Apprentissage par renforcement

Cours 5 : Approximation de la fonction de valeurs

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 !

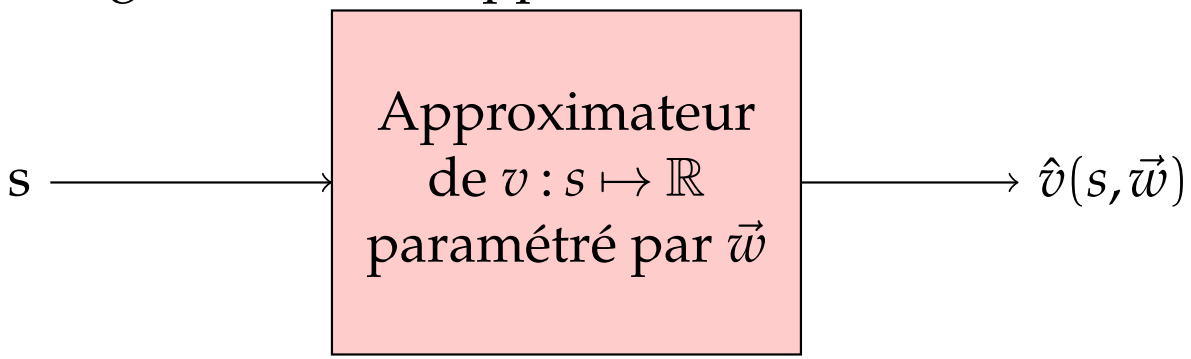
⇨ comment gérer des problèmes avec un grand nombre d'états et/ou d'action au niveau de la mémoire

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

Approximer la fonction de valeurs

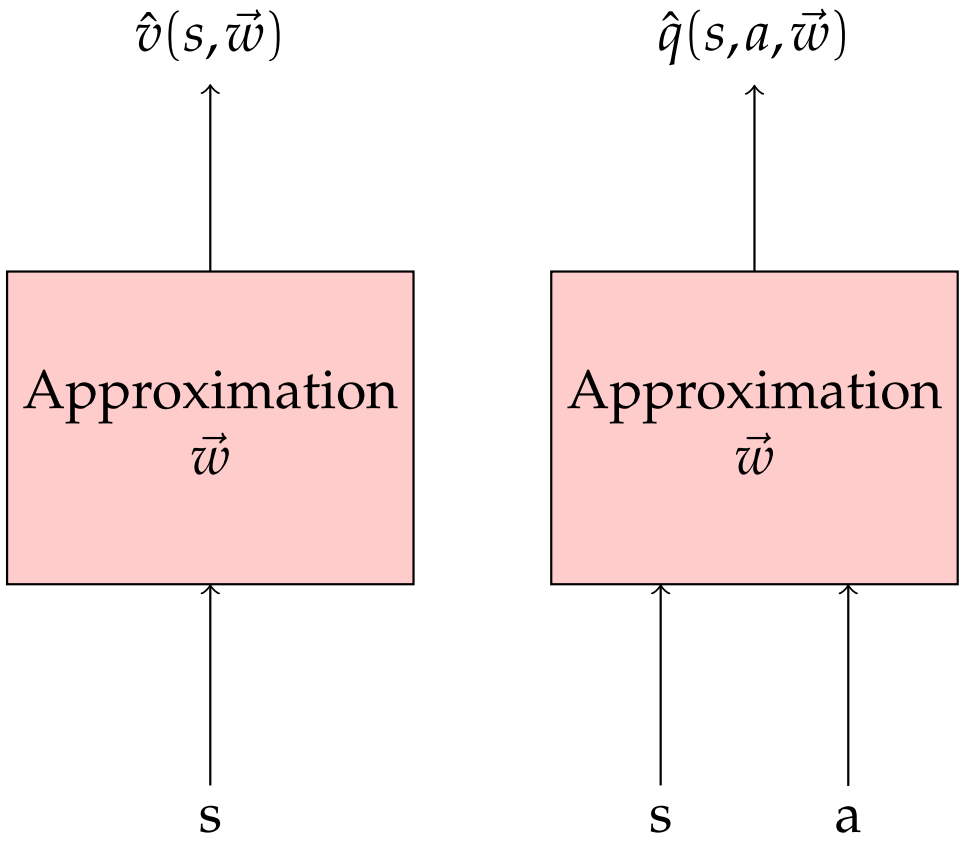
RL : on réalise une action a dans un état $s \in S$: on observe l'état suivant s' et on obtient la récompense r.

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 !

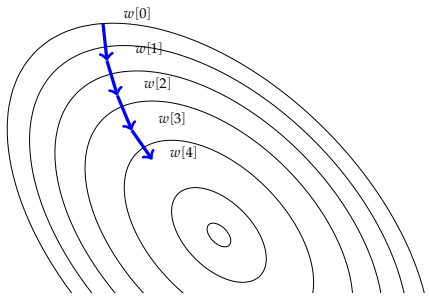
- nos données peuvent être non-stationnaires
  (i.e. elles peuvent dépendre du temps)
- elles **ne** sont **pas** i.i.d !

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 **fixée**

```
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          w ← w + α[r + γ v̂(s', w) − v̂(s, w)] ∇v̂(s, w)
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

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

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 :

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

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.

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.

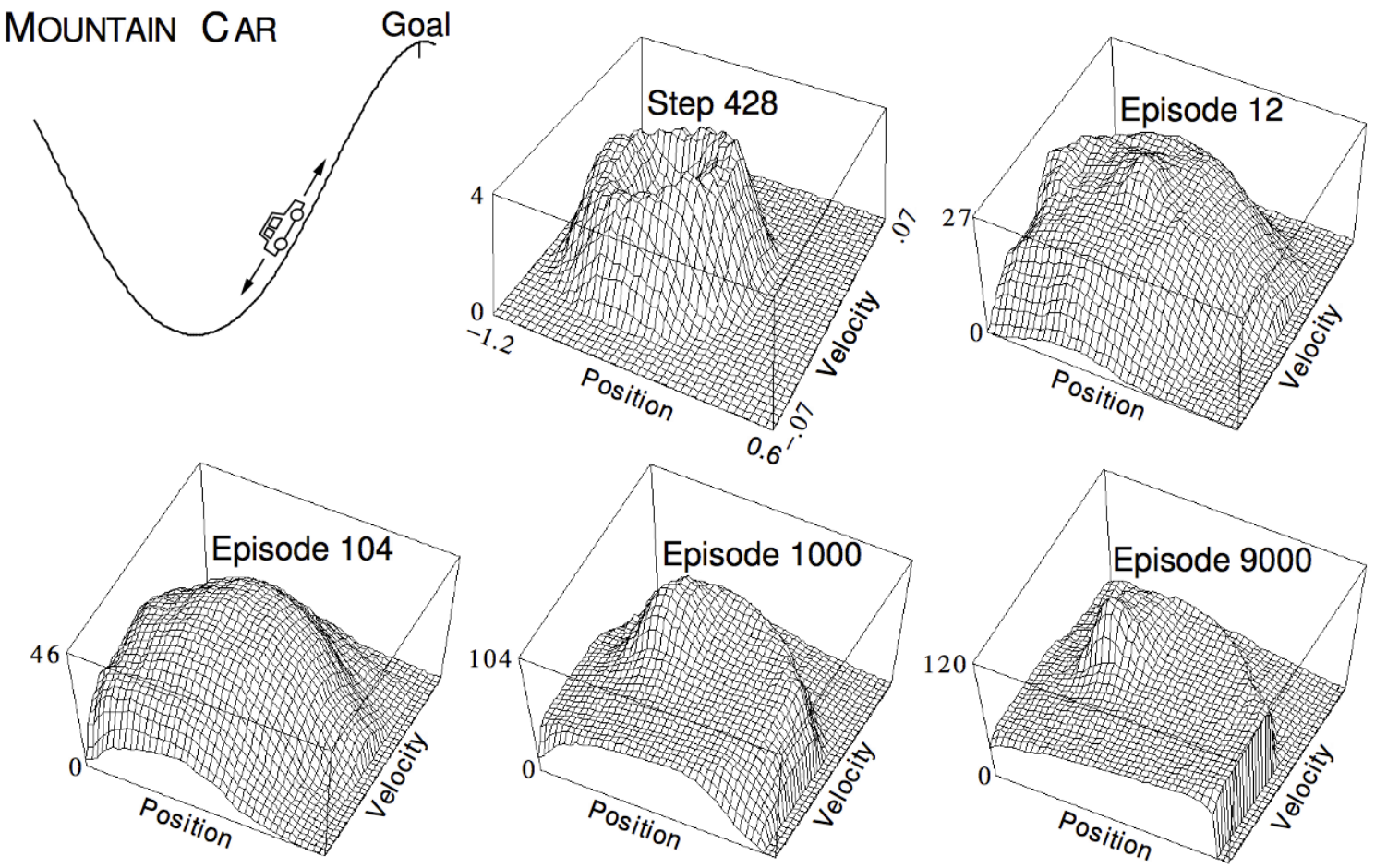
⇨ pas forcémment besoin d'utiliser trop d'échantillons pour trouver la bonne approximation $\hat{q}$

SARSA semi gradient pour estimer $q^\star$

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

Initialise $\vec{w} \in \mathbb{R}^d$ arbitrairement
Répète (éternellement) pour chaque épisode
    aller à un état initial s
    choisir action $a \in A$ pour s avec ϵ-greedy et $\hat{q}(s,\cdot,\vec{w})$
    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' avec ϵ-greedy et $\hat{q}(s',\cdot,\vec{w})$
        si s' est final
            $\vec{w} \leftarrow \vec{w} + \alpha\,[r - \hat{q}(s,a,\vec{w})]\,\nabla\hat{q}(s,a,\vec{w})$
        sinon
            $\vec{w} \leftarrow \vec{w} + \alpha\,[r + \gamma\hat{q}(s',a',\vec{w}) - \hat{q}(s,a,\vec{w})]\,\nabla\hat{q}(s,a,\vec{w})$
        $s \leftarrow s'$
        $a \leftarrow a'$
    jusqu'à ce que s soit terminal

Exemple : SARSA linéaire (coarse coding)



SARSA semi gradient pour estimer $q^\star$ – version neural networks

Initialise $\vec{w} \in \mathbb{R}^d$ arbitrairement, fixer N constant : taille du batch, $i = 0$
Répète (éternellement) pour chaque épisode
    aller à un état initial s_i
    choisir $a_i \in A$ pour s avec ϵ-greedy et $\hat{q}(s_i, \cdot, \vec{w})$
    Répète pour chaque étape de l'épisode
        Exécute action a_i, observe $r_i \in \mathbb{R}$ et état suivant $s'_i \in S$
        choisir action $a'_i \in A$ pour s'_i avec ϵ-greedy et $\hat{q}(s'_i, \cdot, \vec{w})$
        si s'_i est final
            $y_i \leftarrow r$
        sinon
            $y_i \leftarrow r + \gamma \hat{q}(s - i', a'_i, \vec{w})$
        $s_{i+1} \leftarrow s'_i$, $a_{i+1} \leftarrow a'_i$, $i \leftarrow i + 1$
        si $i == N$
            $\mathcal{L}(\vec{w}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - Q(s_i, a_i))^2$ // calcul de la loss
            $\vec{w} \leftarrow \vec{w} \alpha \nabla_{\vec{w}} \mathcal{L}(\vec{w})$ // mise à jour du réseau
            $i \leftarrow 0$
    jusqu'à ce que s soit terminal

Deep Q-networks

Initialise $\vec{w} \in \mathbb{R}^d$ arbitrairement, fixer N constant : taille du batch, $i = 0$

Répète (éternellement) pour chaque épisode

    aller à un état initial s_i

    choisir $a_i \in A$ pour s avec ϵ-greedy et $\hat{q}(s_i, \cdot, \vec{w})$

    Répète pour chaque étape de l'épisode

        Exécute action a_i, observe $r_i \in \mathbb{R}$ et état suivant $s'_i \in S$

        si s'_i est final

            $y_i \leftarrow r$

        sinon

            $y_i \leftarrow r + \gamma \max_{a''} \hat{q}(s - i', a'', \vec{w})$

        $s_{i+1} \leftarrow s'_i$, $a_{i+1} \leftarrow a'_i$, $i \leftarrow i + 1$

        si $i == N$

            $\mathcal{L}(\vec{w}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - Q(s_i, a_i))^2$ // calcul de la loss

            $\vec{w} \leftarrow \vec{w} \alpha \nabla_{\vec{w}} \mathcal{L}(\vec{w})$ // mise à jour du réseau

            $i \leftarrow 0$

        choisir action $a'_i \in A$ pour s'_i avec ϵ-greedy et $\hat{q}(s'_i, \cdot, \vec{w})$

    jusqu'à ce que s soit terminal

Différences clés

- off-policy / on policy

⇨ avec SARSA, on ne peut utiliser que les données "courantes" pour faire la mise à jour
avec Q-learning, on fait comme si on mettait à jour la politique optimale

⇨ on peut utiliser tous les échantillons de la mémoire !

- Avec SARSA, les échantillons sont très corrélés, ce qui peut poser problème au réseau de neurones
- avec Deep Q-learning, on peut donc essayer de dé-corréler les échantillons...

Résultats de Convergence pour l'évaluation **d'une** 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

Experience replay et Deep Q-networks – Prédiction

DeepQN utilise experience replay

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

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

avec une descente stochastique de gradient

Permet d'apprendre à jouer à des jeux sous Atari