Notes de cours d'apprentissage par renforcement
En cours de rédaction

philippe.preux@univ-lille.fr
Université de Lille, CRIStAL, INRIA

Version du 10 mai 2021

Résumé

Ces quelques pages résument mon cours de master recherche en informatique sur le problème d'apprentissage par renforcement. La présentation est volontairement informelle ; néanmoins, elle se veut rigoureuse. Je veux que les notions introduites le soient en en comprenant leur sens, leurs limites, leurs possibilités et non pas comme de simples objets abstraits que l'on peut manipuler abstraitement. Je veux que ces notions soient ancréees autant que faire se peut dans le réel. Mon souhait est que ce texte soit utile à celui qui veut appliquer ces méthodes pour résoudre des problèmes concrets. Sans l'expliciter formellement, j'indique les limites théoriques connues pour que le lecteur soit conscient qu'il est en terrain théoriquement sûr ou s'il quitte cette sécurité. On sait que toutes les vraies applications vont au-delà de ce que dit, voire garantit, la théorie.

Ces notes sont rédigées avec pour cible l'étudiant ayant un niveau de début de master 2 en informatique. Les rappels de mathématiques nécessaires sont faits : on suppose que le lecteur a fait un peu de mathématiques après le bac. Le plus important pour lire la suite et d'être motivé, d'essayer de comprendre.

1 Introduction

Ce cours concerne le problème de prise de décisions séquentielle, c'est-à-dire des problèmes dans lesquels on doit prendre une succession de décisions, une décision prise à un moment pouvant avoir des répercussions immédiates et sur le long terme. Il existe différentes classes de tels problèmes. Nous nous concentrerons sur ceux qui sont généralement traités en apprentissage par renforcement, c'est-à-dire les problèmes de décision de Markov. Avec l'apprentissage supevisé et l'apprentisage non supervisé, l'apprentissage par renforcement constitue le troisième champ de l'apprentissage automatique. L'apprentissage supervisé concerne des problématiques de classement et de prédiction ; l'apprentissage non supervisé est très vaste et concerne l'identification de structures dans des données (l'exemple le plus connu est le regroupement de données qui se ressemblent, le *clustering* en anglais) ; l'apprentissage par renforcement consiste à « apprendre à faire » : apprendre à jouer aux échecs, apprendre à faire du vélo, apprendre à conduire une voiture, ... Naturellement, l'apprentissage par renforcement fait partie de l'intelligence artificielle.

L'apprentissage par renforcement a pour objet de permettre à un agent artificiel à apprendre à réaliser une tâche. Un agent est donc impliqué qui interagit avec son environnement. Cette interaction crée une boucle de rétro-action entre l'agent et son environnement qui rend ce type d'apprentissage très différent de l'apprentissage supervisé et de l'apprentissage non supervisé dans lesquels l'environnement ne réagit pas aux actions d'un agent. On sait qu'une boucle de rétro-action peut facilement être instable, ce qui rend la tâche d'apprentissage assez complexe : quand on apprend à faire du vélo, on nous explique un peu comment il faut faire, mais c'est en pratiquant, et souvent, en tombant (le système est instable), que l'on apprend à faire du vélo.

Ce cours est organisé comme suit. La section 2 introduit les problèmes de décision de Markov, l'outil et le concept de base sur lequel s'appuie ce cours. La résolution de ce problème quand il est complétement spécifié se nomme le problème de planification et il est grossièrement décrit dans la section 3. Il y a une littérature considérable sur ce problème. Lorsque le problème de décision de Markov n'est pas complétement spécifié, on aborde un problème d'apprentissage, en particulier d'apprentissage par renforcement : c'est le cœur de ce cours, introduit en section 4. Pour la mise en œuvre de ces notions on doit généralement représenter de manière approchée le problème et sa solution : c'est l'objet de la section suivante 5. Nous introduisons brièvement ensuite quelques algorithmes importants de l'apprentissage par renforcement tel qu'il se pratique actuellement en section 6. Enfin, nous balayons quelques sujets importants dans la 7 : nous touchons là les limites de ce que nous savons aujourd'hui faire en apprentissage par renforcement et les questions qui doivent être résolues si l'on veut aller plus loin dans sa mise en application. Je pense que pour tout sujet, son histoire est importante et conditionne l'essentiel de sa situation présente : je rappelle donc quelques étapes historiques du développement de l'apprentissage par renforcement en section 8, avant de donner des pointeurs vers des lectures indispensables pour le lecteur qui veut aller plus loin que ces simples notes.

2 Problèmes de décision de Markov

2.1 Un exemple introductif

Pour introduire le sujet, on va considérer un jeu très simple. Appelons-le le 21-avec-un-dé. Il ressemble au 21 que l'on joue avec un jeu de cartes.

On joue avec un dé normal à 6 faces, numérotées de 1 à 6. On le lance et il retombe ; on note le numéro sur la face supérieure ; on le relance et on ajoute le nouveau numéro de la face supérieure ; et ainsi de suite : il ne faut pas dépasser 21 ; avant chaque lancer, on décide de lancer à nouveau le dé ou de s'arrêter. Si on s'arrête, notre performance est le score final ; si celui-ci dépasse 21, le score final est de -1000. Le but est de faire un score final maximal.

On peut imaginer des tas de manières de jouer à ce jeu très simple. Par exemple :

— on lance le dé 5 fois.
— on lance le dé tant que le score ne dépasse pas 15 (car $21 - 6 = 15$)
— on lance le dé tant que le score ne dépasse pas 15 et ensuite on lance une pièce : si elle tombe sur pile, on relance ; si elle tombe sur face, on arrête

— on lance le dé tant que le score ne dépasse pas 18 car ensuite on a une chance sur 2 de dépasser 21.

On peut imaginer des tas de stratégies plus ou moins complexes. En existe-t-il une meilleure que les autres ? C'est tout l'objet de ce que je vais présenter dans la suite. Ce que je présente est très général : cela s'applique à ce jeu de dé, au jeu d'échecs, au contrôle des mouvements d'un robot aspirateur, à l'organisation d'une chaîne logistique, ...

2.2 Analyse

Analysons la situation rencontrée dans ce jeu de dé.

Pour jouer, on doit lancer un dé et le résultat de ce lancer nous est totalement hors de contrôle (à moins que le dé ne soit truqué). On doit lancer ce dé répétitivement ; à chaque lancer, notre score augmente ; avant de relancer le dé, on doit décider si on le lance ou si on s'arrête. On réalise donc une succession de décisions ; à chaque moment de décision, on doit décider si on lance le dé ou si on arrête. On peut donc définir un ensemble d'actions possibles à chaque instant de décision : $\mathcal{A} = \{\text{lancer, arrêter}\}$.

À chaque instant de décision t, la décision repose sur le score actuel. On imagine que tant qu'on est loin de 21, on va lancer le dé et que si on se rapproche de 21, il arrivera un moment où on arrêtera. On peut bien entendu imaginer des tas d'autres manières de prendre sa décision (l'heure du jour, la météo à Honolulu, les résultats du loto, la position de saturne dans le ciel, ...) mais il semble raisonnable que la décision se base exclusivement sur le score courant. Pour un esprit rationnel, c'est la seule information qui compte pour prendre sa décision. Cette information sur laquelle est basée la décision qui est prise se nomme l'*état* du système. Le système que l'on considère ici est dans un état appartenant à un certain ensemble de valeurs possibles : $\mathcal{S} = \{0, 1, 2, ..., 21, \text{plus que } 21\}$.

Le système étant dans un état, en fonction de l'action que l'on effectue, on peut déterminer dans quel état on va ensuite se trouver, de manière probabiliste. Si le score courant est 7, on a une chance sur 6 d'arriver ensuite dans l'un des états 8, 9, 10, 11, 12 ou 13. On peut donc définir une fonction $\mathcal{P}(s_{t+1}|s_t, a_t)$ qui donne la probabilité que le système soit dans l'état s_{t+1} (le score suivant l'action) étant donné l'état courant s_t (le score courant) et l'action réalisée $a_t \in \mathcal{A}$.

Quand on joue, à l'issue de la dernière action que nous réalisons, on détermine la performance finale : soit on a atteint un score ≤ 21, soit on a dépassé 21 et on a perdu. Quand on perd, on peut dire que l'on a atteint un score arbitraire, négatif. On peut définir l'objectif du jeu comme maximiser ce score final.

À ce stade, on a réduit le jeu de 21-avec-un-dé à un ensemble d'éléments :

— un ensemble d'instants de décision, $t \in \mathcal{T}$
— un ensemble d'états du jeu, $s \in \mathcal{S}$
— un ensemble d'actions possibles, $a \in \mathcal{A}$
— une fonction définissant comment l'état évolue à la suite de chaque action, $\mathcal{P}$
— une fonction score
— un objectif que l'on veut maximiser

$(\mathcal{S}, \mathcal{P})$ définissent un système qui évolue au fil du temps, ce qui s'appelle un *système dynamique*. $\mathcal{A}$ indique comment ce système évolue : il n'évolue pas de lui-même comme une pomme qui tombe mais sous l'effet d'actions.

L'état « plus que 21 » est un état particulier : quand le système est dans cet état, la tâche est terminée (on a perdu) : c'est un état *terminal*.

De même, on part toujours du même état, l'état 0 : c'est un état *initial*.

$\mathcal{P}$ spécifie la dynamique du système, c'est-à-dire sa loi d'évolution.

Concernant la fonction score, d'une manière générale, elle indique les conséquences de l'exécution de chaque action dans chaque état. Ici, on a indiqué ces conséquences à la fin de la manche : on perd ou on gagne un certain nombre de points qui est le score final s'il est inférieur à 21.

On notera que l'état du système à un instant donné ne dépend que de son état à l'instant précédent et de l'action qui a été réalisée. Un tel système est dit *markovien*, ou *a-historique* : en ce qui concerne l'évolution future du système, on n'a pas à connaître son évolution passée : seul compte son état courant.

On notera ausssi que l'état du système à un moment résulte d'une action et, dans le cas où l'action consiste à lancer le dé, du hasard. Lorsqu'il en est ainsi, on dit que la dynamique du système est *non déterministe* ou *stochastique*.

On notera encore que l'état contient toute l'information nécessaire à la prise de la décision : il n'y a pas d'*information cachée*. Un exemple où la prise de décision optimale dépend d'information cachée est donné par les jeux de cartes où chaque joueur cache son jeu : à un moment, un joueur doit jouer une carte (décider quelle carte jouer) sans connaître le jeu de ses adversaires : sans cette information, il ne peut pas savoir s'il prend une bonne décision, encore moins, s'il prend la meilleure décision possible.

Dans la section suivante, nous allons reprendre toutes ces notions de manière abstraite pour définir la notion fondamentale dans ce cours de *problème de décision de Markov*.

2.3 Définition d'un problème de décision de Markov

Dans cette section, on définit tout d'abord un *processus* de décision de Markov, puis la notion de *problème* de décision de Markov qui s'appuie sur celle de processus de Markov.

2.3.1 Processus de décision de Markov

Définition 1 *Un processus de décision de Markov décrit l'évolution d'un système dynamique contrôlé. On le définit pas un quadruplet $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ où :*

- *$\mathcal{S}$ est l'ensemble des états du processus. $\mathcal{S}$ est fini. On notera N son cardinal,* i.e. *le nombre d'états.*
- *$\mathcal{A}$ est l'ensemble des actions possibles sur ce processus. $\mathcal{A}$ est fini. On notera P le nombre d'actions.*

  Remarque : de manière générale, l'ensemble d'actions peut dépendre de l'état, donc devoir écrire $\mathcal{A}(s)$. Afin de rester simple, on supposera que l'ensemble des actions est le même

dans tous les états, sauf dans quelques cas particuliers où cela sera précisé.

— $\mathcal{P}$ *est la* fonction de transition *: pour chaque couple (état, action), cette fonction indique la probabilité que le système soit ensuite dans chaque état :*

$$\begin{aligned}\mathcal{P} \quad : \quad & \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0,1] \\ & (s,a,s') \mapsto \mathcal{P}(s,a,s') = Pr[s_{t+1} = s' | s_t = s, a_t = a]\end{aligned}$$

— $\mathcal{R}$ *est la* fonction de retour. *À valeur réelle, cette fonction formalise les conséquences d'une action émise dans un état. Cette notion porte le nom de* retour. *Dans un système stochastique, ce retour varie au fil des expériences ; $\mathcal{R}(s,a)$ est le retour moyen (espéré) quand on exécute l'action a dans l'état s. On suppose que ce retour est fini.*

$$\begin{aligned}\mathcal{R} \quad : \quad & \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R} \\ & (s,a,s') \mapsto \mathcal{R}(s,a,s') = \mathbb{E}[r_t | s_{t+1} = s', s_t = s, a_t = a]\end{aligned}$$

La notion de retour est assez compliquée à présenter, mais on la comprend bien dans la pratique. Un retour peut être positif ou négatif : positif si les conséquences de l'action sont bonnes pour atteindre l'objectif, négatif sinon. Positif, cela peut être un gain d'argent, de points, ... Négatif, cela peut être un coût énergétique, la perte d'une pièce à un jeu, ... Face à un problème, la définition du retour peut être contrainte par le problème lui-même, ou pas. Dans le jeu de 21-avec-un-dé, il est possible de définir ce retour de différentes manières, bien que l'objectif du jeu demeure le même.

On pourrait ajouter à cette définition d'un processus de décision de Markov l'ensemble des instants de décision. Nous ne le faisons pas pour simplifier la présentation et les notations. Cet ensemble des instants de décision est un sous-ensemble de $\mathbb{N}$, typiquement les naturels qui se suivent à partir de 0.

Résumons la situation : un agent (humain, virtuel) doit contrôler un système. Celui-ci est dans un certain état au début de cette interaction, à l'instant initial que l'on note t_0 ; à tout instant t, le système est dans l'état s_t ; l'agent choisit une action à effectuer a_t (en fonction de s_t) ; suite à cette action, le système fournit un retour r_t à l'agent et passe dans l'état s_{t+1}. La succession d'états $(s_0, ..., s_t, ...)$ visités par le système se nomme une *trajectoire*. La trajectoire peut aussi se définir comme la séquence des états et des actions $(s_0, a_0, s_1, s_1, ...)$ ou encore la séquence (état, action, retour) : $(s_0, a_0, r_0, ...)$. La définition utilisée dépend de l'usage que l'on en fait.

2.3.2 Problème de décision de Markov

Définition 2 *Informellement, un problème de décision de Markov (abrégé en PDM) s'appuie sur un processus de décision de Markov et le but est de trouver les actions à réaliser afin d'optimiser une certaine fonction objectif qui s'exprime généralement en fonction des retours.*

Cette fonction objectif peut prendre beaucoup de formes. On en donne quelques-unes ci-dessous :

— la somme des retours collectés au long d'une trajectoire : $\sum_{t\geq 0} r_t$: il faut que la trajectoire soit de longueur finie.
— la moyenne des retours collectés au long d'une trajectoire : $\frac{1}{T}\sum_{t=0}^{t=T-1} r_t$: il faut que la trajectoire soit de longueur finie.
— la somme pondérée par un facteur déprécié : $\sum_t \gamma^t r_t$ où $\gamma \in [0,1]$; la trajectoire peut être de longueur infinie.

On peut adapter ces définitions pour considérer ces sommes non pas à partir de $t = 0$ mais à partir d'un t quelconque : $R_t = \sum_{k\geq 0} r_{t+k}$, $R_t = \frac{1}{K}\sum_{k=0}^{k=K-1} r_{t+k}$, $R_t = \sum_{k\geq 0} \gamma^k r_{t+k}$ respectivement.

La dernière formulation peut sembler bien compliquée. Elle est en fait très générale, très utile et très utilisée en pratique. Notons que si $\gamma = 1$, on retrouve la première définition (somme des retours collectés au long d'une trajectoire). Dans la suite, on utilisera cette définition.

Quand $\gamma = 1$, $R(t)$ peut tendre vers l'infini si un état terminal n'est pas atteint au bout d'un nombre fini de transitions. Donc, on ne considère $\gamma = 1$ que quand c'est le cas, sinon on prend $\gamma \in [0, 1[$. Il y a de nombreux cas pratiques où on sait que l'on va prendre un nombre fini, fixé et connu de décisions successives (par exemple, un jeu dans lequel on joue un nombre fixé et connu à l'avance de coups, comme le 421) ; le nombre peut être fini sans être connu, comme dans de nombreux jeux de plateau (même si certains peuvent en principe se poursuivre à l'infini) ou le 21-avec-un-dé ou l'exemple introduit à la section 2.4. On parle d'*horizon* fini ou infini selon que l'on sait que la tâche s'arrêtera au bout d'un nombre fini ou potentiellement infini de décisions.

Il faut distinguer le retour $\mathcal{R}$ qui définit les conséquences immédiates d'une action avec ce R que l'agent cherche à optimiser pour accomplir sa tâche. Pour optimiser R, il est parfois, souvent, nécessaire de subir de temps à autre de mauvaises conséquences immédiates ; par exemple, pour gagner aux échecs, on doit parfois sacrifier une pièce. $\mathcal{R}$ se nomme aussi le *retour immédiat*.

L'objectif de l'agent est donc de trouver quelles sont les actions à exécuter pour optimiser R, ce que l'on appelle sa *politique*.

Il est vraiment très important que dès maintenant, le lecteur sache que face à un problème qu'il veut résoudre, sa modélisation sous la forme d'un PDM est une tâche qui peut être difficile. Il est courant que la fonction $\mathcal{R}$ ne soit pas clairement spécifiée dans la tâche et c'est souvent un élément important de la modélisation ; il peut être utile de tester plusieurs idées. De même, l'espace d'états est rarement facile à définir. Également, même si nous ne traitons que de la fonction objectif « somme pondérée par un facteur déprécié », la fonction à optimiser peut souvent être débattue. Il faut être néanmoins prudent quant à la fonction choisie car il faut qu'elle puisse être optimisée, si possible efficacement, ce qui n'est pas le cas de toutes les fonctions que l'on pourrait imaginer.

Notons enfin que nous supposons que l'on dispose d'un simulateur du PDM, c'est-à-dire d'une fonction qui prend en entrée un état et une action et retourne l'état suivant et le retour (en fonction de $\mathcal{P}$ et $\mathcal{R}$). Cette fonction est plus ou moins facile à écrire pour les jeux ; elle peut être beaucoup plus difficile à écrire, voire impossible à écrire, pour de très nombreuses autres tâches (en robotique, pilotage de véhicule autonome, pilotage d'une chaîne de production, ...). Ces problèmes seront brièvement discutés à la fin du cours.

TABLE 1 – Exemple du chauffeur de taxi : probabilité de transition pour les différents états et différentes actions.

$\mathcal{P}$	état courant s_t (ville courante)	action a_t	état suivant s_{t+1} (ville suivante)		
			A	B	C
	A	a_1	1/2	1/4	1/4
		a_2	1/16	3/4	3/16
		a_3	1/4	1/8	5/8
	B	a_1	1/2	0	1/2
		a_3	1/16	7/8	1/16
	C	a_1	1/4	1/4	1/2
		a_2	1/8	3/4	1/8
		a_3	3/4	1/16	3/16

L'objectif de ce cours est donc de résoudre un problème de décision de Markov. On va distinguer deux cas : le cas où ce problème est complétement spécifié et le cas où $\mathcal{P}$ et $\mathcal{R}$ sont inconnues. Dans le premier cas, il s'agit d'un problème d'otpimisation portant le nom de *planification* qui sera briévement étudié à la section 3. Dans le second, il s'agît du problème d'apprentissage par renforcement proprement dit.

Avant d'aller plus loin, nous présentons un autre exemple de problème qui va nous permettre d'illustrer toutes les notions de manière simple : avec ces 23 états, le 21-avec-un-dé ne se prête pas facilement à une illustration très simple.

2.4 Un second exemple illustratif : le problème du chauffeur de taxi

Un chauffeur de taxi souhaite maximiser ses revenus. Il travaille dans 3 villes A, B et C. Quand il ne transporte pas déjà un client, il a le choix entre :

— rouler en espérant être hélé par un client,

— s'arrêter à une station de taxis dans la ville où il se trouve et attendre un client,

— stationner là où il se trouve et attendre un appel radio pour aller chercher un client.

Dans la ville B, il n'y a pas de station de taxis, donc la deuxième action est impossible.

Cette situation peut se modéliser par un PDM avec un ensemble de 3 états, l'état indiquant la ville dans laquelle il se trouve présentement. Quand il charge un client, en fonction de la destination demandée par le client, il reste dans la même ville ou va vers une autre ville. On suppose que la probabilité de chaque transition est donnée dans la table 1.

On suppose que les gains pour chaque course sont en moyenne ceux indiqués dans la table 2.

On considère que le chauffeur de taxi entend maximiser la somme dépréciée de ses gains, soit $R_t = \sum_{k\geq 0} \gamma^k r_{t+k}$.

Une manière d'illustrer un tel PDM par un graphique consiste à représenter le problème par un graphe dans lequel chaque état est un nœud et les actions étiquettent les arcs entre les nœuds.

TABLE 2 – Exemple du chauffeur de taxi : gain espéré pour les différents états et différentes actions.

$\mathcal{R}$	état courant s_t (ville courante)	action a_t	état suivant s_{t+1} (ville suivante)		
			A	B	C
	A	a_1	10	4	8
		a_2	8	2	4
		a_3	4	6	4
	B	a_1	14	0	18
		a_3	8	16	8
	C	a_1	10	2	8
		a_2	6	4	2
		a_3	4	0	8

Ces étiquettes peuvent être complétées par la probabilité de transition et le retour espéré sur la transition. La figure 1 illustre ainsi le problème du chauffeur de taxi qui, bien que très simple, engendre déjà un graphique assez chargé.

2.5 Politique

Définition 3 *Une politique, notée π, spécifie la manière dont l'agent choisit son action.*

Une politique peut être formulée de différente manière. Les plus courantes sont de l'une de ces 3 formes :

— politique détermministe : à un état, la politique associe une action :

$$\begin{aligned} \pi \quad &: \quad \mathcal{S} \to \mathcal{A} \\ &\quad\; s \mapsto a \end{aligned}$$

— politique stochastique : à un état, la politique associe une distribution de probabilités sur les actions :

$$\begin{aligned} \pi \quad &: \quad \mathcal{S} \times \mathcal{A} \to [0,1] \\ &\quad\; (s,a) \mapsto \pi(s,a) = Pr[a_t = a | s_t = s] \end{aligned}$$

— politique non stationnaire : la politique dépend de l'instant de décision :

$$\begin{aligned} \pi \quad &: \quad \mathcal{S} \times \mathcal{A} \times \mathcal{T} \to [0,1] \\ &\quad\; (s,a,t) \mapsto \pi(s,a,t) = Pr[a_t = a | s_t = s] \end{aligned}$$

Ici on donne la forme d'une politique stochastique. Elle poourrait être déterministe ($\pi(s,t)$).

La solution d'un PDM est une politique. On peut voir la suite du cours comme la réponse à la question : comment trouve-t-on la politique solution d'un PDM ?

On va voir qu'il existe des politiques plus ou moins bonnes pour un PDM donné. Si la fonction objectif est $R(t) = \sum_{k \geq 0} \gamma^k r_{t+k}$, il en existe une qui est meilleure, ou plusieurs qui

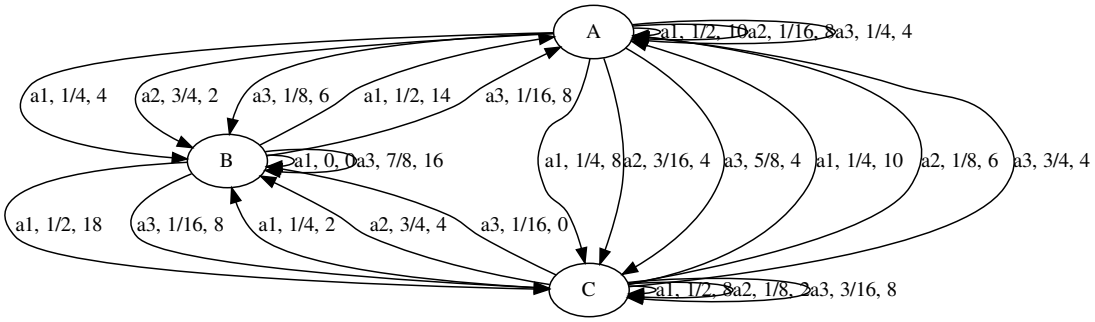


FIGURE 1 – Exemple du chauffeur de taxi : représentation graphique du PDM. Chaque arc représente une transition possible et est étiquété par l'action qui la provoque, avec la probabilité que cette action dans cet état provoque cette transtion, et le retour espéré sur cette transition.

sont équivalentes. Pour cette fonction objectif, le théorème de Blackwell démontre qu'une/la politique optimale est déterministe et stationnaire, donc de la forme $\pi(s)$. On a besoin d'un critère de jugement des politiques pour pouvoir les comparer et déterminer la « meilleure ». La meilleure est celle qui satisfait au mieux la fonction objectif ; si cette notion est claire dans un environnement déterministe, elle l'est moins dans un environnement non déterministe. Ce critère se nomme la *valeur* d'une politique ; on définit ce critère dans la section suivante.

2.6 Valeur d'un état

Informellement, la valeur d'un état indique s'il est bénéfique d'être dans cet état si on veut optimiser la fonction objectif. Si c'est le cas, une bonne politique va essayer d'atteindre des états dont la valeur est élevée si on maximise la fonction objectif, faible si on la minimise.

On suppose que l'on s'intéresse à un PDM dont la fonction objectif est $R(t) = \sum_{k\geq 0} \gamma^k r_{t+k}$. Supposons que l'on dispose d'une politique π.

Une propriété remarquable de la valeur de l'état que nous allons définir pour cette fonction objectif est que situé dans un état, on peut trouver un état dont la valeur est meilleure dans son voisinage immédiat, à moins que l'on n'ait déjà atteint l'optimum. On peut ainsi atteindre un des états les meilleurs facilement, au moins en principe. Il n'existe pas une telle notion pour la plupart des problèmes d'optimisation dont la résolution est complexe à cause de la présence d'optima locaux. Pour le problème qui nous intéresse ici, la fonction valeur possède un seul optimum global (ou plusieurs équivalents) et n'a pas d'optimum local.

Supposons que l'on dispose d'une politique π et d'un PDM. À partir d'un état initial s_0, on applique la politique π : le système va passer successivement par différents états, générer une séquence de retours au long d'une trajectoire. On peut calculer R à partir de s_0. Si l'on recommence la même procédure toujours à partir de s_0, l'environnement étant non déterministe, on va obtenir une autre trajectoire, donc une autre valeur de R. Et ainsi de suite : recommencons cette procédure n fois : on obtient n valeurs de R. Si n est assez grand, ces n valeurs de R

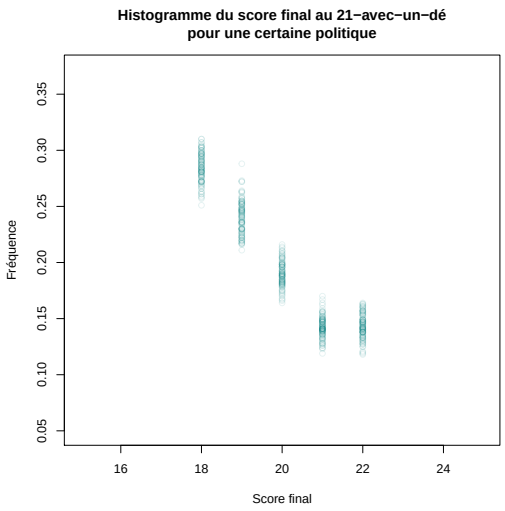


FIGURE 2 – Pour le jeu de 21-avec-un-dé, on a réalisé 100 fois $n = 1000$ parties à partir de l'état 0, en suivant une politique π qui consiste à lancer le dé jusqu'à atteindre le score de 18 et s'arrêter. On représente ici la distribution du score final pour chacune de ces parties en indiquant la variabilité observée sur ces 100 n parties. La valeur 22 en abscisses doit s'interpréter comme « plus que 21 » : cela signifie que l'on a dépassé 21, donc que l'on a perdu.

ont tendance à se rassembler (se concentrer) autour d'une valeur moyenne. Ainsi, la figure 2 représente ce R pour le jeu de 21-avec-un-dé pour $n = 1000$.

Cette valeur moyenne tend vers une valeur qui se nomme son *espérance*. D'une manière générale en probabilités, l'espérance d'une variable aléatoire est sa valeur moyenne quand on effectue une infinité de réalisations.

Définition 4 *La valeur de l'état s pour la politique π est l'espérance de R quand l'environnement est initialement dans l'état s. On la note $V^\pi(s)$.*

$V^\pi(s) \equiv \mathbb{E}[R(s_0)|s_0 = s]$. $\mathbb{E}[v]$ dénote l'espérance de la variable aléatoire v et la notation $R(s_0)|s_0 = s$ signifie que l'on s'intéresse à la variable aléatoire $R(s)$ quand l'état initial est s.

Définition 5 *La fonction valeur, ou valeur, est une application qui associe à tout état sa valeur (pour une certaine politique). On la note V^π.*

2.7 Valeur d'une paire état-action

Une autre notion de valeur concerne les paires état-action : la valeur (ou l'appelle aussi la *qualité*) d'une paire état action (s, a) pour une politique π que nous noterons $Q^\pi(s, a)$.

Définition 6 *$Q^\pi(s, a)$ est la valeur de l'état s quand on y effectue l'action a puis que l'on applique la politique π. On peut écrire cela comme suit :*

$$Q^\pi(s, a) \equiv \mathbb{E}[V^\pi(s_0)|s_0 = s, a_0 = a, a_{t>0} \sim \pi]$$

La notation $a_{t>0} \sim \pi$ signifie que l'on choisit l'action a_t en fonction de π.

Alors que la valeur d'un état indique s'il est bon de passer par cet état pour optimiser la fonction objectif, la qualité indique s'il est bon d'exécuter une certaine action dans un certain état.

2.8 Les équations de Bellman pour V^π et Q^π

V^π et Q^π vérifient chacune une équation qui permet de calculer exactement leur valeur.

On montre que :

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(s,a) \sum_{s' \in \mathcal{S}} \mathcal{P}(s,a,s')[\mathcal{R}(s,a,s') + \gamma V^\pi(s')] \quad (1)$$

et

$$Q^\pi(s,a) = \sum_{s'' \in \mathcal{X}} \mathcal{P}(s,a,s')[\mathcal{R}(s,a,s') + \gamma \sum_{a' \in \mathcal{A}} \pi(s',a') Q^\pi(s')] \quad (2)$$

La démonstration de ces deux équations n'est pas très difficile. ; c'est un excellent exercice que d'essayer de la faire. L'idée est de partir de la définition de V^π et d'exprimer l'espérance en fonction de $\mathcal{P}$, $\mathcal{R}$ et π.

Derrière leur aspect qui peut paraître complexe se cache une réalité très simple. En effet, π, $\mathcal{S}$, $\mathcal{R}$, γ sont connus. Pour la première équation de Bellman (en V^π), seul $V^\pi(s)$ est inconnu pour chaque état s. Aussi, en développant la double somme, on va tout simplement obtenir une relation linéaire entre la valeur d'un état s et la valeur de tous les états de $\mathcal{S}$. Aussi, derrière cette relation récurrente, se cache un simple système d'équations linéaires avec N équations et N inconnues, où N est le nombre d'états.

Pour Q^π, c'est exactement la même chose : on a $N \times P$ inconnues et autant d'équations linéaires, où N est le nombre d'états et P le nombre d'actions.

2.9 Valeur d'une politique

Étant donnée un PDM et une politique π, on veut calculer la valeur de cette politique, c'est-à-dire la valeur de chaque état du PDM pour cette politique. Les équations de Bellman permettent facilement de faire ce calcul, au moins en principe. Comme on l'a vu dans la section précédente, les équations de Bellman expriment un systèmes d'équations linéaires ayant autant d'inconnues que d'équations. C'est donc, en principe, l'un des problèmes les plus simples que l'on puisse imaginer.

Illustrons cela sur le problème du chauffeur de taxi. On considère une politique stochastique uniforme, *i.e.* $\pi(A,a) = \pi(C,a) = 1/3$ pour chacune des trois actions dans les villes A et C, et $\pi(B,a) = 1/2$ pour les deux actions possibles dans la ville B. On utilise les données du problème décrit à la section 2.4 et on prend $\gamma = 0{,}9$.

Pour la ville A, on obtient l'équation :

$$
\begin{aligned}
V^\pi(A) \quad &= \pi(A,a_1)\{ \quad P(A,a_1,A)[R(A,a_1,A)+\gamma V^\pi(A)] + \\
&\qquad\qquad\quad P(A,a_1,B)[R(A,a_1,B)+\gamma V^\pi(B)] + \\
&\qquad\qquad\quad P(A,a_1,C)[R(A,a_1,C)+\gamma V^\pi(C)]\} \\
&+\pi(A,a_2)\{ \quad P(A,a_2,A)[R(A,a_2,A)+\gamma V^\pi(A)] + \\
&\qquad\qquad\quad P(A,a_2,B)[R(A,a_2,B)+\gamma V^\pi(B)] + \\
&\qquad\qquad\quad P(A,a_2,C)[R(A,a_2,C)+\gamma V^\pi(C)]\} \\
&+\pi(A,a_3)\{ \quad P(A,a_3,A)[R(A,a_3,A)+\gamma V^\pi(A)] + \\
&\qquad\qquad\quad P(A,a_3,B)[R(A,a_3,B)+\gamma V^\pi(B)] + \\
&\qquad\qquad\quad P(A,a_3,C)[R(A,a_3,C)+\gamma V^\pi(C)]\}
\end{aligned}
$$

ce qui se ré-écrit :

$$V^\pi(A) = 5 + 0{,}9 \times 13/48 V^\pi(A) + 0{,}9 \times 3/8 V^\pi(B) + 0{,}9 \times 17/48 V^\pi(C)$$

On procède de même pour les villes B et C et on obtient le système d'équations linéaires :

$$
\begin{cases}
V^\pi(A) = 5 + 0{,}9 \times 13/48 V^\pi(A) + 0{,}9 \times 3/8 V^\pi(B) + 0{,}9 \times 17/48 V^\pi(C) \\
V^\pi(B) = 31/2 + 0{,}9 \times 9/32 V^\pi(A) + 0{,}9 \times 7/16 V^\pi(B) + 0{,}9 \times 9/32 V^\pi(C) \\
V^\pi(C) = 31/6 + 0{,}9 \times 3/8 V^\pi(A) + 0{,}9 \times 17/48 V^\pi(B) + 0{,}9 \times 13/48 V^\pi(C)
\end{cases}
$$

On trouve :

$$
\begin{cases}
V^\pi(A) = \frac{156420}{1789} \approx 87{,}43 \\
V^\pi(B) = \frac{5113540}{51881} \approx 98{,}6 \\
V^\pi(C) = \frac{13602460}{155643} \approx 87{,}40
\end{cases}
$$

Cela signifie que si le chauffeur de taxi suit la politique stochastique uniforme décrite plus haut, son revenu à long terme sera en moyenne de 87,43 s'il démarre dans l'état A, 98,6 s'il démarre dans l'état B et 87,40 s'il démarre dans l'état C.

On va voir plus loin d'autres méthodes pour calculer la valeur d'une politique, méthodes qui sont plus efficaces que la résolution d'un système d'équations linéaires. Il est plaisant de calculer la valeur d'une politique de cette manière, et intellectuellement parlant, c'est important de savoir que l'on peut résoudre ce problème de cette manière. Néanmoins, quand le nombre d'états est grand, la résolution de ce système d'équations linéaires devient problématique. Toute une communauté scientifique de mathématiciens étudie d'arrache-pied ce problème depuis des générations, des dizaines de milliers de publications ont été écrites sur le sujet, mais le problème demeure difficile si le nombre d'équations est grand. Nous voulons pouvoir traiter des problèmes pour lesquels N est des ordres de grandeur plus grand que le milliard et on est loin de savoir résoudre des systèmes d'équations linéaires de cette taille.

2.10 Les équations d'optimalité de Bellman

Pour un PDM dont la fonction objectif est la maximisation de la somme pondérée par un facteur déprécié, en supposant que l'ensemble des états et l'ensemble des actions sont finis et les retours sont finis et que $\gamma \in [0, 1[$, le théorème de Blackwell nous dit qu'il existe une politique optimale et que cette politique optimale est déterministe et stationnaire. Ce théorème est important puisqu'il nous affirme qu'il existe une solution au problème que nous voulons résoudre.

On peut écrire une équation de Bellman vérifiée par la politique optimale. Comme les équations de Bellman vues plus haut, on peut l'écrire pour V et pour Q. Ces équations donnent l'espoir qu'en les résolvant on obtiendra directement la politique optimale pour contrôler un PDM : c'est vrai, mais leur résolution n'est pas facile. Néanmoins, ces équations sont fondamentales pour résoudre des PDM et seront mises en œuvre à de nombreuses reprises dans la suite.

Avant tout, pour ce type de PDM, on peut définir un ordre partiel sur les politiques. Informellement, une politique π est meilleure qu'une politique π' si on a $V^{\pi}(s) \geq V^{\pi'}(s), \forall s \in \mathcal{S}$: la fonction valeur de π est toujours « au dessus » de celle de π'.

Aussi, la politique optimale est elle la meilleure possible en tout état. En notant V^* la fonction valeur d'une politique optimale, on peut donc écrire :

$$V^*(s) \equiv \max_{\pi} V^{\pi}(s) = \max_{a \in \mathcal{A}(s)} Q^{\pi}(s, a), \forall s \in \mathcal{S}$$

On en déduit l'équation d'optimalité de Bellman pour V :

$$V^*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s, a, s')[\mathcal{R}(s, a, s') + \gamma V^*(s')] \tag{3}$$

Si on minimise la fonction objectif, on remplace le max par un min.
Entre V^* et Q^*, on a la propriété suivante :

$$V^*(s) = \max_{a \in \mathcal{A}} Q^*(s, a) \tag{4}$$

Et l'équation d'optimalité de Bellman pour Q^* :

$$Q^*(s, a) = \sum_{s' \in \mathcal{S}} \mathcal{P}(s, a, s')[\mathcal{R}(s, a, s') + \gamma \max_{a' \in \mathcal{A}} Q^*(s', a')] \tag{5}$$

Muni de cet outillage théorique, nous allons maintenant étudier des algorithmes calculant cette politique optimale pour un PDM quelconque. Dans la section 3, nous étudions le cas où le PDM est complétement spécifié. Dans la section 4, nous étudions le cas où $\mathcal{P}$ et $\mathcal{R}$ sont inconnues, ce qui constitue le problème d'apprentissage par renforcement.

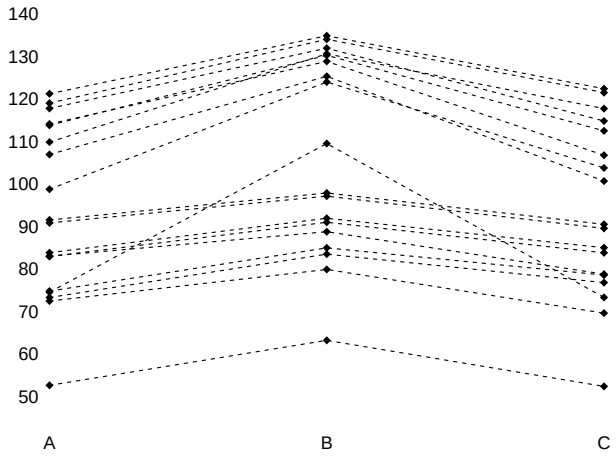


FIGURE 3 – On a représenté la fonction valeur de toutes les politiques stationnaires et déterministes possibles du problème du chauffeur de taxi, pour $\gamma = 0{,}9$. Les valeurs des trois états correspondant à une même politique sont reliées par des traits en pointillées. On voit que l'une de ces fonctions est toujours « au-dessus » des autres : c'est la valeur de la politique optimale pour ce PDM. On voit aussi qu'une autre est toujours en-dessous : c'est la pire des politiques, néanmoins optimale si on minimise l'objectif. On peut remarquer qu'entre ces deux extrêmes, certaines fonctions valeurs se croisent : l'ordre est partiel.

3 Résolution du problème de planification

Dans cette section, on suppose que l'on dispose d'un PDM complétement spécifié. On étudie deux questions :

— calculer la valeur de la politique optimale

— calculer la politique optimale

Pour cela, on présente des algorithmes de programmation dynamique qui calculent directement la fonction valeur à partir du PDM. Pour son intérêt théorique au moins, on présente aussi une approche à base de programme linéaire.

3.1 Valeur d'une politique

On a vu plus haut comment calculer V^π en résolvant un système d'équations linéaires. On voit ici une autre approche basée sur l'équation de Bellman. On peut donc voir cette méthode comme une autre manière de résoudre un système d'équations linéaires.

Comme on l'a vu plus haut, l'équation de Bellman indique que la valeur d'une politique vérifie : $V^\pi(s) = \sum_{a\in\mathcal{A}} \pi(s,a) \sum_{s'\in\mathcal{S}} \mathcal{P}(s,a,s')[\mathcal{R}(s,a,s') + \gamma V^\pi(s')]$

Pour des raisons que nous n'expliquerons pas ici qui sont liées aux propriétés du problème que nous etudions, un algorithme très simple pour calculer V^π est le suivant :

— initialiser $V(s) \leftarrow 0, \forall s \in \mathcal{S}, k \leftarrow 0$

— itérer : $V_{k+1}(s) \leftarrow \sum_{a\in\mathcal{A}} \pi(s,a) \sum_{s'\in\mathcal{S}} \mathcal{P}(s,a,s')[\mathcal{R}(s,a,s') + \gamma V_k(s')]$; $k \leftarrow k+1$

— jusqu'à ce que l'écart entre V_k et V_{k-1} soit faible.

Ces itérés convergent vers V^π asymptotiquement. On mesure l'« écart entre deux itérations » par la $||V_{k+1} - V_k||_\infty$ où V_k est la valeur de V à la k^{e} itération et la notation $||.||_\infty$ veut simplement dire $\max_{s \in \mathcal{S}} |V_{k+1}(s) - V_k(s)|$.

En guise d'exercice, le lecteur est invité à écrire un programme qui effectue ce calcul, à le tester sur le problème du chauffeur de taxi (avec $\gamma = 0{,}9$), et évaluer la politique stochastique uniforme pour laquelle on a calculé la fonction valeur à la section 2.9.

3.2 Amélioration d'une politique

Ayant calculé la valeur d'une politique π, on peut l'utiliser pour trouver une politique meilleure que π.

Propriété 1 *Soit la politique* $\pi'(s) \equiv arg\ max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s, a, s')[\mathcal{R}(s, a, s') + \gamma V^\pi(s')]$

$\pi' \geq \pi$ au sens de l'ordre partiel défini plus haut sur les politiques (section 2.10) et si $\pi' = \pi$, alors c'est une politique optimale.

La politique qui choisit l'action qui maximise $Q(s, a)$ en tout s se nomme la *politique gloutonne* (*greedy* en anglais).

Notons que la nouvelle politique est déterministe.

En guise d'exercice, appliquer cet algorithme pour améliorer la politique stochastique uniforme pour le problème du chauffeur de taxi (avec $\gamma = 0{,}9$).

3.3 Politique optimale d'un PDM

Donc, nous avons vu comment, disposant d'une politique, on peut estimer sa valeur et nous venons de voir comment, disposant d'une fonction valeur, on peut produire une politique meilleure que celle associée à cette fonction valeur, à moins que l'on ne puisse tout simplement pas l'améliorer car étant déjà optimale. Cela nous fournit l'idée d'un algorithme pour produire une politique optimale pour un PDM :

1. démarrer avec une politique quelconque
2. calculer sa valeur
3. construire une politique meilleure que la précédente
4. retourner à l'étape 2 tant que l'on arrive à produire une politique strictement meilleure.

Formalisé, cela nous donne l'algorithme 1 d'itération sur les politiques (IP). La seule très légère difficulté est que pour améliorer la politique, nous travaillons avec des politiques déterministes alors que dans l'évaluation de la politique, nous travaillons avec des politiques non déterministes. Pour mettre cela en cohérence, nous adaptons l'algorithme d'évaluation de la politique à une politique déterministe.

Propriété 2 *Le nombre de politiques étant fini et l'algorithme d'itération sur les politiques améliorant la politique courante à chaque itération sauf s'il a trouvé la politique optimale, l'algorithme d'itération sur les politiques converge vers une politique ϵ-optimale au bout d'un nombre fini d'itérations.*

Cette propriété indique la convergence vers une politique ϵ-optimale. Cela signifie que la valeur de la politique calculée par cet algorithme et la valeur de la politique de la politique optimale est plus petite que ϵ. Pour être précis : $||V^{\pi} - V^{*}||_{\infty} \leq \epsilon$. Cette non exactitude provient du fait que l'on ne calcule pas exactement la valeur de la politique courante, mais on l'approxime. À la ligne 11 de l'algorithme 1, on itère jusqu'à ce que l'écart entre deux itérations successives soient plus petit qu'une quantité qui dépend de ϵ ; on ne peut pas calculer exactement la valeur de la politique courante : la convergence est asymptotique. Aussi, comme on calcule ensuite la politique gloutonne d'une fonction valeur approchée, il peut arriver que la meilleure action dans un état donné de cette approximation ne soit pas la meilleure action pour la valeur exacte. On ne peut jamais garantir que l'on a calculé la politique optimale ; on peut seulement garantir que l'on a calculé une politique dont la valeur ne s'écarte pas trop de la valeur optimale. Pour que cet écart soit inférieur à ϵ, il faut itérer la boucle répéter jusqu'à ce que l'écart soit celui indiqué à la ligne 11.

Algorithm 1 L'algorithme d'itération sur les politiques.

Require: un PDM : $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$
Require: un seuil de précision ϵ
1: initialiser π_0 (aléatoirement ou autrement)
2: $k \leftarrow 0$
3: **repeat**
4: initialiser V_0^{π} (aléatoirement ou autrement)
5: $i \leftarrow 0$
6: **repeat**
7: **for** tout état $s \in \mathcal{S}$ **do**
8: $V_{i+1}^{\pi_k}(s) \leftarrow \sum_{s' \in \mathcal{S}} \mathcal{P}(s, \pi_k(s), s')[\mathcal{R}(s, \pi(s), s') + \gamma V_i^{\pi_k}(s')]$
9: **end for**
10: $i \leftarrow i + 1$
11: **until** $||V_i^{\pi_k} - V_{i-1}^{\pi_k}||_{\infty} \leq \epsilon \frac{(1-\gamma)}{2\gamma}$
12: **for** tout état $s \in \mathcal{S}$ **do**
13: $\pi_{k+1}(s) \leftarrow \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s, a, s')[\mathcal{R}(s, a, s') + \gamma V^{\pi_k}(s')]$
14: **end for**
15: $k \leftarrow k + 1$
16: **until** $\pi_k = \pi_{k-1}$

3.4 Valeur optimale d'un PDM

Plutôt que de s'appuyer sur l'équation de Bellman pour évaluer une politique, on peut utiliser l'équation d'optimalité de Bellman pour obtenir directement la politique optimale. Rappelons-nous cette équation d'optimalité de Bellman :

$$V^*(s) = \max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \mathcal{P}(s, a, s')[\mathcal{R}(s, a, s') + \gamma V^*(s')] \quad (6)$$

Comme on l'a fait plus haut pour évaluer une politique, on peut utiliser cette équation pour en tirer directement un algorithme itératif qui calcule la valeur d'une politique optimale π^*.

On obtient ainsi l'algorithme 2 d'itération sur la valeur (IV).

Algorithm 2 L'algorithme d'itération sur la valeur.

Require: un PDM : $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$
Require: un seuil de précision ϵ
initialiser $V_0 \leftarrow 0$
$k \leftarrow 0$
repeat
  for tout état $s \in \mathcal{S}$ **do**
    $V_{k+1}(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s, a, s')[\mathcal{R}(s, a, s') + \gamma V_k(s')]$
  end for
  $k \leftarrow k + 1$
until $||V_k - V_{k-1}||_\infty \leq \frac{\epsilon(1-\gamma)}{2\gamma}$
for tout état $s \in \mathcal{S}$ **do**
  $\pi(s) \leftarrow \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s, a, s')[\mathcal{R}(s, a, s') + \gamma V(s')]$
end for

Intuitivement, cet algorithme se justifie très facilement. Considérons un PDM, basé sur le même processus de décision de Markov, mais dont la fonction à optimiser est à horizon T, soit $R_t(s) = \sum_{k=0}^{k=T-1} [\gamma^k r_{t+k} | s_t = s]$.

Considérons la boucle **répéter** externe. Après la première itération, V_1 contient, pour chaque état, le meilleur retour qui peut être obtenu pour ce problème à horizon de 1 pas, donc après avoir effectué 1 action. À la 2$^{\text{e}}$ itération, V_2 contient le meilleur retour qui peut être obtenu pour un horizon de 2 pas, donc après effectué 2 actions. Après la k$^{\text{e}}$ itération, V_k contient, pour chaque état, le meilleur retour qui peut être obtenu pour un horizon de k pas. Si on fait tendre k vers l'infini, on obtient donc que V_k tend vers V^*. Ainsi, on voit que l'algorithme d'itération sur la valeur peut se voir comme un algorithme considérant une suite de problèmes de décision de Markov avec un horizon qui recule jusque l'infini.

Propriété 3 *Pour tout PDM, l'algorithme d'itération sur la valeur converge asymptotiquement vers V^*.*

À nouveau, cette convergence est asymptotique. Dans la pratique, on va forcément arrêter les itérations au bout d'un certain temps. On aura obtenu une estimation de la fonction valeur de laquelle on déduira une politiqe ϵ-optimale.

Il faut bien prendre garde à la différence entre l'approximation de la fonction valeur, la politique que l'on en déduit en considérant la meilleure action dans chaque état étant donnée

cette approximation (la boucle **pour** à la fin de l'algorithme d'itération sur les valeurs) et la politique optimale.

En guise d'exercice, on appliquera cette approche au problème du chauffeur de taxi et au 21-avec-un-dé.

Pour le chauffeur de taxi, on trouve que $V^*(A) \approx 121{,}65, V^*(B) \approx 135{,}31, V^*(C) \approx 122{,}84$.

3.5 Approche par programmation linéaire

Plusieurs formes de programmes linéaires sont possibles pour exprimer la résolution du problème de planification ; nous en présentons une ci-dessous.

Pour un problème de maximisation, la fonction V^* est la solution du programme linéaire suivant : trouver le vecteur $V \in \mathbb{R}^N$ tel que :

$$
\begin{array}{l}
min \sum_{s=1}^{s=N} V[s] \\
\text{vérifiant } V[s] - \sum_{s'} \gamma \mathcal{P}(s,a,s') V[s'] \geq \sum_{s'} \mathcal{P}(s,a,s') \mathcal{R}(s,a,s'), \forall (s,a) \in \mathcal{S} \times \mathcal{A}
\end{array}
$$

Dans cette formulation, il y a une contrainte associée à chaque paire état-action et une variable par état.

Le problème dual est :

$$
\begin{array}{l}
max \sum_{(s,a)} \sum_{s'} \mathcal{P}(s,a,s') \mathcal{R}(s,a,s') \xi(s,a) \\
\text{vérifiant } \sum_a \xi(s',a) - \sum_{s,a} \gamma \mathcal{P}(s,a,s') \xi(s,a) \leq 1, \forall s' \in \mathcal{S} \\
\text{et } \xi(s,a) \geq 0, \forall (s,a) \in \mathcal{S} \times \mathcal{A}
\end{array}
$$

Il y a une variable (duale) pour chaque paire état-action et une contrainte par état.

Le dual associe donc une valeur $\xi(s,a)$ à chaque paire état-action. Pour un état fixé s, il peut y avoir une action optimale ou plusieurs actions optimales équivalentes. Dans ces deux cas, la variable duale est non nulle si et seulement si l'action est optimale. S'il y a plusieurs actions optimales a et a' dans le même état, alors $\xi(s,a) = \xi(s,a') \neq 0$.

Ainsi, les méthodes de résolution de programmation linéaire qui calculent en même temps la solution du problème et de son dual fournissent directement la fonction valeur d'une politique optimale et une politique optimale.

On sait que si on résout ce programme linéaire par l'algorithme du simplexe, d'un point de vue pratique, il vaut mieux résoudre le problème qui a le moins de contraintes. Le primal a N variables et $N \times P$ contraintes ; le dual a donc $N \times P$ variables et N contraintes. P étant un entier supérieur à 1, il vaut mieux résoudre le dual.

Notons néanmoins que pour résoudre des PDM, l'approche par programmation linéaire n'est pas utilisée dans la pratique à cause de la lourdeur des algorithmes en temps de calcul : les algorithmes d'itération sur les politiques ou sur la valeur sont bien plus rapides en temps d'exécution que les algorithmes de résolution de programmes linéaires. Cependant, l'approche par programmation linéaire est importante d'un point de vue fondamental, en ce qui concerne la

caractérisation de la complexité théorique du problème de décision de Markov : puisque la résolution d'un PDM correspond à la résolution d'un certain programme linéaire et que la résolution d'un programme linéaire est de complexité polynômiale, on en déduit que la résolution d'un PDM est de complexité polynômiale.

En guise d'exercice, on appliquera cette approche au problème du chauffeur de taxi et au 21-avec-un-dé.

4 Résolution dans l'incertain

Dans cette section, on suppose que l'on doit résoudre un PDM spécifié de manière incomplète : on connaît l'ensemble des états et l'ensemble des actions mais on ne connaît pas les fonctions de transition et de retour.

On continue à considérer le PDM $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ et l'objectif est, comme précédemment, de maximiser la somme des retours pondérés par un facteur déprécié au fil du temps : $R(r) = \sum_{k\geq 0} \gamma^k r_{t+k}$ où $\gamma \in [0, 1]$.

Ce qui va changer ici est que l'on va considérer que l'on ne connaît ni $\mathcal{P}$, ni $\mathcal{R}$. On ne va donc pas pouvoir optimiser *a priori* le comportement de l'agent en fonction de ces informations. Au contraire, l'agent va devoir interagir avec son environnement dont la dynamique et les réactions sont décrites par $\mathcal{P}$ et $\mathcal{R}$: l'agent devient un agent *apprenant* qui petit à petit va collecter de l'information qui va lui permettre d'apprendre à agir optimalement. On change donc totalement de perspective puique l'on passe de l'optimisation d'une fonction dont on connaît tous les paramètres à l'estimation d'une fonction à partir d'échantillons. On passe d'un problème d'optimisation de fonction (déterministe) à un problème de statistique.

En effectuant une action a dans l'état courant s à l'instant de décision t, l'agent observe le retour r_t et l'état suivant s_{t+1}. L'environnement étant *a priori* non déterministe, il observe donc une réalisation du processus aléatoire correspondant au retour $\mathcal{R}(s_t, a_t, s_{t+1})$ et une réalisation du processus aléatoire fonction de transition $\mathcal{P}(s_t, a_t, s_{t+1})$. De même que dans une expérience consistant à déterminer le biais d'une pièce en la lançant un certain nombre de fois pour compter combien de fois elle retombe sur chacune de ses faces, on va laisser l'agent réaliser de nombreuses interactions avec son environnement pour, petit à petit, obtenir des estimations des fonctions de transition et de retour. Le lecteur a raison de se dire que cela peut prendre du temps, mais c'est le prix à payer quand on ne dispose pas de $\mathcal{P}$ et $\mathcal{R}$. C'est pour cette raison que l'apprentissage par renforcement est lent.

Ayant réalisé une interaction et ayant donc collecté l'information concernant une transition (s_t, a_t, r_t, s_{t+1}), nous introduisons maintenant la notion fondamentale de *différence temporelle* qui va nous permettre d'estimer la valeur d'une politique. On cherche donc à résoudre le même problème qu'à la section 3.1 mais sans connaître ni $\mathcal{P}$ ni $\mathcal{R}$.

4.1 Différence temporelle

On considère l'équation de Bellman en supposant que tout est déterministe ($\pi, \mathcal{P}, \mathcal{R}$ sont déterministes). Dans ce cas, l'équation se simplifie et devient $V^\pi(s) = \mathcal{R}(s, \pi(s), s') + \gamma V^\pi(s')$. Au long d'une trajectoire, cette équation s'instancie à l'instant t : $V^\pi(s_t) = r_t + \gamma V^\pi(s_{t+1})$, soit $r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t) = 0$

V^π est inconnue. Petit à petit, au fil de l'apprentissage (c'est à cela que sert l'apprentissage ici), on estime sa valeur, en espérant s'approcher de sa vraie valeur. On commence donc avec une valeur pouvant être arbitraire $\widehat{V}$ (comme dans l'algorithme d'évaluation d'une politique ou dans l'algorithe d'itération sur les politiques). Quand l'agent effectue une interaction, en utilisant cette estimation de la valeur $\widehat{V}$, on peut écrire $r_t + \gamma\widehat{V}(s_{t+1}) - \widehat{V}(s_t)$ qui n'a aucune raison d'être nul comme ce terme le serait si $\widehat{V}$ était égal à V^π (dans un environnement déterministe). Par contre, ce terme peut être vu comme une correction à apporter à $\widehat{V}(s_t)$ pour que celui-ci se rapproche de $V^\pi(s_t)$. Faisons-le : $\widehat{V}(s_t) \leftarrow \widehat{V}(s_t) + \alpha[r_t + \gamma\widehat{V}(s_{t+1}) - \widehat{V}(s_t)]$ où α est un coefficient compris entre 0 et 1.

Par ce procédé très simple, si on effectue un grand nombre d'interactions et de telles corrections, $\widehat{V}$ va tendre vers V^π. La convergence est asymptotique comme dans l'algorithme vu plus haut en section 3.1, mais aussi dans les algorithmes de résolution de systèmes d'équations linéaires (*cf.* sec. 3.1).

Ce terme correctif est donc important : il porte le nom de *différence temporelle*. C'est le cœur d'une bonne partie de l'apprentissage par renforcement.

Nous avons raisonné en supposant que l'environnement est déterministe mais on peut tenir le même raisonnement si l'environnement est non déterministe. Sur le principe, cela ne change rien ; dans la pratique, cela fait seulement que l'agent devra effectuer encore plus d'interactions avec son environnement pour estimer correctement V^π que dans un environnement déterministe.

Une remarque concernant la notation : nous avons utilisé et utiliserons désormais la notation $\widehat{v}$ pour indiquer une estimation de la valeur de la variable aléatoire v. Si on reprend l'exemple de la pièce de monnaie : si on la lance 10 fois, elle va par exemple retomber 4 fois sur pile et 6 fois sur face : l'estimation de la probabilité p de retomber sur pile est $\widehat{p} = 4/10$; lançons-la 100 fois et observons 56 faces et 44 piles, on aura $\widehat{p} = 44/100$; et ainsi de suite. On sait intuitivement et plus ou moins rigoureusement selon son bagage en mathématiques (loi des grands nombres) que plus le nombre de lancers augmente, plus la différence entre $\widehat{p}$ et p se réduit.

Cet algorithme que nous décrivons porte le nom de TD. Il est spécifié dans l'algorithme 3.

Pour un PDM fixé et une politique π fixée, V^π calculée par l'algorithme TD converge asymptotiquement vers V^π à condition que :

— chaque état soit visité une infinité de fois,

— on ait $\forall s \in \mathcal{S}$:

$$\sum_{\{t \text{ auxquels l'état } s \text{ est visité}\}} \alpha_t(s, n(s)) = +\infty \tag{7}$$

$$\sum_{\{t \text{ auxquels l'état } s \text{ est visité}\}} \alpha_t^2(s, n(s)) < +\infty \tag{8}$$

Le premier point ($\sum_t \alpha_t(s, n(s)) = +\infty$) indique que l'on visite chaque état une infinité de

Algorithm 3 L'algorithme TD pour une politique déterministe (et stationnaire).

Require: $\mathcal{S}$, $\mathcal{A}$, γ et une politique π.

$\widehat{V}^{\pi}(s) \leftarrow 0, \forall s \in \mathcal{S}$
$n(s) \leftarrow 0, \forall s \in \mathcal{S}$
repeat
  initialiser l'état initial s_0
  $t \leftarrow 0$
  repeat
    émettre l'action $a_t = \pi(s_t)$
    observer r_t et s_{t+1}
    $\widehat{V}^{\pi}(s_t) \leftarrow \widehat{V}^{\pi}(s) + \alpha(s_t, n(s_t))[r_t + \gamma \widehat{V}^{\pi}(s_{t+1}) - \widehat{V}^{\pi}(s_t)]$
    $n(s_t) \leftarrow n(s_t) + 1$
    $t \leftarrow t + 1$
  until s_t est un état final
until critère d'arrêt vérifié.

fois, donc que la loi des grands nombres s'applique. Le second ($\sum_t \alpha_t^2(s, n(s)) < +\infty, \forall s \in \mathcal{S}$ garantit que la somme des corrections est finie pour chaque $\widehat{V}^{\pi}(s)$.

Pour remplir ces deux conditions, on peut prendre :

$$\alpha(s, n(s)) \equiv \frac{1}{n(s) + 1} \tag{9}$$

Ce paramètre α se nomme le *taux d'apprentissage*.

Comme on l'a fait pour la planification, après avoir évalué une politique, on va chercher à apprendre une politique optimale. C'est l'objet de l'algorithme Q-Learning décrit dans la section suivante.

4.2 Q-Learning

Dans cette section, on va introduire l'algorithme Q-Learning, principal algorithme de l'apprentissage par renforcement permettant d'apprendre la politique optimale d'un PDM, sans connaître ni la fonction de transitions $\mathcal{R}$, ni la fonction de retour $\mathcal{R}$.

Comme pour l'évaluation d'une politique dans la section précédente, l'apprentissage de la politique optimale va être obtenu par interaction avec l'environnement. Remarquons qu'ayant estimé la valeur d'une politique sans connaître ni $\mathcal{P}$ ni $\mathcal{R}$ on pourrait espérer l'améliorer et itérer le processus comme dans l'algorithme IP. Seulement, pour améliorer la politique, on a besoin de $\mathcal{P}$ et $\mathcal{R}$ donc il faut pratiquer autrement.

On l'a vu, la qualité d'une paire (état, action) indique si la réalisation de cette action dans cet état en suivant ensuite une certaine politique est bénéfique ou pas. On a donc le sentiment que si l'on connaissait cette fonction qualité, on pourrait déterminer comment agir au mieux. On va donc apprendre/estimer cette fonction qualité. Comme dans la section précédente, sans

connaître ni $\mathcal{P}$ ni $\mathcal{R}$, l'agent va interagir avec son environnement pour estimer Q. Cette fois-ci, on va être capable d'apprendre directement la qualité de la politique optimale grâce à l'équation d'optimalité de Bellman pour Q et en utilisant à nouveau la notion de différence temporelle.

Rappelons cette équation d'optimalité de Bellman pour Q :

$$Q^*(s,a) = \sum_{s' \in \mathcal{S}} \mathcal{P}(s,a,s')[\mathcal{R}(s,a,s') + \gamma \max_{a' \in \mathcal{A}} Q^*(s',a')]$$

En suivant le même raisonnement intuitif que précédemment, on peut dire que l'agent ayant effectué l'action a dans l'état s à l'instant de décision t, il reçoit un retour r_t qui est une réalisation du processus aléatoire $\mathcal{R}(s_t, a_t)$ et transite dans l'état s_{t+1} qui est une réalisation du processus aléatoire $\mathcal{P}(s_t, a_t, s_{t+1})$. On considère que l'on dispose d'une estimation de la fonction Q notée avec un chapeau : $\widehat{Q}$.

On peut écrire que $r_r + \max_{b \in \mathcal{A}} \widehat{Q}(s_{t+1}, b) - \widehat{Q}(s_t, a_t)$ est une différence temporelle, ou encore, une correction que l'on peut appliquer à $\widehat{Q}(s_t, a_t)$.

On démontre (ce n'est pas facile) que si un agent interagit avec son environnement en mettant à jour son estimation de la fonction qualité de cette manière, $\widehat{Q}$ convergera asymptotiquement vers Q.

Cela nous donne l'idée du squelette d'un algorithme :

1. initialiser $Q(s,a), \forall (s,a)$ arbitrairement
2. $t \leftarrow 0$
3. **répéter**
   (a) choisir l'action a_t à effectuer dans l'état courant s_t
   (b) effectuer cette action, observer r_t et s_{t+1}
   (c) mettre à jour Q
   (d) $t \leftarrow t + 1$

Tout est clair et simple dans cet algorithme, sauf la ligne 3.(a) : comment choisit-on l'action ?

Ce choix n'est pas simple, il est l'objet de nombreuses recherches, mais on a quand même des idées simples pour le faire. On se dit que si Q contient une bonne estimation de la fonction qualité, la meilleure action à effectuer dans un état s est celle qui maximise sa qualité $\max_{a \in \mathcal{A}} Q(s,a)$. Au début du fonctionnement de l'algorithme, durant ces premières itérations (« premières » qui peuvent être nombreuses), l'agent ne connaît rien de Q et doit donc découvrir quelles sont les bonnes actions dans les différents états. Il faut donc qu'il essaie : l'algorithme apprend initialement par *essai-erreur*, on dit qu'il *explore*. Quand l'agent sait quelle action est la meilleure (ou qu'il est relativement confiant), il va choisir l'action qu'il croit être la meilleure : il *exploite* la connaissance qu'il a acquise. L'agent est donc confronté à un dilemme au fil de son apprentissage : doit-il exploiter la connaissance déjà acquise ou doit-il explorer pour trouver de meilleures opportunités. C'est tout l'art de la résolution du *dilemme exploration/exploitation*. C'est un sujet central en intelligence artificielle et nous lui consacrons la section suivante avant de revenir vers le Q-Learning en le spécifiant complétement.

4.2.1 Exploration – Exploitation

Il existe de nombreuses stratégies pour gérer le compromis entre l'exploration et l'exploitation. Aucune n'est totalement satisfaisante mais d'expérience, on sait que certaines fonctionnent correctement en pratique. Pour l'instant, les stratégies ayant des fondements théoriques sont moins performantes que les heuristiques que nous présentons ci-dessous.

aléatoire : elle consiste à choisir l'action uniformément aléatoirement parmi les actions possibles dans l'état courant s_t.

Cette stratégie de choix a l'avantage de la simplicité, laquelle est également son inconvénient : elle ne tient aucunement compte de l'expérience accumulée par l'algorithme au fil de son apprentissage (estimation $\hat{Q}$). Le choix d'une action est identique au début de l'apprentissage quand on ne sait rien sur l'environnement, et plus tard, quand l'estimation $\hat{Q}$ est proche de Q donc fiable et permettant de choisir l'action optimale avec certitude.

gloutonne : elle consiste à toujours choisir l'action estimée comme étant la meilleure, soit $a_{\text{gloutonne}}(s_t) = \arg\max_{a \in \mathcal{A}(s_t)} \hat{Q}(s_t, a)$;

Cette stratégie est peut-être encore plus simple que la précédente puisqu'elle est déterministe. On l'a vu, c'est la meilleure stratégie de choix d'action quand l'algorithme connaît Q. Cependant, en attendant, pendant toute la phase d'apprentissage, le choix glouton est mauvais car il limite fortement l'exploration.

ϵ-gloutonne : elle consiste à choisir l'action gloutonne avec une probabilité $1-\epsilon$ et à choisir une action au hasard avec une probabilité ϵ, soit :

$$a_{\epsilon-\text{gloutonne}}(s_t) = \begin{cases} \arg\max_{a \in \mathcal{A}(s_t)} \hat{Q}(s_t, a) \text{ avec probabilité } 1-\epsilon \\ \text{action prise au hasard (uniformément) dans } \mathcal{A}(s_t) \text{ avec probabilité } \epsilon \end{cases} \tag{10}$$

Un cas particulier est la sélection 1-gloutonne qui consiste à choisir une action au hasard. Un autre est la 0-gloutonne qui est la stratégie gloutonne (oui c'est bizarre ce choix de $\epsilon = 0$ pour la stratégie gloutonne, mais c'est comme cela que l'on a l'habitude de faire).

Cette stratégie est utilisée en faisant varier ϵ au fil des itérations (stratégie gloutonne avec ϵ décroissant). En effet, au début de l'apprentissage, quand $\hat{Q}$ n'a pas grand chose à voir avec Q, il faut explorer les actions possibles pour déterminer la meilleure dans chaque état, donc prendre ϵ petit. Par contre, quand $\hat{Q} \approx Q$, on sait que l'action gloutonne est optimale. Cela donne immédiatement l'idée de faire varier doucement ϵ, de 0 en début d'apprentissage à une valeur proche de 1.

Cette stratégie souffre d'un défaut qui est que concernant l'exploration, lors d'un choix d'action non glouton, l'action est choisie au hasard, sans tenir compte de $\widehat{Q}$, donc de ce qui a été appris. Hors $\widehat{Q}$ contient une information qui pourrait, et devrait, guider le choix de l'action.

Très simple à coder, la stratégie de choix ϵ décroissant glouton est très souvent utilisée malgré le défaut qui vient d'être mentionné car elle est finalement efficace en pratique.

softmax : elle consiste à choisir une action en fonction de sa qualité relativement à la qualité des autres actions dans le même état, soit : par exemple, on associe à chaque

action a la probabilité d'être choisie selon l'équation suivante :

$$Pr[a_t|s_t] = \frac{\widehat{Q}(s_t, a_t)}{\sum_{a \in \mathcal{A}(s_t)} \widehat{Q}(s_t, a)}$$

Cette stratégie de choix possède l'avantage d'utiliser toute l'information accumulée dans $\widehat{Q}$ pour choisir une action : c'est une bonne stratégie tant que l'exploration doit être importante. Néanmoins, on peut lui reprocher de ne pas être assez gloutonne quand $\widehat{Q}$ s'approche de Q.

Ainsi, on ressent qu'en début d'apprentissage, tirer les actions au hasard uniformément est probablement une bonne idée, qu'un peu plus tard quand $\widehat{Q}$ commence à contenir des informations fiables, *softmax* est probablement une bonne stratégie, et que pour finir, quand $\widehat{Q}$ est proche de Q, une stratégie (presque) gloutonne est optimale. Cette combinaison est naturellement réalisée par la stratégie suivante :

Boltzmann : c'est un cas particulier de sélection *softmax* : la probabilité est calculée avec l'équation suivante qui produit une distribution dite de Boltzmann :

$$Pr[a_t|s_t] = \frac{e^{\frac{\widehat{Q}(s_t,a_t)}{\tau}}}{\sum_{a \in \mathcal{A}(s_t)} e^{\frac{\widehat{Q}(s_t,a)}{\tau}}}$$

où τ est un réel positif parfois appelé température. Si τ est grand, cette méthode tend vers une sélection aléatoire ; si τ est proche de 0, elle tend alors vers un choix glouton. On utilise donc cette méthode en faisant varier τ au cours de l'apprentissage, en démarrant avec une grande valeur assurant l'exploration de l'ensemble des actions, puis en diminuant progressivement pour augmenter l'exploitation de l'apprentissage déjà effectué.

Prenons un exemple pour illustrer ces différentes méthodes de sélection : supposons que dans l'état courant s_t, 5 actions soient possibles et que :

$$\begin{cases} \widehat{Q}(s_t, a_1) = 23, \\ \widehat{Q}(s_t, a_2) = 21, \\ \widehat{Q}(s_t, a_3) = 12, \\ \widehat{Q}(s_t, a_4) = 3, \\ \widehat{Q}(s_t, a_5) = 1. \end{cases}$$

— La sélection aléatoire choisit l'une des 5 actions, tout à fait au hasard. Chacune a donc 20% de chances d'être sélectionnée.

— La sélection gloutonne choisit a_1.

— En supposant que $\epsilon = 0{,}8$, la sélection ϵ-gloutonne choisira a_1 dans 8 cas sur 10, et une action parmi les 4 autres actions dans 2 cas sur 10.

— La sélection softmax calcule $Pr[a|s_t]$ pour chacune des 5 actions :

	a_1	a_2	a_3	a_4	a_5
$Pr[a\|s_t]$	$\frac{23}{60} \approx 0{,}38$	$\frac{21}{60} = 0{,}35$	$\frac{12}{60} = 0{,}2$	$\frac{3}{60} = 0{,}05$	$\frac{1}{60} \approx 0{,}02$

Autrement dit, les actions a_1 et a_2 ont toutes deux une probabilité importante d'être sélectionnées, tandis que les actions a_4 et a_5 ont une probabilité faible d'être sélectionnées.

— pour la sélection Boltzmann, on va considérer 3 cas : τ grand, moyen ou petit :

	a_1	a_2	a_3	a_4	a_5
$Pr[a\|s_t, \tau = 10^3]$	$\frac{23}{60} \approx 0{,}202$	$\frac{21}{60} \approx 0{,}202$	$\frac{12}{60} \approx 0{,}200$	$\frac{3}{60} \approx 0{,}198$	$\frac{1}{60} \approx 0{,}198$
$Pr[a\|s_t, \tau = 10]$	$\frac{23}{60} \approx 0{,}42$	$\frac{21}{60} \approx 0{,}34$	$\frac{12}{60} \approx 0{,}14$	$\frac{3}{60} \approx 0{,}06$	$\frac{1}{60} \approx 0{,}05$
$Pr[a\|s_t, \tau = 0{,}1]$	$\frac{23}{60} \approx 1$	$\frac{21}{60} \approx 10^{-9}$	$\frac{12}{60} \approx 10^{-48}$	$\frac{3}{60} \approx 10^{-87}$	$\frac{1}{60} \approx 10^{-96}$

Pour $\tau = 1000$, on voit que chaque action est sélectionnée de manière quasi-équiprobable. Pour $\tau = 10$, la meilleure action a une probabilité significativement plus élevée que toute autre d'être sélectionnée, mais chaque action conserve une probabilité significative d'être sélectionnée. Pour $\tau = 0{,}1$, ce n'est plus le cas : seule l'action gloutonne sera sélectionnée dans la pratique.

4.2.2 Algorithme Q-Learning

Nous sommes maintenant prêts à exprimer l'algorithme Q-Learning complétement. Typiquement, après son initialisation, l'agent va réaliser une séquence d'épisodes, c'est-à-dire d'un ensemble d'interactions le menant d'un état initial à un certain état (final ou pas, selon le cas). Quand un épisode est terminé, on en recommence un autre. Il faut imaginer qu'un épisode est une partie d'un jeu par exemple. Progressivement, au fil des épisodes, $\widehat{Q}$ va se rapprocher de Q. La stratégie de choix de l'action doit être bien implantée ; en particulier si on utilise une stratégie gloutonne avec ϵ décroissant, la valeur de ϵ doit être un peu diminuée après chaque épisode. De combien ? Ça dépend de la tâche, ... : il faut essayer et trouver une manière de faire décroître ϵ qui fonctionne bien. On notera que le taux d'apprentissage α introduit plus haut est ici dépendant du nombre de visites à la paire (état, action) courante ; c'est un point important : pour une paire (état, action) déjà visitée de nombreuses fois, Q peut être relativement bien estimé et si la différence temporelle est élevée, c'est probablement l'effet du hasard : il ne faut pas trop pertuber a valeur de Q. Au contraire, pour une paire rarement visitée jusque maintenant, l'estimation de sa qualité est peu fiable et donc on peut corriger sa valeur de manière plus agressive. Toutes ces idées sont naturellement juste des principes qu'il faut ajuster à chaque cas particulier. L'utilisation du Q-Learning, l'apprentissage par renforcement plus généralement, repose beaucoup sur le savoir-faire.

Comme pour l'algorithme TD, $\widehat{Q}$ calculé par l'algorithme Q-Learning converge asymptotiquement vers Q à condition que :

— chaque paire (état, action) soit visitée une infinité de fois ;

— les conditions décrites par les équations (7 et 8) soient respectées.

Notons que si on minimise la fonction objectif, on remplace le max par un min à la ligne 9 et on adapte la stratégie de choix de l'action.

Le Q-Learning est vraiment un algorithme très important dans ce cours. Il faut absolument que le lecteur l'implante par exemple sur le 21-avec-un-dé avant d'aller plus loin. La suite de ce cours ne peut s'appréhender correctement que si le lecteur a réellement implanté et testé l'algorithme avant de la lire.

Illustrons le Q-Learning sur un problème de labyrinthe. Nous considérons le labyrinthe de la

Algorithm 4 L'algorithme *Q-Learning*.

Require: $\mathcal{S}$, $\mathcal{A}$, γ.

$\widehat{Q}(s,a) \leftarrow 0, \forall (s,a)$
$n(s,a) \leftarrow 0, \forall (s,a) \in (\mathcal{S}, \mathcal{A})$
repeat
  initialiser l'état initial s_0
  $t \leftarrow 0$
  while épisode non terminé **do**
    sélectionner l'action a_t et l'émettre
    observer r_t et s_{t+1}
    $\widehat{Q}(s_t,a_t) \leftarrow \widehat{Q}(s_t,a_t) + \alpha(s_t,a_t,n(s_t,a_t))[r_t + \gamma \max_{a' \in \mathcal{A}} \widehat{Q}(s_{t+1},a') - \widehat{Q}(s_t,a_t)]$
    $n(s_t,a_t) \leftarrow n(s_t,a_t) + 1$
    $t \leftarrow t+1$
  end while
until critère d'arrêt vérifié.

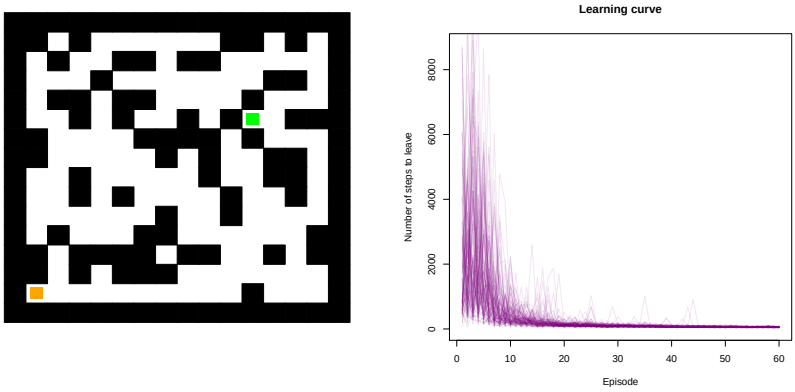


FIGURE 4 – Un labyrinthe et la courbe d'apprentissage du Q-Learning : pour chaque épisode (partir de la case orange et atteindre la case verte), l'ordonnée indique le nombre de pas réalisé par l'agent. Initialement très grand car n'ayant aucune information sur le labyrinthe, ce nombre de pas diminue fortement au bout de quelques atteintes jusqu'à atteindre la case verte selon une trajectoire quasi-optimale.

figure 4. L'agent peut naviguer de case en case et il doit apprendre à aller de la case orange à la case verte. Initialement, le Q-Learning ne disposant d'aucune information quant à la position de la sortie, il va chercher celle-ci ; il va réaliser une exploration aléatoire du labyrinthe et trouver la sortie va prendre du temps. Une fois trouvée, en répétant cette expérience, il va atteindre la sortie de plus en plus vite pour finir par y aller par le chemin le plus court, ou presque. La figure 4 à droite montre la courbe d'apprentissage, c'est-à-dire le nombre de pas réalisés par l'agent au fil des épisodes.

La différence temporelle peut s'interprêter comme un effet de surprise. En effet, l'agent choisit une certaine action a_t à effectuer en s'appuyant sur son estimation courante $\widehat{Q}(s_t, a), a \in \mathcal{A}$: la fonction Q représente les attentes de l'agent quant aux conséquences du choix de l'action a dans l'état s_t. Une fois l'action émise, il observe un retour r_t et l'état suivant. Ces deux éléments lui permettent d'améliorer son estimation de la qualité de l'action a_t dans l'état s_t. Pour cela, l'agent fait la différence entre les conséquences observées de l'émission de l'action a_t dans s_t $(r_t + \gamma \max \widehat{Q}(s_{t+1}, b))$ et ce qu'il attendait avec cette observation $(\widehat{Q}(s_t, a_t))$: si les conséquences de l'action choisie sont conformes à ses attentes, cette différence est petite et la correction apportée à $\widehat{Q}(s_t, a_t)$ est petite ; si les conséquences sont différentes de celles attendues (surprise), la correction est plus importante.

Une propriété remarquable de l'algorithme Q-Learning est sa capacité d'adaptation à un environnement qui change au cours du temps. Supposons qu'une politique ait été apprise ; si l'environnement change, le Q-Learning va s'adapter à son nouvel environnement. Si après apprentissage d'une politique pour le jeu du 21-avec-un-dé on fixe désormais le seuil à 23, le Q-Learning s'adaptera à ce nouveau seuil. De même dans le cas du labyrinthe, on peut bouger la position de la sortie et le Q-Learning va s'adapter. Pour que cette adaptation ait lieu, il est crucial que l'exploration se poursuive, même avec une faible probabilité : il faut toujours que l'algorithme puisse tester des actions qui semblent sous-optimales, et puisse ainsi saisir l'opportunité d'améliorer son comportement. Cela signifie que le comportement de l'agent ne doit jamais être tout à fait optimal : l'optimalité du comportement l'empêcherait de s'adapter à un environnement changeant. Les environnements qui changent dans le temps sont qualifiés de *non stationnaires*. Ceux-ci ne sont pas traités dans ce cours et ils ne sont d'ailleurs pas vraiment traités dans aucun cours : c'est un sujet de recherche.

Nous invitons le lecteur à implanter le Q-Learning pour les deux tâches présentées plus haut et le labyrinthe puis, une fois une politique apprise, à changer l'environnement et observer comment le comportement s'adapte.

Si on regarde bien l'algorithme du Q-Learning (ligne 7 de l'algorithme 4), on note que la politique qu'il suit est définie par la stratégie de choix de l'action. Dans la mise à jour de $Q(s_t, a_t)$ (ligne 9 de l'algorithme 4), l'équation d'optimalité de Bellman est utilisée et celle-ci prend en compte l'action qui semble la meilleure : il n'y a aucune raison que ce soit l'action qui aurait été choisie. C'est une caractéristique qui porte le nom de *off-policy*. Au contraire, un algorithme *on-policy* utilise la même action dans les deux cas. Quelle différence cela fait-il en pratique ? Un algorithme off-policy peut mettre à jour son estimation de $\widehat{Q}$ (ou autre chose) avec des

échantillons collectés par une autre politique que sa politique courante : il peut donc réaliser ses mises à jour avec plus d'échantillons. Un algorithme on-policy ne doit utiliser que des échantillons collectés avec sa politique courante ; à chaque mise à jour, il doit oublier les échantillons collectés précédemment, donc généralement, mettre à jour $\widehat{Q}$ (ou autre chose) avec moins d'échantillons. Pour ce qui est des performances, rien n'est clair : certains algorithmes *on-policy* apprennent finalement plus vite que des *off-policy*, ce qui est un peu le contraire de ce à quoi l'on s'attend. Dans un environnement changeant au cours du temps (non stationnaire), on peut s'attendre à ce qu'un algorithme *on-policy* soit plus réactif qu'un algorithme *off-policy* car il effectue sa mise à jour avec des échantillons plus récents. Dans la section suivante, nous présentons SARSA qui est l'algorithme *on-policy* qui ressemble par ailleurs énormément au Q-Learning.

4.3 SARSA

Nous présentons SARSA, un autre algorithme qui estime itérativement Q. SARSA est très ressemblant au Q-Learning, si ce n'est qu'il est *on-policy*, c'est-à-dire que l'action utilisée dans la mise à jour de $\widehat{Q}$ est l'action qui est exécutée : au lieu de $\max_{a' \in \mathcal{A}} \widehat{Q}(s_{t+1}, a')$ (ligne 9 de l'algorithme Q-learning 4), on a $\widehat{Q}(s_{t+1}, a_{t+1})$.

Algorithm 5 L'algorithme SARSA.

Require: $\mathcal{S}$, $\mathcal{A}$, γ.

1: $\widehat{Q}(s,a) \leftarrow 0, \forall (s,a) \in (\mathcal{S}, \mathcal{A})$
2: $n(s,a) \leftarrow 0, \forall (s,a) \in (\mathcal{S}, \mathcal{A})$
3: **repeat**
4: initialiser l'état initial s_0
5: $t \leftarrow 0$
6: choisir l'action à émettre a_0 en fonction de la politique dérivée de $\widehat{Q}$ (ϵ-gloutonne par exemple)
7: **while** épisode non terminé **do**
8: émettre a_t
9: observer r_t et s_{t+1}
10: choisir l'action à émettre a_{t+1} en fonction de la politique dérivée de $\widehat{Q}$ (ϵ-gloutonne par exemple)
11: $\widehat{Q}(s_t, a_t) \leftarrow \widehat{Q}(s_t, a_t) + \alpha(s_t, a_t, n(s_t, a_t))[r_t + \gamma \widehat{Q}(s_{t+1}, a_{t+1}) - \widehat{Q}(s_t, a_t)]$
12: $n(s_t, a_t) \leftarrow n(s_t, a_t) + 1$
13: $t \leftarrow t + 1$
14: **end while**
15: **until** critère d'arrêt vérifié.

SARSA converge asymptotiquement vers Q dans les mêmes conditions que le Q-Learning.

4.4 Traces d'éligibilité

Le retour perçu à l'instant t, r_t est du à l'action a_t effectuée dans l'état s_t. Cet état est lui-même du à l'émission de l'action a_{t-1} dans l'état s_{t-1} : à ce titre, la paire s_{t-1}, a_{t-1} est un peu responsable de r_t. Et on peut continuer puisque s_{t-1} est du à l'émission de a_{t-2} dans l'état s_{t-2}, et ainsi de suite. Ainsi, r_t est une conséquence de cette séquence d'actions émises dans les états rencontrés successivement dans le passé, (s_{t-k}, a_{t-k}). Intuitivement, on peut se dire que plus on remonte dans le passé (plus k est grand), moins l'influence de l'émission de a_{t-k} est importante sur la perception de r_t. De ces deux intuitions, on peut en déduire qu'il serait pertinent de transmettre à rebours la perception de r_t et mieux encore, la différence temporelle à t, pondérée par un coefficient décroissant avec k. C'est l'idée des traces d'éligilité : à l'itération t, on ne corrige pas seulement $\widehat{Q}(s_t, a_t)$ mais aussi les paires qui ont influencé l'observation de la différence temporelle courante.

Pour cela, chaque paire état-action (s, a) est caractérisée par son éligibilité $e(s, a)$ qui quantifie son influence sur la perception de r_t. Initialement, cette éligibilité est nulle pour toutes les paires. Elle est mise à 1 (ou incrémentée, selon la variante) pour la paire (s_t, a_t) ; à chaque t, les éligibilités sont ensuite multipliées par un coefficient λ et par γ. Ce λ soit être ajusté. Pour le Q-learning, cela donne l'algorithme 6.

Algorithm 6 L'algorithme Q (λ).

Require: $\mathcal{S}$, $\mathcal{A}$, γ, λ.
  initialiser $\widehat{Q} \leftarrow 0$
  for ∞ **do**
    $e(s, a) \leftarrow 0, \forall (s, a) \in \mathcal{S} \times \mathcal{A}$
    $t \leftarrow 0$
    initialiser l'état initial s_0
    choisir l'action à émettre a_0
    repeat
      émettre l'action a_t
      observer r_t et s_{t+1}
      choisir l'action a_{t+1} qui sera émise dans s_{t+1}
      $a^* \leftarrow \arg\max_{a \in \mathcal{A}(s_{t+1})} \widehat{Q}(s_{t+1}, a)$
      $\delta \leftarrow r_t + \gamma \widehat{Q}(s_{t+1}, a^*) - \widehat{Q}(s_t, a_t)$
      $e(s_t, a_t) \leftarrow e(s_t, a_t) + 1$
      for $(s, a) \in \mathcal{S} \times \mathcal{A}$ **do**
        $\widehat{Q}(s, a) \leftarrow \widehat{Q}(s, a) + \alpha \delta e(s, a)$
        $e(s, a) \leftarrow \gamma \lambda e(s, a)$
      end for
      $t \leftarrow t + 1$
    until ...
  end for

4.5 Méthodes de Monte Carlo

Une autre approche, très simple, pour estimer Q consiste à utiliser une méthode de Monte Carlo. D'une manière générale, une méthode de Monte Carlo consiste à simuler un système dynamique pour mesurer une certaine quantité. Dans notre cas, on veut mesurer $Q^\pi(s,a)$ pour toutes les paires (état, action). Dans le cas où on dispose d'un simulateur de la tâche à résoudre (c'est très courant et assez simple à réaliser pour les jeux par exemple), pour estimer $Q^\pi(s,a)$, on simule l'action a dans l'état s et ensuite on agît selon la politique π (on dit aussi que l'on déroule la politique π, ce qui se dit *rollout* en anglais, technique connue sous ce nom). Tout au long de la trajectoire, on va collecter les retours et finalement calculer la somme de ces retours dépréciés ; si l'environnement est non déterministe, on refait cela plusieurs fois pour obtenir une estimation de l'espérance du retour, autrement dit de $Q^\pi(s,a)$. Il s'agît donc d'une quadruple boucle imbriquée illustrée par l'algorithme 7.

Algorithm 7 Principe de l'algorithme de Monte Carlo.

Require: un simulateur de la dynamique basé sur la définition du PDM $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$.

$\widehat{Q}(s,a) \leftarrow 0, \forall (s,a) \in (\mathcal{S}, \mathcal{A})$
for tout état $s \in \mathcal{S}$ **do**
  for toute action $a \in \mathcal{A}$ **do**
    $R \leftarrow 0$
    for i de 0 à $n-1$ **do**
      $e \leftarrow s$
      $b \leftarrow a$
      $j \leftarrow 0$
      somme $\leftarrow 0$
      repeat
        effectuer l'action b dans l'état e, observer le retour r et l'état suivant e'
        somme $\leftarrow$ somme $+\gamma^j r$
        $e \leftarrow e'$
        $b \leftarrow \pi(e)$
      until e est terminal (ou autre critère)
      $R \leftarrow R+$ somme
    end for
    $\widehat{Q}^\pi(s,a) \leftarrow R/n$
  end for
end for

Malgré sa simplicité, cette approche peut donner de surprenamment bons résultats, en particulier si la tâche est déterministe, auquel cas on gagne les deux boucles les plus imbriquées.

5 Résolution approchée : S continu

Jusqu'à maintenant, nous avons traité la situation dans laquelle on peut stocker la valeur de chaque état ou chaque paire (état, action). Pour cela on a utilisé un tableau : `q [s, a]` : ces algorithmes sont qualifiés de *tabulaires*. C'est un cas idéal, voire simpliste : la plupart du temps, la taille de l'ensemble des états est beaucoup trop grand pour que l'on puisse effectivement stocker cette information dans la mémoire d'un ordinateur. Dans cette section, nous dépassons cette limite pour traiter des ensembles d'états et d'actions en principe aussi grand que l'on veut. Entre autres, on traite des cas où l'ensemble des états est $\mathbb{R}$, voire $\mathbb{R}^p$. *Stricto senso*, on quitte donc les PDM définis au début du cours qui, par hypothèse, ont un ensemble d'états (et un ensemble d'actions) fini.

Dans cette section, on va commencer par présenter une approche tabulaire approchée. Si celle-ci n'est pas totalement dépourvue d'intérêt, son application est très limitée. On présentera ensuite l'approche utilisée actuellement dans laquelle on remplace le tableau par une fonction permettant de représentation la valeur, cette fonction étant implantée avec un réseau de neurones. Avant cela, on discute un peu les problèmes posés par une représentation non exacte, approchée, d'une fonction.

5.1 Représentation approchée d'une fonction réelle

5.1.1 Cas non bruité

La valeur est une fonction réelle qui peut être repréentée exactement si son domaine de définition (ensemble des états) est fini et pas trop grand : pour chaque point du domaine de définition, on stocke la valeur de la fonction en ce point.

Supposons que le domaine de définition soit infini : typiquement, on considère alors que c'est un compact sur $\mathbb{R}$, c'est-à-dire un segment de droite, ou sur $\mathbb{R}^P$, c'est-à-dire un hyper-parallélépipède (rectangle si $P = 2$, ...).

Si le domaine de définition est infini, on doit utiliser un modèle de représentation, qui utilise un nombre fini de paramètres. Par exemple, si on veut représenter une droite dans le plan, la droite est un modèle qui a deux paramètres. Plus généralement, on peut considérer un modèle polynomial de degré d avec ses $d + 1$ paramètres. Plus généralement encore, on peut considérer un modèle exprimé par une fonction ayant un certain nombre de paramètres.

Une difficulté est que nous ne connaissons par la forme de la fonction que nous voulons représenter que nous appelerons la fonction cible dans ce document, qui est ici une fonction valeur. Aussi, on doit choisir un modèle pour représenter une fonction que l'on ne connait pas. Il n'y a aucune chance pour que le modèle permette de représenter exactement la fonction cible. Le modèle est décrit par une fonction que nous notons f_m, spécifiée à l'aide d'un ensemble de paramètres $\mathbf{\Theta} = \{\theta_0, ...\theta_k\}$.

La figure 5 illustre cette idée. On suppose que l'on connait la valeur de la fonction cible en un certain nombre N de points : on note les points x_i et la valeur de fonction cible y_i. $y_i \equiv f(x_i)$ et f est inconnue. En haut à gauche de cette figure, on a représenté ces points (x_i, y_i) que l'on connait.

On se donne un modèle, par exemple une droite $y = ax + b$ où $\theta_0 = b$ et $\theta_1 = a$ pour reprendre nos notations et on cherche la droite qui passe « au mieux » par les points (en haut au milieu de la figure). On définit la notion de « au mieux » à l'aide d'une mesure d'erreur : pour chacun des points x_i, pour des paramètres fixés, le modèle prédit $\hat{y}_i = \theta_1 x_i + \theta_0$. On mesure l'erreur de prédiction entre $\hat{y}_i$ et y_i. La mesure la plus classique est $(\hat{y}_i - y_i)^2$. L'erreur totale du modèle est $E = 1/N \sum_i (\hat{y}_i - y_i)^2$: c'est l'erreur quadratique moyenne (MSE en anglais). On peut faire d'autres choix mais on s'en tiendra à celui-ci dans ce cours. On cherche alors les paramètres qui minimisent cette erreur. Puisque $\hat{y}_i = \theta_1 x_i + \theta_0$, on a $E = 1/N \sum_i (\theta_1 x_i + \theta_0 - y_i)^2$; on cherche donc des paramètres qui annulent les dérivées partielles de E par rapport à chacun des paramètres. Pour un modèle linéaire, la solution exacte peut être calculée. Plus généralement, la solution s'obtient numériquement en utilisant un algorithme d'optimisation de fonction (minimisation de E ici). On voit sur la figure que l'approximation ne semble pas parfaite. Cette droite qui minimise le carré des erreurs se nomme la « droite des moindres carrés ».

On voit sur la figure que les points ne semblent pas alignés ni sur cette droite, ni sur aucune. On peut donc essayer d'autres modèles, comme des polynômes. Notons $Poly_d(x) \equiv \sum_{k=0}^{k=d} \theta_k x^k$ un polynôme de degré d. Pour un d donné, on peut chercher les paramètres $\{\theta_k\}$ qui minimisent E de la même manière que précédemment. En cherchant un peu, on trouve un excellent ajustement (en haut à droite dans la figure 5).

Si on n'a pas pensé à essayer d'ajuster un polynôme ou si on veut être dans le vent, ne se poser aucune question et appliquer sans réflêchir un réseau de neurones, on peut le faire. La ligne du bas de la figure 5 indique la prédiction réalisée par trois modèles neuronaux : à gauche avec une couche avec 1 neurone caché avec une fonction d'activation logistique, au milieu avec 4 neurones cachés et à droite avec 18 neurones cachés. Avec 1 neurone, l'approximation est très grossière : on retrouve la forme de la fonction d'activation, rien de plus. J'ai testé un nombre de neurones cachés variant de 1 à 20 ; l'erreur E minimale est obtenue pour 4 neurones cachées (en bas au milieu). Avec 18 neurones cachées, on est en situation de sur-apprentissage : le modèle f_m est trop complexe pour la fonction qui doit être représentée : il prédit n'importe quoi.

Un dernier point qui est un point de détail dans ce cours d'apprentissage par renforcement (dans le sens où le lecteur de ce cours est supposé avoir au préalable suivi et maîtrisé un cours d'apprentissage supervisé), mais qui est un point très important dans un cours d'apprentissage supervisé. Pour ajuster un modèle donné, on génère 100 instances de ce modèle, chacune étant entraînée avec 80% des exemples pris au hasard et les 20% restant étant utilisés pour mesurer l'erreur ; parmi ces 100 instances du même modèle, on retient celle qui donne l'erreur la plus faible sur ces 20%.

5.1.2 Cas bruité

Le problème se complique par le fait que la valeur est bruitée : on ne connait pas la valeur exacte des y_i, mais seulement approximative. On pose l'hypothèse qu'il existe une fonction f qui associe sa valeur à chaque point et que pour chaque x_i, on observe une version bruitée

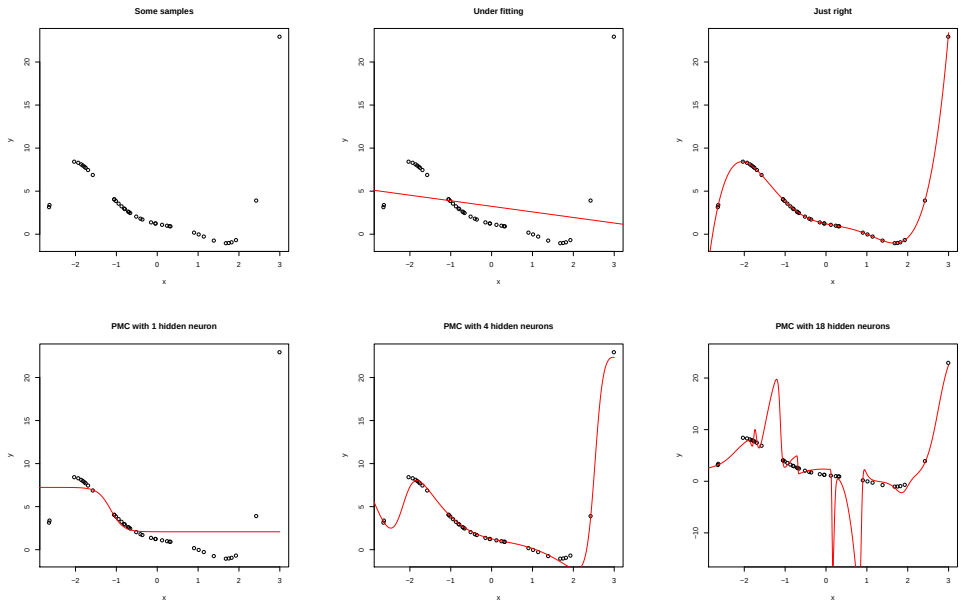


FIGURE 5 – Illustration de l'approximation d'une fonction définie par un ensemble d'échantillons. Voir le texte pour les explications.

(approchée) de $f(x_i)$, $f(x_i) + \eta$ où η est un bruit [1]. Généralement, on suppose que ce bruit ne dépend pas de x_i, qu'il est gaussien, de moyenne nulle et de variance constante (i.i.d.). Pour créer la figure 5, j'ai utilisé une certaine fonction f pour générer les points (x_i, y_i). Pour créer la figure 6, j'ai utilisé la même fonction f et les mêmes x_i et cette fois, $y_i = f(x_i) + \eta$. Ces points sont représentés tout en haut de la figure 6, que l'on peut comparer à ceux en haut à gauche de la figure 5 : on reconnait la forme de la fonction, mais elle est bien moins nette : elle est bruitée.

On peut appliquer la même démarche que précédemment pour ajuster un modèle. Sur la ligne centrale à gauche, on a dessiné la droite des moindres carrées. On voit que cette droite prédit mal chaque point. On est en train d'ajuster un modèle avec 2 paramètres à un jeu de données qui nécessite plus de deux paramètres : on est dans une situation de sous-apprentissage, *under-fitting* en anglais. On peut ajuster un modèle beaucoup plus complexe et on obtient typiquement la figure au milieu de la ligne centrale : ces zigzags sont dus au fait que le modèle est trop complexe/riche et qu'il ne peut que représenter des fonctions ayant cette allure. Ce modèle contient trop de paramètres par rapport aux données disponibles : il fait du sur-apprentissage (*over-fitting* en anglais). Enfin, si on a de la chance, on peut trouver un modèle qui va bien, comme celui illustré à droite de la ligne centrale : ce modèle passe par les points comme « on s'y attend ». Notre attente est liée aux points disponibles ; notre cerveau construit une courbe qui passe au milieu des points et c'est ce modèle que l'on espère trouver : étant donnés les (x_i, y_i) dont on dispose, c'est la courbe la plus probable.

À nouveau, on peut céder à la mode des réseaux de neurones. Comme plus haut, on a testé des réseaux de neurones avec une couche cachée comprenant de 1 à 20 neurones dont la fonction

1. On peut aussi se dire que les x_i sont bruités mais habituellement, on met toute l'incertitude sur y_i.

d'activation est la fonction logistique. Le meilleur ajustement est obtenu avec 4 neurones cachés ; avec plus de 4 neurones cachés, il y a trop de paramètres à ajuster et on est en situation de sur-apprentissage : le modèle f_m est trop complexe pour la fonction qui doit être approximée. La ligne du bas de la figure 6 indique la prédiction réalisée par trois modèles neuronaux : à gauche avec une couche de 4 neurones cachés, au milieu 13 neurone cachés et à droite, 18 neurones cachés.

Ce problème qui consiste à ajuster un modèle pour prédire une valeur réelle associée à un point se nomme le problème de régression. Il relève de l'apprentissage supervisé.

Ce problème a été beaucoup étudié et demeure très étudié. D'une manière générale, on associe une valeur réelle à un point défini dans un sous-espace de $\mathbb{R}^P$. Si on peut diagnostiquer la qualité de l'ajustement à l'aide d'un graphique quand $P = 1$, cela est plus difficile pour $P = 2$, impossible pour P plus grand. On doit alors faire confiance à la mesure d'erreur et on ne peut pas voir si le modèle fait des zigzags ou s'il s'ajuste bien aux données. Il est crucial que le modèle permette de prédire correctement pour des données qui n'ont pas été utilisées pour construire le modèle, qu'il soit capable de généraliser à tout point du domaine. Même s'il est difficile de véritablement juger du résultat, il existe des méthodes pour essayer de bien réaliser cet apprentissage d'un modèle de régression à partir d'un jeu de données et pour diagnostiquer la qualité du modèle appris.

Le problème d'approximer une fonction réelle a été étudié en mathématiques depuis le XIX$^{\text{e}}$ siècle. On a montré qu'il existe des ensembles de fonctions avec lesquelles on peut approximer avec une précision arbitraire n'importe quelle fonction réelle continue par combinaison linéaire. Par exemple, l'ensemble des monômes permet d'approximer aussi précisèment que l'on veut n'importe quelle fonction réelle continue en 1 dimension : c'est le développement en série de Taylor. Il existe d'autres familles, comme l'ensemble des gaussiennes. On sait aussi que l'on peut approximer n'importe quelle fonction réelle continue en combinant linéairement des fonctions logistiques ou des tangentes hyperboliques. C'est exactement ce que fait un perceptron à une couche cachée. Ceci est un argument pour utiliser de tels modèles : on sait que si on utilise suffisamment de neurones cachés, on peut représenter n'importe quelle fonction réelle continue. La difficulté est de savoir combien de neurones utiliser : il n'y a aucune théorie pour cela, seule l'expérience et l'expertise permettent de guider le choix. Pour éviter d'utiliser une couche cachée composée de très nombreux neurones, on peut disposer les neurones en plusieurs couches (perceptrons multi-couches) ayant chacune beaucoup moins de neurones. Plutôt que le nombre de neurones, c'est le nombre de paramètres qui compte pour estimer grossièrement l'équivalence entre un perceptron à une couche cachée et un perceptron avec plusieurs couches cachées [2].

2. On lit parfois qu'un perceptron à une couche cachée (ou à plusieurs couches cachées) peut approximer n'importe quelle fonction : c'est faux. Ce qui est vrai c'est que pour une fonction donnée, pour une précision fixée, il existe au moins un perceptron à une couche cachée qui approxime la fonction avec cette précision. Qu'il y ait une ou plusieurs couches ne change rien au pouvoir de représentation de l'ensemble des perceptrons à couches cachées : on ne peut pas représenter plus de fonctions avec plusieurs couches cachées qu'avec une seule. Utiliser plus d'une couche a pour seul effet de réduire le nombre de neurones par couche : le plus important est le nombre de paramètres et non le nombre de neurones ou leur organisation.

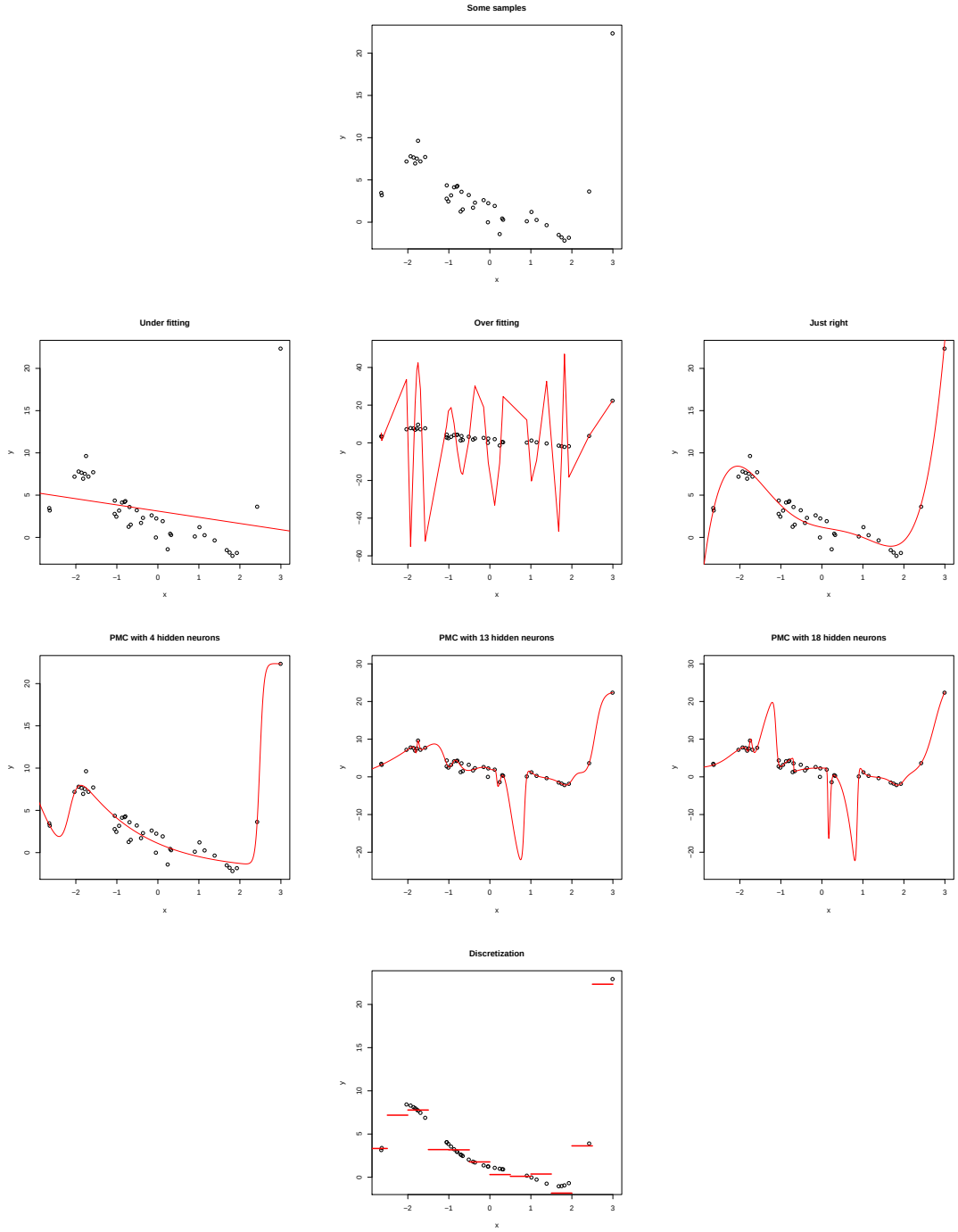


FIGURE 6 – Illustration de l'approximation d'une fonction définie par un ensemble d'échantillons bruités. Voir le texte pour les explications.

Ensemble d'états très grand, voire infini, non continu : on n'aborde pas ici le cas où l'état est (équivalent à) un numéro : c'est un cas très difficile. On rencontre cette situation par exemple dans les jeux, comme le cube de Rubik ou les échecs : le nombre d'états distincts est très grand, mais demeure fini. On rencontre aussi cette situation dans les problèmes combinatoires. Quand on travaille dans un compact de $\mathbb{R}^P$, on suppose que l'espace est métrique. Par contre, dans tous ces problèmes, les états sont seulement numérotés et il n'existe généralement pas de notion de distance pertinente entre deux états ; s'il y en a une, le problème est grandement simplifié.

De la régression à l'apprentissage par renforcement : en apprentissage par renforcement, le problème est plus complexe : on ne connait pas la valeur de la fonction cible en certains points. Cette fonction cible doit être construite progessivement à partir des échantillons (retours) collectés au long de la trajectoire de l'agent. On va adapter les méthodes vues à la section 4.

5.2 Approche tabulaire approchée

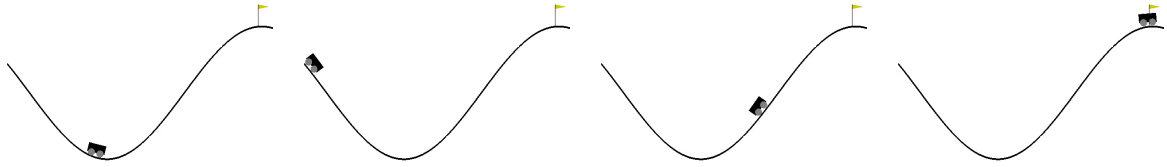
Pour représenter une fonction continue, le modèle le plus simple consiste à discrétiser le domaine de définition de cette fonction en un ensemble de cellules et à associer à chaque cellule une valeur numérique, par exemple la valeur moyenne des valeurs des points de la cellule. Cette idée de discrétisation est illustrée tout en bas de la figure 6 : on a discrétisé le domaine en $n_c = 12$ parties de même taille et pour chacune, on approxime la valeur de y par la moyenne des y_i pour les données localisées dans l'intervalle. La fonction cible est donc représentée par n_c nombres réels.

Pour l'apprentissage par renforcement, l'idée générale est donc d'utiliser un tableau pour représenter la fonction valeur mais cette fois, chaque case du tableau correspond à plusieurs états, voire une infinité d'états distincts. On applique l'un des algorithmes vus plus haut, le *Q-Learning* par exemple. D'une manière générale, il n'y a aucune garantie que cela fonctionne encore. Ça marchera plutôt bien si pour chaque ensemble d'états qui ont été regroupés, leur valeur est la même ; si au contraire des états dont la valeur est significativement différente sont regroupés dans un même élément du tableau représentant la fonction valeur, l'algorithme ne pourra pas apprendre une estimation précise de la fonction valeur pour tout état. Plus la discrétisation est fine (plus n_c est grand), meilleure est l'approximation de la fonction valeur.

Dans la suite de cette section, nous allons discuter de cette approche sur un exemple de PDM, la voiture-dans-la-colline.

5.2.1 La voiture-dans-la-colline

La voiture-sur-la-colline est constituée d'un véhicule peu puissant qui doit atteindre le sommet d'une colline. On suppose qu'il se déplace dans un monde uni-dimensionnel. Situé dans une vallée encaissée, le véhicule ne peut pas directement grimper la colline : il doit petit à petit acquérir de l'inertie pour atteindre ce sommet (*cf.* Fig. 7). Plus précisément, l'espace d'états est un couple (position x, vitesse $\dot{x}$) : $\mathcal{S} = [-1{,}2; 0{,}6] \times [-0{,}07; 0{,}07]$, l'état initial



(a) (b) (c) (d)

FIGURE 7 – Illustration du problème de la voiture-sur-la-colline. Initialement à fond de vallée (a), la voiture doit osciller pour remonter le flanc gauche (b), puis monter la pente côté droit (c) et enfin atteindre son but (d).

étant fixé à $s_0 \in ([-0{,}6; -0{,}4]; 0)$, au hasard en ce qui concerne la position. Il y a 3 actions possibles consistant à accélérer au maximum en avant ou en arrière, ou ne rien faire : $\mathcal{A} = \{-1, 0, +1\}$: c'est une accélération. La dynamique est déterministe et suit la loi de Newton : $\dot{s_{t+1}} \equiv \mathit{bound}(\dot{s_t} + 0{,}001 a_t - \cos(3 s_t))$ et $s_{t+1} \equiv \mathit{bound}(s_t + \dot{s_{t+1}})$. La fonction $\mathit{bound}()$ dénote ici le fait que le véhicule est bloqué dans le domaine de définition de l'état indiqué précédemment. Le retour immédiat $r_t = -1$ à chaque pas de temps. L'objectif est d'atteindre le sommet de la colline ($x \geq 0{,}5$) en moins de 200 pas de temps. S'il n'a pas atteint le sommet de la colline au bout de 200 pas de temps, le véhicule est replacé dans son état initial. La fonction objectif à maximiser est $R = \sum_t r_t$.

5.2.2 Le *Q-Learning* sur la voiture-dans-la-colline

On discrétise l'espace d'états : on découpe chaque dimension en un certain nombre d'intervalles, n_x intervalles pour la position, $n_{\dot{x}}$ intervalles pour la vitesse. Un état $s = (x, \dot{x})$ est associé la case d'indices $(i, j) \equiv (\frac{x - x_{min}}{n_x}, \frac{\dot{x} - \dot{x}_{min}}{n_{\dot{x}}})$. La valeur d'une paire (état, action) est stockée dans l'élément `q [i, j, a]`.

La figure 8 fournit une courbe d'apprentissage avec une discrétisation en 19 intervalles pour la position et 15 intervalles pour la vitesse, pour chacune des 3 actions. Pendant plus de 2000 épisodes, la voiture est incapable d'atteindre le sommet de la colline. Progressivement, elle arrive à l'atteindre de plus en plus rapidement ; au bout de 5000 épisodes, l'algorithme arrive à atteindre le sommet en environ 165 actions. Cela n'est toujours pas une très bonne performance, l'optimum étant inférieur à 100 actions (donc un retour total > -100).

En augmentant la discrétisation, la performance s'améliore (*cf.* table 3). Le temps de calcul augmente.

5.2.3 Discrétisation adaptative

À partir de la valeur des états, notre objectif est d'en déduire une politique gloutonne. En utilisant une discrétisation, on rassemble un ensemble (souvent une infinité) d'états dans une même case de la grille. En termes de politique, cela signifie que l'on associe la même action à tous les états rassemblés dan une même case. La plupart du temps, l'action optimale n'est pas la même pour tous les états d'une même case ; mais avec une telle représentation grossière, on ne discrimine

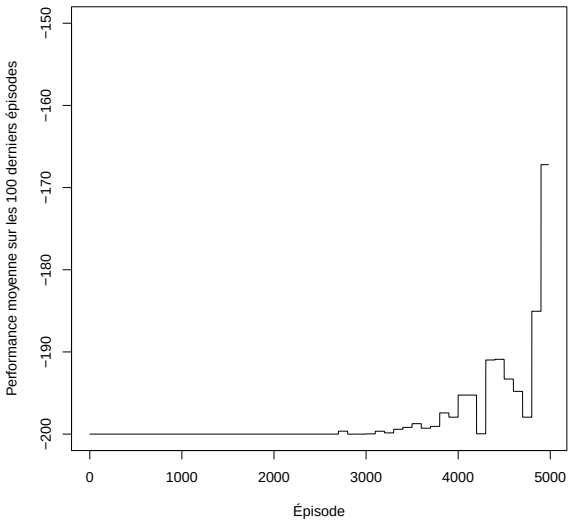
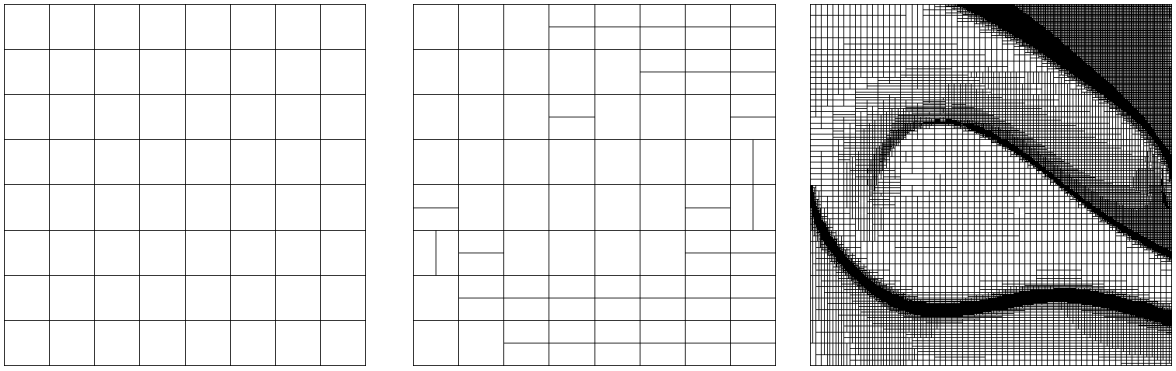


FIGURE 8 – Pour la voiture-sur-la-colline, courbe d'apprentissage typique d'un *Q-Learning* tabulaire utilisant une discrétisation de l'espace d'états : au fil des épisodes, on voit que la performance s'améliore, *i.e.* que la voiture atteint de plus en plus vite le sommet de la colline.

TABLE 3 – Performance du *Q-Learning* pour la voiture-sur-la-colline avec diverses discrétisations. Dans chaque cas, on effectue 20000 épisodes d'apprentissage.

taille de la grille	performance médiane	écart-type
19 x 15	−158,73	7,71
37 x 29	−153,23	6,93
55 x 43	−142,50	4,74



Discrétisation initiale grossière | On découpe quelques cellules | Un peu plus tard

FIGURE 9 – Illustration de l'approche par discrétisation adaptative sur la voiture-sur-la-colline. (source `http://www.cmap.polytechnique.fr/~munos/variable/`.)

pas les états et donc, on ne peut pas représenter la politique optimale pour les différentes états d'une même case. Pour détecter ce type de situation, on peut utiliser l'heuristique suivante : si la valeur estimée pour différents états d'une même case varie significativement, il faut distinguer ces états ; pour cela, on peut découper la case en plus petite case, donc raffiner la discrétisation là où cela semble nécessaire.

Nous n'entrerons pas plus dans les détails ici. L'idée de discrétisation adaptative a été introduite par A. Moore dans l'algorithme *Parti-game* [7], pour lequel R. Munos [8] étudie différentes manières de découper les cellules. Nous illustrons cette idée sur la voiture-sur-la-colline à la figure 9.

5.2.4 Conclusion sur la discrétisation

Les approches par discrétisation ont le mérite d'être assez simples à implanter et très ressemblantes au *Q-Learning* tabulaires. Outre les deux approches qui ont été présentées, on pourrait encore mentionner une approche consistant à utiliser plusieurs discrétisations de même finesse décalées les unes par rapport aux autres (tuilage) ; la valeur d'un état donné est alors la somme de la vaeur estimée dans chaque grille ; en combinant plusieurs grilles grossières, on obtient une discrimination entre états relativement fine. Elles doivent êtres connues par le lecteur car elles peuvent être utiles pour certains problèmes. Néanmoins, elles souffrent de nombreuses limitations. La fonction valeur apprise est très grossièrement approximée ; elle combine la valeur de nombreux états qui n'ont aucune raison d'avoir une valeur proche les uns des autres ; de fait, si des états ayant des valeurs significativement différentes sont regroupés dans un même intervalle, l'algorithme devient aisément instable. Le découpage de l'espace d'états requiert rapidement un espace mémoire important lorsque sa dimension augmente (malédiction de la dimension). L'algorithme met à jour uniquement la valeur de l'état courant ; aussi doit-il visiter, de nombreuses fois, chaque état pour en estimer la valeur : à l'encombrement mémoire s'ajoute donc le temps d'exécution qui peut rapidement devenir rédhibitoire.