ par les points s_i



On définit la fonction de valeur $V_\xi(s)$ en tout point s comme la combinaison linéaire

$$V_\xi(s) = \lambda_{s_{i_0}}(s)V_\xi(s_{i_0}) + \cdots + \lambda_{s_{i_n}}(s)V_\xi(s_{i_n})$$

où les $\lambda_{s_{i_j}}(s)$ sont les coordonnées barycentriques du point s par rapport aux $n + 1$ points s_{i_0}, \dots, s_{i_n} du simplexe qui le contient.

Les fonctions coordonnées

On projette S dans un sous-espace \mathbb{R}^K , en définissant K nouvelles coordonnées fonctions des anciennes.

Les nouvelles coordonnées résument le mieux possible les propriétés d'un état vis-à-vis de la tâche à apprendre.

Exemple : Tétris. $h \times l$ variables d'états binaires $\Rightarrow 2l + 1$ coordonnées

- les hauteurs h_k des l colonnes
- les différences $|h_k - h_{k+1}|$
- la hauteur maximale du mur $\max_k h_k$
- le nombre de trous dans le mur

Les fonctions heuristiques

Permet d'exploiter la connaissance de stratégies répondant partiellement au problème

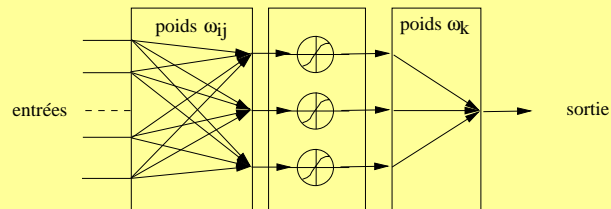
Les $\psi_i()$ sont les fonctions de valeur associées à des politiques obtenues par expertise ou par une première résolution approchée.

$$V_{\xi}(s) = \xi(1)V_{\pi_1}(s) + \dots + \xi(K)V_{\pi_K}(s)$$

Les V_{π_i} est estimées par simulation, et elles-même paramétrées

Le perceptron multi-couches

L'architecture non-linéaire la plus employée en apprentissage par renforcement.



Pour un état s défini par ses variables d'état $(s(1), \dots, s(i), \dots)$, la fonction de valeur $V(s)$ est approchée par

$$V_{\xi}(s) = \sum_k \omega_k \sigma \left(\sum_i \omega_{k,i} s(i) \right)$$

avec (par exemple)

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

Apprentissage du perceptron multi-couches

Le vecteur ξ des paramètres du perceptron est constitué des poids $\omega_i, \omega_{jk}, \dots$ du réseau.

L'apprentissage de ces poids est réalisé à partir de la règle de gradient.

Le calcul du terme de gradient $\nabla_{\xi_n} V_{\xi_n}$ est effectué par des techniques de rétro-propagation au sein du réseau.

Il existe de nombreux algorithmes d'apprentissage par renforcement couplant Q-learning et/ou TD(λ) avec un perceptron.

Apprentissage de structures non-différentiables

Toutes les structures des représentations précédentes doivent être adaptées à la tâche à apprendre.

Il existe quelques algorithmes d'apprentissage par renforcement cherchant à apprendre une telle résolution spatiale optimale.

Les méthodes de gradient ne sont plus adaptées

Méthodes à base d'arbres de régression

En entrée de l'arbre : les variables d'état ($s(1), \dots, s(i), \dots$)

En sortie : la fonction de valeur $V(s)$ approchée.

A chaque nœud interne de l'arbre correspond un test particulier sur la valeur d'un attribut, et les branches issues de ce nœud sont labellées par les différentes possibilités. Chaque feuille de l'arbre spécifie une valeur particulière pour la fonction V .

Différents algorithmes (*Parti-Game*, *G*, *U-tree*) proposent d'apprendre la structure de ces arbres.

Amélioration de la politique π_n

Lorsqu'un modèle est maintenu, l'algorithme d'itération de la politique peut être appliqué (*approximate policy iteration*)

$$\pi_n(s) = \operatorname{argmax}_a \left\{ r(s, a) + \gamma \sum_{s'} p(s' | s, a) V_n(s') \right\}$$

Ce calcul de π_n depuis V_n peut être très coûteux ou même impossible. Il est alors nécessaire de représenter

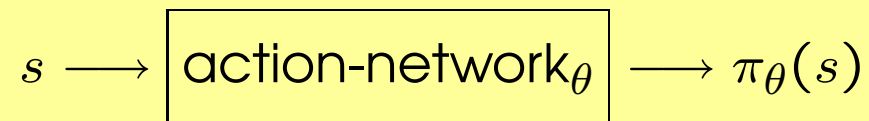
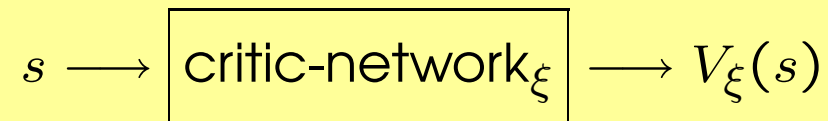
$$\pi_n(s) = \operatorname{argmax}_a Q_n(s, a)$$

par une fonction paramétrée (*action network*)

$$\pi_n = \pi_{\theta_n}$$

Algorithmes de type *actor-critic*

Paramétrisation de la politique π_{θ_n} (*action network*) séparément de la fonction de valeur V_{ξ_n} (*critic network*)



- mises à jour de ξ_n dans le sens de $V_{\pi_{\theta_n}}$
- mises à jour de θ_n dans le sens de π^*

Approches par gradient

Mise à jour du paramètre θ selon le gradient de V

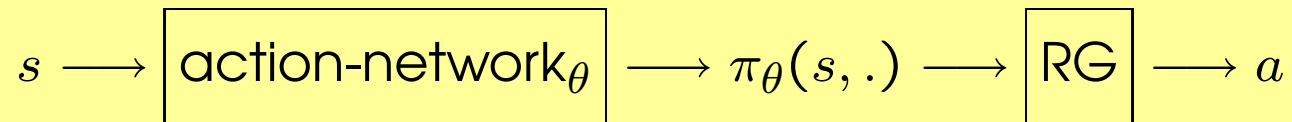
$$\theta_{n+1} = \theta_n + \alpha_n \hat{\nabla}_{\theta} V(\theta_n)$$

Estimation du gradient sur la chaîne de Markov

Si l'on considère des politiques stochastiques

$$\pi : S \times A \mapsto [0, 1]$$

où $\pi(s, a)$ représente la probabilité de choisir l'action a dans l'état s



le gradient de V vérifie

$$\nabla_{\theta} V = \langle Q_{\pi_{\theta}}, \nabla_{\theta} \ln \pi_{\theta} \rangle$$

Optimisation à base de simulation

Soit $R(\theta)$ le critère que l'on cherche à maximiser en moyenne.

$$\text{Par exemple } R(\theta) = \lim_{N \rightarrow \infty} r_0 + \gamma r_1 + \cdots + \gamma^N r_N$$

$$\theta \longrightarrow \boxed{\text{simulation de } \pi_\theta} \longrightarrow R(\theta)$$

On recherche

$$\theta^* = \underset{\theta}{\operatorname{argmax}} V(\theta) \text{ avec } V(\theta) = E[R(\theta)]$$

Différentes méthodes possibles ...

Algorithme de *Kiefer-Wolfowitz*

$$\nabla V(\hat{\theta}_n)_i = \frac{(R(\theta_n + \beta_n e_i) - R(\theta_n - \beta_n e_i))}{2\beta_n}$$

L'algorithme converge vers θ^* si α_n et β_n converge correctement vers 0.

Quelques conclusions

- les approches de type *actor-critic* semblent plus efficaces
- mieux vaut maintenir un modèle si on le peut
- on gagne à utiliser TD(λ) pour estimer une politique, ou pour améliorer le terme d'erreur du Q-learning

Quelques conclusions

Spécificités des méthodes d'AR :

- Les algorithmes convergent en probabilité
- les solutions obtenues sont des approximation des politiques optimales
- les applications abordées peuvent être complexes

Quelques conclusions

Les recherches actuelles portent sur :

- les propriétés de convergence des algorithmes classiques
- le développement des *actor-critic methods*
- l'emploi de représentations hiérarchisées
- le passage au cadre multi-agents
- etc.