Agent Oriented Learning

Apprentissage par renforcement & approximation

Stéphane Airiau

Université Paris-Dauphine

Quelques limitations

- backgammon : $\approx 10^{20}$ états
- les échecs : $\approx 10^{50}$ états
- Go $\approx 10^{170}$ états, 400 actions
- Robotique : beaucoup de degrés de liberté

avec un stockage dans des tables, on ne peut pas résoudre ces problèmes !

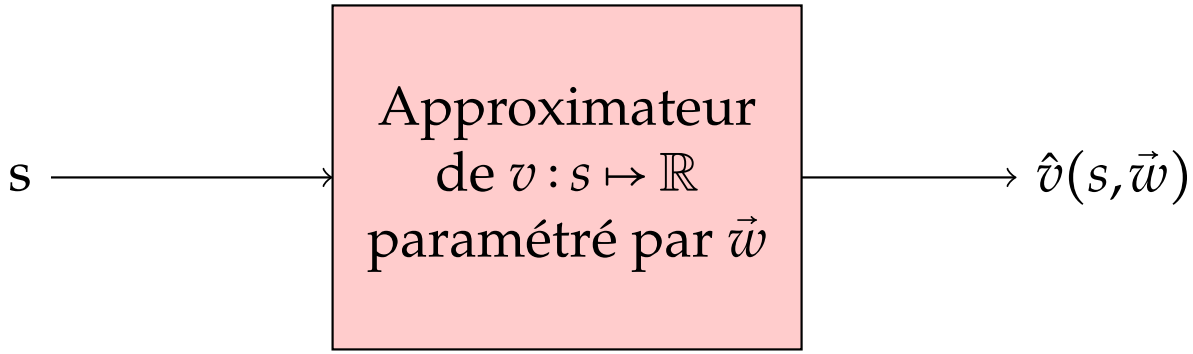
⇨ comment gérer des problèmes avec un grand nombre d'états et/ou d'action au niveau de la mémoire

⇨ comment va-t-on apprendre assez vite et généraliser sur des états/actions ?

Approximer la fonction de valeurs

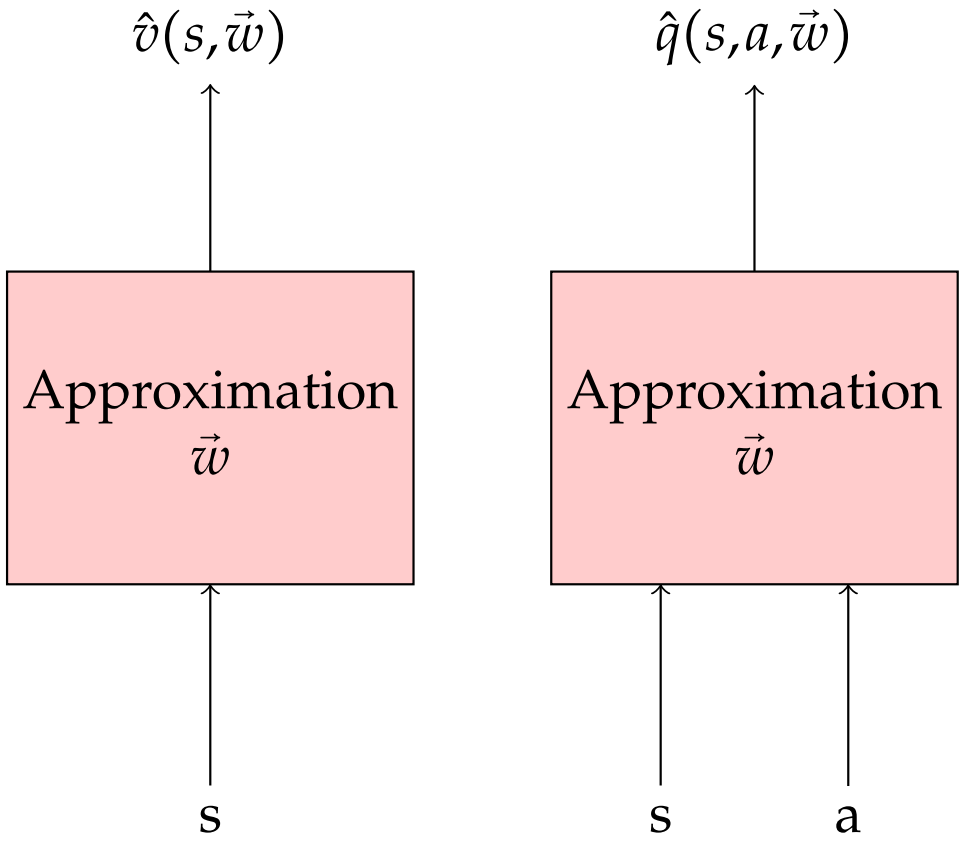
RL : on réalise une action a dans un état $s \in S$: on observe l'état suivant s' et on obtient la récompense r.

On peut voir RL + approximation comme un problème d'apprentissage supervisé : on observe s,a,s',r et on veut une fonction qui pour s donne une valeur à long terme u : on apprend la fonction $s \mapsto u$.



- Avec les méthodes tabulaires : on met à jour la valeur de s, et on ne change pas les valeurs pour les autres états.
- Avec l'approximation, on met à jour l'approximation
  ⇨ peut changer la valeur des autres états !

Type d'approximation de fonctions de valeurs



Quelle technique utiliser ?

- combinaison linéaire d'attributs
- réseau de neurones
- arbre de décision
- transformée de Fourier
- …

On va considérer les fonctions d'approximations qui sont différentiables.

Attention !

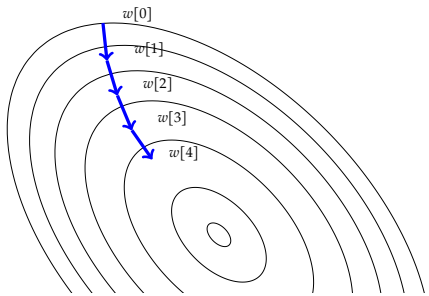
- nos données peuvent être non-stationnaires
  (i.e. elles peuvent dépendre du temps)
- elles **ne** sont **pas** i.i.d !

Descente de gradient

- Soit $J(\vec{w})$ une fonction différentiable par rapport au paramètre $\vec{w}$.
- Le gradient $\nabla J(\vec{w}) = \begin{pmatrix} \frac{\partial J(\vec{w})}{\partial w_1} \\ \vdots \\ \frac{\partial J(\vec{w})}{\partial w_n} \end{pmatrix}$
- Pour trouver un minimum local de J,
  on ajuste $\vec{w}$ dans la direction opposée au gradient

$$\Delta \vec{w} = -\frac{1}{2} \alpha \nabla J(\vec{w}),$$

où α est un paramètre qui mesure la taille du pas de la mise à jour.



Descente stochastique de Gradient

- $\vec{w} \in \mathbb{R}^d$ vecteur de poids
- $\hat{v}$ est notre approximation
- $\hat{v}(s,\vec{w})$ approxime l'état s et est differentiable par rapport à $\vec{w}$ pour tout $s \in S$.

⇨ on va mettre à jour $\vec{w}$ pour améliorer l'approximation à chaque étape.

- But : minimiser l'erreur moyenne

$$J(\vec{w}) = \mathbb{E}_\pi\left[(v_\pi(s) - \hat{v}(s,\vec{w}))^2\right]$$

- Le gradient est donc

$$\begin{aligned}\Delta\vec{w} &= -\frac{1}{2}\alpha\nabla J(\vec{w}) \\ &= \alpha\mathbb{E}_\pi\left[(v_\pi(s) - \hat{v}(s,\vec{w}))\nabla\hat{v}(S,\vec{w})\right]\end{aligned}$$

Gradient stochastique

Supposons qu'on ait accès à des exemples $(s, v_\pi(s))$. Une bonne stratégie est d'effectuer une descente de gradient : ajuster les poids dans la direction qui réduit le plus possible l'erreur sur ces exemples.

$$\begin{aligned} \vec{w}_{t+1} &= \vec{w}_t - \frac{1}{2}\nabla\left[v_\pi(s) - \hat{v}(s_t, \vec{w}_t)\right]^2 \\ &= \vec{w}_t + \alpha\left[v_\pi(s) - \hat{v}(s_t, \vec{w}_t)\right]\nabla\hat{v}(s_t, \vec{w}_t) \end{aligned}$$

où $\nabla f(\vec{w}) = \left(\frac{\partial f(\vec{w})}{\partial w_1}, \frac{\partial f(\vec{w})}{\partial w_2}, \ldots, \frac{\partial f(\vec{w})}{\partial w_d}\right)^\top$ est le gradient de f par rapport à $\vec{w}$.

On dit que le gradient est stochastique quand on fait une mise à jour à chaque example.

Attention : on pourrait chercher à minimiser l'erreur sur les exemples que l'on a observé, mais il faut trouver une bonne approximation pour **tout** l'espace !

Gradient stochastique

On ne connait pas $v_\pi(s)$. Par contre, on peut utiliser une de nos approximations via

- Monte Carlo : $u_t = G_t$
- $TD(0)$: $u_t = r_{t+1} + \gamma \hat{v}(s_{t+1}, \vec{w}_t)$
- programmation dynamique : $u_t = \mathbb{E}\left[r_{t+1} + \gamma \hat{v}(s_{t+1}, \vec{w}_t) | S_t = s\right]$

On aura alors :

$$\vec{w}_{t+1} = \vec{w}_t + \alpha \left[u_t - \hat{v}(s_t, \vec{w}_t)\right] \nabla \hat{v}(s_t, \vec{w}_t)$$

Si l'approximation u_t est non-biaisée, alors on a la garantie de convergence vers un minimum local avec des conditions classiques pour un α qui diminue.

C'est le cas pour Monte Carlo !

Gradient stochastique

Pour TD(0) et la programmation dynamique, l'estimation dépend de la valeur de $\vec{w}$, donc on perd le caractère non biaisé : le passage entre les deux expressions n'est pas valide !

$$\begin{aligned} \vec{w}_{t+1} &= \vec{w}_t - \frac{1}{2}\nabla\left[v_\pi(s) - \hat{v}(s_t, \vec{w}_t)\right]^2 \\ &= \vec{w}_t + \alpha\left[v_\pi(s) - \hat{v}(s_t, \vec{w}_t)\right]\nabla\hat{v}(s_t, \vec{w}_t) \end{aligned}$$

Mais on peut utiliser l'expression, ce ne sera pas le vrai gradient, mais cela permettra parfois convergence dans certains cas intéressants.

- on prend bien en compte l'effet du changement de $\vec{w}$ sur l'estimation de $\hat{v}$
- mais pas son effet sur la cible $u_t = r_{t+1} + \gamma\hat{v}(s_t, \vec{w}_t)$

⇨ on parle de "semi gradient"

Semi-gradient TD(0) pour estimer une politique π donnée

```
1  pour chaque épisode
2      s ← S_initial on part d'un état initial
3      tant que l'état s n'est pas terminal
4          choisir une action a à l'aide de π(s)
5          exécute l'action a, oberve l'état suivant s' et la récompense r
6          w⃗ ← w⃗ + α[r + γv̂(s', w⃗) − v̂(s', w⃗)] ∇v̂(s, w⃗)
7          s ← s'
```

On a donc "juste" besoin d'utiliser une fonction d'approximation dont on sait calculer le gradient.

⇨ on va donc regarder avec plus de détails les méthodes linéaires.

Cas particulier 1 : Méthodes linéaires

$\hat{v}(\cdot, \vec{w})$ est une fonction linéaire du vecteur de poids $\vec{w}$:

à chaque état s correspond un vecteur $x(s) = (x_1(s), \ldots, x_d(s))^\top$ $x(s)$ qui représente l'état s, où chaque x_i représente un "attribut" de l'état s. Par exemple

- la distance d'un robot par rapport à certaines cibles
- la présence de certaines configurations de pièces sur un échiquier

Méthodes linéaires

$$\hat{v}(s,\vec{w}) = \sum_{i=1}^{d} w_i x_i(s) = \vec{w}^{\top} \cdot \vec{x}(s).$$

Avec les méthodes linéaires, on veut que les attributs soient les bases de l'espace des états.

A cause de la linéarité, on obtient donc :

$$\nabla \hat{v}(s,\vec{w}) = x(s)$$

et ainsi :

$$\begin{aligned} \vec{w}_{t+1} &= \vec{w}_t + \alpha \left[u_t - \hat{v}(s_t,\vec{w}_t) \right] x(s_t) \\ &= \vec{w}_t + \alpha \left[u_t - \vec{w}_t^{\top} x(s_t) \right] x(s_t) \end{aligned}$$

Méthodes linéaires

- Pour le cas linéaire, l'optimum est unique et toutes les méthodes vont converger !
- Le gradient de Monte Carlo va converger (avec hypothèses sur la fonction α qui décroit au cours du temps) vers l'optimum.
- Le semi gradient $TD(0)$ converge, mais vers un point proche de l'optimal.

Semi gradient TD(0)

On note $x_t = x(s_t)$.

$$\begin{aligned}\vec{w}_{t+1} &\leftarrow \vec{w}_t + \alpha\left[r_{t+1} + \gamma \vec{w}_t^\top \vec{x}_{t+1} - \vec{w}_t^\top \vec{x}_t)\right]\vec{x}_t \\ &\leftarrow \vec{w}_t + \alpha\left[r_{t+1}\vec{x}_t - \vec{x}_t(\vec{x}_t - \gamma \vec{x}_{t+1})^\top \vec{w}_t\right]\end{aligned}$$

quand le système atteint un équilibre on obtient

$$\leftarrow \vec{w}_t + \alpha(b - A\vec{w}_t)$$

où $A = \mathbb{E}\left[\vec{x}_t(\vec{x}_t - \gamma\vec{x}_{t+1})^\top\right] \in \mathbb{R}^d \times \mathbb{R}^d$ et $b = \mathbb{E}\left[r_{t+1}\vec{x}_t\right]$

Si on converge vers $\vec{w}$, on doit avoir $(b - A\vec{w}) = 0$, donc $b = A\vec{w}$ et $\vec{w} = A^{-1}b$
La méthode de semi gradient converge vers $A^{-1}b$.

On peut démontrer la convergence et démontrer que cette solution est à une distance bornée de l'optimal.

Méthodes linéaires : constructions des bases

On peut utiliser des techniques classiques pour construire les bases :

- bases de polynomes
- séries de Fourier
- "codage grossier"
- codage en mosaïque
- fonctions de base radiale

Optimisation avec approximation de fonction

On vient de voir comment on peut évaluer une politique donnée.

On cherche cependant à trouver une politique optimale.

On utilise une nouvelle fois notre stratégie classique

- évaluation de la politique : approximation $\hat{q}(\cdot,\cdot,\vec{w})$
- amélioration de la politique : par exemple avec ϵ-glouton.

⇨ pas forcémment besoin d'utiliser trop d'échantillons pour trouver la bonne approximation $\hat{q}$

SARSA semi gradient pour estimer $q^\star$

State–action–reward–state–action (SARSA)

Initialise $\hat{q} : S \times A \times \mathbb{R}^d \to \mathbb{R}$ arbitrairement
Répète (éternellement) pour chaque épisode
    aller à un état initial s
    choisir action $a \in A$ pour s à l'aide d'une politique dérivée de $\hat{q}(s,\cdot,\vec{w})$ (ex : ϵ-greedy
    Répète pour chaque étape de l'épisode
        Exécute action a, observe $r \in \mathbb{R}$ et état suivant $s' \in S$
        choisir action $a' \in A$ pour s' à l'aide d'une politique dérivée de $\hat{q}(s',\cdot,\vec{w})$
        $\vec{w} \leftarrow \vec{w} + \alpha[r + \gamma\hat{q}(s',a',\vec{w}) - \hat{q}(s,a,\vec{w})]\nabla\hat{q}(s,a,\vec{w})$
        $s \leftarrow s'$
        $a \leftarrow a'$
    jusqu'à ce que s soit terminal
    $\vec{w} \leftarrow \vec{w} + \alpha[r - \hat{q}(s,a,\vec{w})]\nabla\hat{q}(s,a,\vec{w})$

Résultats de Convergence pour l'évaluation de politique

Pour Monte Carlo, l'estimation est non biaisée et on a des garanties de convergence.

Pour les méthodes TD, on peut construire des exemples dans lesquels les poids divergent ! Cependant en pratique, cela marche très souvent.

- TD ne suit pas le gradient d'une fonction objective !
- TD peut donc diverger avec des méthodes off policy ou en utilisant des approximations de fonctions non linéaires.
- certains nouveaux algorithmes arrivent à corriger le gradient pour assurer la convergence

On/Off Policy	Algorithme	Tabulaire	Linéaire	non-linéaire
On-Policy	Monte Carlo	✔	✔	✔
	TD(0)	✔	✔	✘
Off-Policy	Monte Carlo	✔	✔	✔
	TD(0)	✔	✘	✘

Résultats de Convergence pour l'optimisation

Algorithme	Tabulaire	Linéaire	non-linéaire
Monte Carlo	✔	(✔)	✘
Sarsa	✔	(✔)	✘
Q-learning	✔	✘	✘

(✔) on peut osciller autour de la solution optimale

Batch RL – Prédiction

- Descente de gradient est simple
- mais elle n'est pas efficace du point de vue du nombre d'échantillons avant convergence
- les méthodes avec batch cherche à améliorer cela
- utilise l'expérience de l'agent pour former un ensemble d'échantillons et apprendre via ces données

Batch RL – Prédiction avec Moindres carrés

- On enregistre des données $D = \left\{(s_1, v_1^\pi), (s_2, v_2^\pi), \ldots, (s_T, v_T^\pi),\right\}$
- On utilise la technique des moindres carrés pour chercher le vecteur $\vec{w}$ qui minimise l'erreur entre l'approximation et la cible v_π.

⇨ trouver les poids qui minimise l'erreur pour toutes les données que j'ai observé.

$$\begin{aligned} LS(\vec{w}) &= \sum_{t=1}^{T} (v_t^\pi - \hat{v}(s_t, \vec{w}))^2 \\ &= \mathbb{E}_D\left[(v^\pi - \hat{v}(s, \vec{w}))^2\right] \end{aligned}$$

Stochastic gradient descent with experience replay – Prédiction

On répète

1. on tire des échantillons dans la mémoire de l'agent $\{s, v^\pi\} \approx D$
2. On utilise la descente stochastique de gradient

$$\Delta\vec{w} = \alpha(v^\pi - \hat{v}(s,\vec{w}))\nabla\hat{v}(s,\vec{w})$$

converge vers la solution des moindres carrés

$$\vec{w}^\pi = \underset{\vec{w}}{\operatorname{argmin}} LS(\vec{w})$$

En tirant des échantillons au hasard, et seulement une partie, on se "rapproche" de données i.i.d, et donc on améliore la convergence.

Experience replay et Deep Q-networks – Prédiction

DeepQN utilise experience replay

1. choisier a_t avec ϵ-glouton
2. enregistrer la transition $(s_t, a_t, r_{t+1}, s_{t+1})$ dans la mémoire D
3. tirer des transitions $(s, a, r, s') \in D$ (mini batch)
4. calculer les valeurs de Q avec les anciens paramètres $\vec{w}$
5. optimise l'erreur moyenne quadratique entre Q_{old} et Q que l'on est en train de modifier

$$L_i(w_i) = \mathbb{E}_{s,a,r,s' \approx D_i}\left[\left(r + \gamma \max_{a'} Q(s', a'; w_i^{old}) - Q(s, a; w_i)\right)^2\right]$$

avec une descente stochastique de gradient

Permet d'apprendre à jouer à des jeux sous Atari

Agent Oriented Learning

Policy gradient

Stéphane Airiau

Université Paris Dauphine

Apprendre directement une politique

- On a approximé la fonction de valeurs ou la fonction des actions.
- Jusqu'ici, on a généré nos politiques directement par les fonctions de valeurs.
  - ⇨ en utilisant ϵ-glouton
  - ⇨ en utilisant softmax
- Aujourd'hui, on va directement approximer la **politique**
  - ⇨ travailler directement sur la politique sans nécessairement estimer et utiliser v ou q.
  - ⇨ apprendre directement la politique
    - la fonction de valeur peut être parfois compliquée, mais la politique peut être simple !

Pros & Cons

avantages

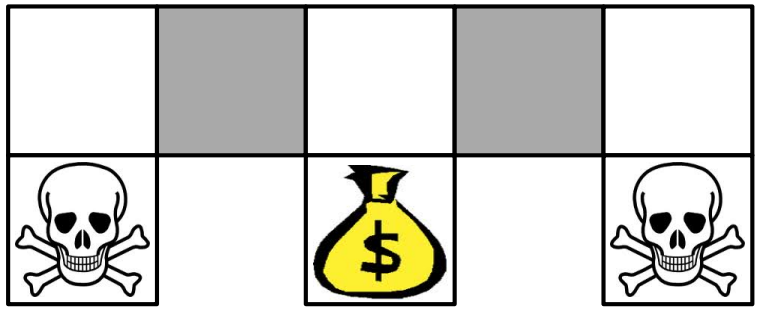
- meilleures propriétés de convergences
- travaillent aussi avec des espaces d'actions continues
  problématique si on doit trouver l'action qui est le max.
- peut apprendre un politique stochastique
  - parfois c'est la meilleure chose : pierre/feuille/ciseaux
  - dans certains cas particuliers ou on n'arrive pas à determiner dans quel état on se trouve (on peut être incertain sur l'état à cause des features)

désavantages

- converge vers un minimum local, pas nécessairement global
- pas forcément efficace et avec hautes variances

Exemple où une politique stochastique peut être précieuse

- jouer à pierre-feuille-ciseaux
- quand les attributs ne permettent pas de différentier des états



  - supposons que nos attributs soient du style

$$x_i(s) = 1_{\text{mur au Nord si l'action est Est}}$$

  - les deux états en gris ne peuvent pas être distingués
  - ⇨ pour une politique deterministe, on prendra la même action dans les deux cas
  - dans certains cas, ce ne serait pas souhaitable !

On a bien un PDM sous jacent, mais notre représentation peut le transformer en PDMPO : on peut être incertain sur l'état du monde.

Méthodes basées sur le gradient de la politique

- on va utiliser une fonction objectif qui mesure la qualité de la politique
- on va utiliser une approximation paramétrée de la politique
- ⇨ on va apprendre les paramètres de la politique pour améliorer la valeur de la fonction objectif
- plusieurs méthodes d'optimisations sont possibles
  - hill climbing
  - algorithmes génétiques
  - descente de gradient
  - gradients conjugués
  - …
  - ⇨ on va se concentrer sur la descente de gradients

Fonctions objectifs

- on approxime la politique $\pi(s,a) = \hat{\pi}_\theta(s,a)$ avec comme paramètres le vecteur θ
- On veut trouver les paramètres qui maximise un objectif $J(\theta)$
- Parcourir un PDM via une politique π revient à avoir une chaîne de Markov avec récompense.

⇨ on peut calculer la proportion de temps $d_{\hat{\pi}_\theta}(s)$ passé dans chaque état en moyenne

Fonctions objectifs

Fonctions possibles :

- Dans des PDMs épisodiques, on peut utiliser la valeur de l'état initial

$$J_1(\theta) = \mathbb{E}_{\hat{\pi}_\theta}(s_1)$$

- Pour des environnements non épisodiques, on peut utiliser la valeur moyenne

$$J_{avT}(\theta) = \sum_{s \in S} d_{\hat{\pi}_\theta}(s) v_{\hat{\pi}_\theta}(s)$$

- Pour des environnements non épisodiques, on peut utiliser la valeur moyenne par étape

$$J_{avS}(\theta) = \sum_{s \in S} \left[d_{\hat{\pi}_\theta}(s) \sum_{a \in A} \hat{\pi}_\theta(s,a) R_s^a \right]$$

Montée de gradient

On cherche à maximiser notre fonction objectif, donc on va essayer de faire une montée de gradient.

NB : dans les réseaux de neurones, on cherche à minimiser l'erreur, donc faire une descente de gradient !

donc il ne faut pouvoir calculer le gradient
⇨si on ne peut pas le calculer exactement, on peut l'estimer à l'aide de différences finies.

$$\frac{\partial J(\theta)}{\partial \theta_k} \approx \frac{J(\theta + \epsilon u_k) - J(\theta)}{\epsilon}$$

où u_k est le vecteur avec 1 pour l'entrée k, et 0 pour les autres.
⇨il faut pouvoir estimer ces quantités facilement...

Astuce

- Supposons qu'on ait accès au gradient
- Supposons que la politique $\hat{\pi}_\theta$ soit différentiable

⇨ on a $\nabla_\theta \hat{\pi}_\theta(s,a)$

$$\begin{aligned} \nabla_\theta \hat{\pi}_\theta(s,a) &= \hat{\pi}_\theta(s,a) \frac{\nabla_\theta \hat{\pi}_\theta(s,a)}{\hat{\pi}_\theta(s,a)} \\ &= \hat{\pi}_\theta(s,a) \nabla_\theta log\, \hat{\pi}_\theta(s,a) \end{aligned}$$

Cas simple

- On considère un PDM épisodique de longueur 1.
- la probabilité de commencer dans l'état s est donnée par $d(s)$
- On obtient donc la récompense $r = R_s^a$

$$
\begin{aligned}
J(\theta) &= \mathbb{E}_{\hat{\pi}_\theta}[r] \\
&= \sum_{s \in S} \left[d(s) \sum_{a \in A} \pi_\theta(s,a) R_s^a \right]
\end{aligned}
$$

$$
\begin{aligned}
\nabla_{\hat{\pi}_\theta} J(\theta) &= \sum_{s \in S} \left[d(s) \sum_{a \in A} \hat{\pi}_\theta(s,a) \nabla_\theta log \hat{\pi}_\theta(s,a) R_a^a \right] \\
&= \mathbb{E}_{\hat{\pi}_\theta} \left[\nabla_\theta log \hat{\pi}_\theta(s,a) \cdot r \right]
\end{aligned}
$$

L'astuce permet de voir ce gradient comme une espérance !

Théorème du gradient de la politique

- Il faut passer aux PDMs à plusieurs transitions...
- Intuition : on remplace la récompense immédiate r avec la fonction de valeurs à long terme des actions $q_\pi(s,a)$
- Le théorème s'applique pour les trois fonctions objectifs

Théorème

- Pour toute politique différentiable $\hat{\pi}_\theta$
- Pour une des fonctions objectifs

Le gradient de la politique est

$$\nabla_{\hat{\pi}_\theta} J(\theta) = \mathbb{E}_{\hat{\pi}_\theta}\left[\nabla_\theta \log \hat{\pi}_\theta(s,a) q_{\hat{\pi}_\theta}(s,a)\right]$$

On peut donc utiliser des échantillons pour calculer le gradient !

Gradient de la politique Monte Carlo (REINFORCE)

Pour des PDS épisodiques – Très similaire à Monte Carlo.

- on met à jour les paramètres par une montée stochastique de gradient
- on utilise le théorème du gradient de la politique
- en utilisant la valeur G_t comme échantillon de la valeur de $q_{\hat{\pi}_\theta}(s,a)$

```
1 | Initialise θ arbitrairement
2 | On exécute un épisode ⟨s_1,a_1,s_2,a_2,...,s_{T-1},a_{T-1},s_T⟩
3 | Pour t = 1 à T − 1
4 |         θ ← θ + α∇_θ log π̂_θ(s,a) G_t
5 | return θ
```

Un des inconvénients majeurs est que cette méthode est très lente !

Méthodes acteur-critiques

Au lieu d'utiliser le gain comme dans les méthodes de Monte Carlo, on voudrait utiliser $Q(s',a')$ comme dans les méthodes TD.

Comme on ne peut pas utiliser une table $Q(s,a)$ on va utiliser une fonction d'approximation pour approximer Q comme dans le cours précédent !

On combine l'idée de

- gradient sur la politique
  avec
- approximer la fonction de valeur $\hat{Q}(s,a,w)$

Méthodes acteur-critiques

Au lieu d'utiliser le gain comme dans les méthodes de Monte Carlo, on voudrait utiliser $Q(s',a')$ comme dans les méthodes TD.

Comme on ne peut pas utiliser une table $Q(s,a)$ on va utiliser une fonction d'approximation pour approximer Q comme dans le cours précédent !

On combine l'idée de

- gradient sur la politique ⇨ **acteur**
  avec
- approximer la fonction de valeur $\hat{Q}(s,a,w)$ ⇨ **critique**

Amélioration de l'estimation de Q : utilisation d'un critique

- on utilise une approximation de $q_{\hat{\pi}_\theta}(s,a)$

$$\hat{q}_w(s,a) \approx q_{\hat{\pi}_\theta}(s,a)$$

- On va donc avoir deux ensembles de paramètres
  - w pour paramétrer l'approximation de q : critique
  - θ pour paramétrer l'approximation de la politique : acteur
  - ⇨ méthode acteur-critique
- Les algorithmes acteur-critiques suivent donc un gradient approché de la politique

$$\nabla_{\hat{\pi}_\theta} J(\theta) \approx \mathbb{E}_{\hat{\pi}_\theta}\left[\nabla_\theta \log \hat{\pi}_\theta(s,a)\hat{q}_w(s,a)\right]$$

$$\Delta\theta = \alpha \nabla_\theta \log \hat{\pi}_\theta(s,a)\hat{q}_w(s,a)$$

Evaluation de la politique

- on a maintenant des algorithmes pour évaluer une politique
  - Monte Carlo
  - TD
  - Moindres carrés
  - …

Action-Value Actor-Critic

- On utilise une approximation linéaire pour $q_w(s,a) = x \cdot w$
- Critique : mise à jour des poids w avec TD(0)
- Actor : mise à jour des poids θ avec la montée de gradient

1 on choisit une action avec la politique $\hat{\pi}_\theta$
2 A chaque étape
3 On exécute l'action a, on obtient la récompense r et on observe l'état suivant s'
4 On choisit l'action a' avec $\hat{\pi}_\theta(s',a')$
5 $\delta = r + \gamma q_w(s',a') - q_w(s,a)$
6 $\theta = \theta + \alpha \nabla_\theta \log \hat{\pi}_\theta(s,a) q_w(s,a)$
7 $w \leftarrow w + \beta \delta \phi(s,a)$
8 $a \leftarrow a'$, $s \leftarrow s'$

Conclusion

- on a maintenant une méthode qui marche pour un grand nombre d'états et un grand nombre d'actions !
- fonctionne aussi pour les PDMs non épisodiques

⇨ on a un outil général pour résoudre un problème d'apprentissage par renforcement.