Notes de cours d'apprentissage par renforcement

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

Version du 15 décembre 2024

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 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 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 est 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 supervisé 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 d'apprendre à réaliser une tâche. Un agent est donc impliqué et il 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. Quand il est complétement spécifié, la résolution de ce problème 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 :*

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

— $\mathcal{R}$ *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 ; $\mathcal{R}(s,a,s')$ est le retour moyen (espéré) quand on exécute l'action a dans l'état s et que état suivant est s'. On suppose que ce retour est fini.*

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

Selon ce qui nous arrange, on peut aussi définir $\mathcal{R}$ par $\mathcal{R}(s,a)$), le retour moyen (espéré) quand on exécute l'action a dans l'état s ; c'est alors la moyenne des $\mathcal{R}(s,a,s')$, pondérée par les $\mathcal{P}(s,a,s')$. Même si cette notation est plus rarement utilisée, on peut aussi parler de $\mathcal{R}(s)$, le retour immédiat moyen quand on est dans l'état s.

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. Pour certaines tâches comme 21-avec-un-dé, le nombre d'instants de décision est borné, pour d'autres il ne l'est pas.

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 [1] : $\sum_t \gamma^t r_t$ où $\gamma \in [0, 1[$; la trajectoire peut être de longueur infinie du moment que les retours immédiats sont bornés.

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 $\mathcal{R}$ (noté 'R' calligraphié) qui définit les conséquences immédiates d'une action avec ce R (noté 'R' en majuscule d'imprimerie) 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. Or, la définition de la notion d'« état » est extrêmement importante dans le succès, ou l'échec, de l'approche : en effet, la notion même de PDM implique que l'état doit déterminer de manière non ambigüe l'action optimale à effectuer dans un état donné : lors de la modélisation, si le modélisateur constate que dans un état donné, l'algorithme ne peut pas déterminer de manière certaine quelle est l'action optimale mais que parfois c'est une action et parfois une autre, en fonction de ce qui est arrivé précédemment à l'agent, cela signifie que l'état ne contient pas toute l'information concernant l'historique de

1. *discount factor* en anglais.

l'agent et que donc, l'état est mal défini. 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.

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'optimisation portant le nom de *planification* qui sera briévement étudié à la section 3. Dans le second cas, il s'agit 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 :

— a_1 : rouler en espérant être hélé par un client,

— a_2 : s'arrêter à une station de taxis dans la ville où il se trouve et attendre un client,

— a_3 : 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. 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.

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

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

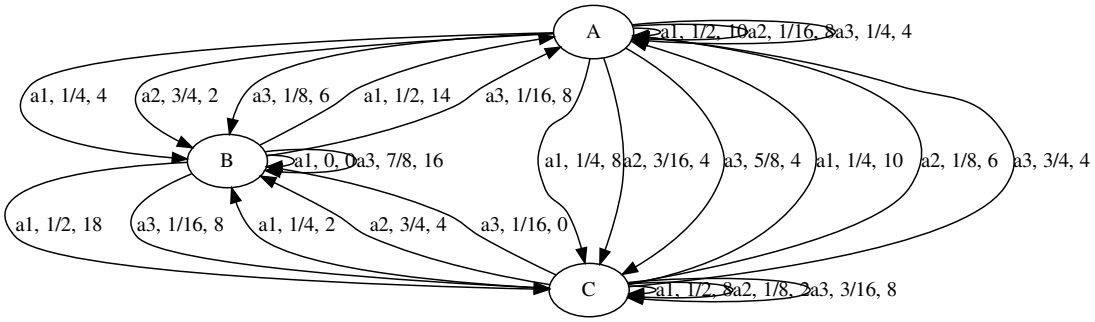


FIGURE 1 – Exemple du chauffeur de taxi : représentation graphique du PDM. Chaque arc représente une transition possible et est étiqueté par l'action qui la provoque, avec la probabilité que cette action dans cet état provoque cette transtion, et le retour espéré sur cette transition.

2.5 Politique

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

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

— politique déterministe : 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 :

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

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 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 $R(t) = \sum_{k \geq 0} \gamma^k r_{t+k}$.

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 un état optimal. 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 le 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 : recommençons cette procédure n fois : on obtient n valeurs de R. Si n est assez grand, ces n valeurs de R 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 $V^\pi(s)$.*

$V^\pi(s) \equiv \mathbb{E}[R(s)|s_0 = s, a_{t \geq 0} \sim \pi]$. $\mathbb{E}[v]$ dénote l'espérance de la variable aléatoire v et la notation $R(s)|s_0 = s$ signifie que l'on s'intéresse à la variable aléatoire $R(s)$ quand l'état initial à $t = 0$ est s.

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

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 (on l'appelle aussi la *qualité*) d'une paire état action (s, a) pour une politique π que nous noterons $Q^\pi(s, a)$.

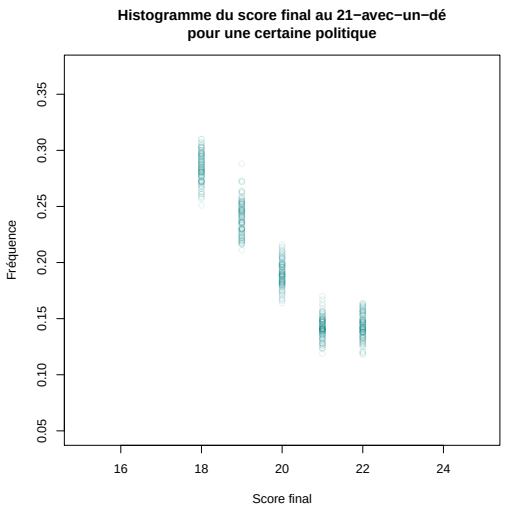


FIGURE 2 – Pour le jeu de 21-avec-un-dé, on a réalisé 100 fois $n = 1000$ parties à partir de l'état 0, en suivant une politique π qui consiste à lancer le dé jusqu'à atteindre le score de 18 et s'arrêter. On représente ici la distribution du score final pour chacune de ces parties en indiquant la variabilité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

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 :*

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

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

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

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 donnés 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 :

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

d'où l'on déduit en remplaçant par les valeurs numériques :

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

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.

D'une manière générale, un système d'équations linéaires s'exprime sous la forme matricielle $\mathbf{Ax} = \mathbf{b}$ où $\mathbf{x} \in \mathbb{R}^N$ est l'inconnue, $\mathbf{b} \in \mathbb{R}^N$ et $\mathbf{A} \in \mathbb{R}^{N \times N}$ sont connues. Résoudre un tel système d'équations consiste à inverser $\mathbf{A}$. Dans les faits, on ne résout jamais le problème de cette manière car l'inversion d'une matrice est à la fois très lourd en temps de calcul, et numériquement très instable.

En guise d'exercice, on exprimera $\mathbf{A}$ et $\mathbf{b}$ pour le calcul de la valeur d'une politique π pour un problème de décision de Markov $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$. Autrement dit, on exprimera chaque élément $A_{i,j}$ et B_i en fonction des éléments de ce quintuplet et de π.

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 bornés 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.

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.

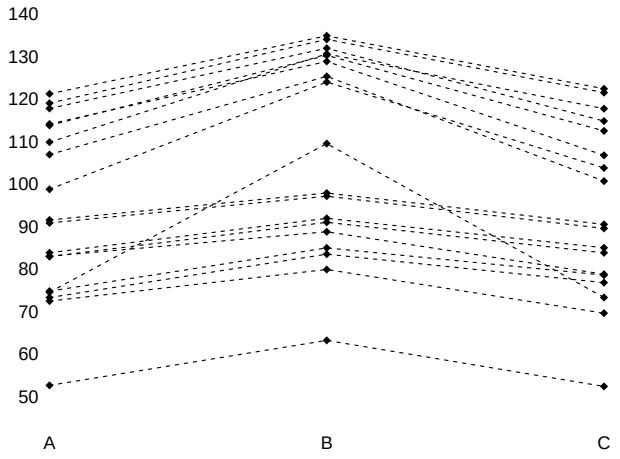


FIGURE 3 – On a représenté la fonction valeur de toutes les politiques stationnaires et déterministes possibles du problème du chauffeur de taxi, pour $\gamma = 0{,}9$. Les valeurs des trois états correspondant à une même politique sont reliées par des traits en pointillées. On voit que l'une de ces 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.

3.1 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 $\max_{s\in\mathcal{S}} |V_{k+1}(s) - V_k(s)|$. Pour montrer que la suite des V_k converge vers V^π, on procède en deux étapes :

1. on montre que $||V_{k+1} - V_k||_\infty \leq \gamma ||V_k - V_{k-1}||_\infty$,
2. on applique le théorème du point fixe qui implique que la suite V_k converge vers un point fixe unique.
3. On sait que ce point fixe est V^π en vertu de l'équation de Bellman.

Pour obtenir une précision ϵ dans l'estimation de V^π, il faut itérer jusqu'à ce que $||V_i^{\pi_k} - V_{i-1}^{\pi_k}||_\infty \leq \epsilon\frac{(1-\gamma)}{2\gamma}$.

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 à évaluer la politique stochastique uniforme pour laquelle on a calculé la fonction valeur à la section 2.9. On vérifiera que le résultat est le même que celui précédemment obtenu.

On invite également le lecteur à reproduire la figure 3.

3.2 Amélioration d'une politique

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

Propriété 1 *Soit la politique*

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

$\pi' \geq \pi$ 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*[2].

On peut écrire cette politique sous la forme : $Q_{\text{gloutonne}}(s) \equiv \arg\max_{a\in\mathcal{A}} Q(s,a)$.

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.

2. *greedy* en anglais.

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 différence entre 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 : $||V^\pi - V^*||_\infty \leq \epsilon$ [3]. 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 12 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 12.

Remarque concernant la manière dont sont exprimés les algorithmes dans ce cours : les algorithmes sont exprimés de manière à être clairs et suffisamment précis pour que leur implantation soit non ambiguë ; nous ne cherchons absolument pas à les optimiser et à les rendre efficaces en temps de calcul. À l'occasion de leur implantation sous la forme d'un programme d'ordinateur qui va être exécuté, ils peuvent naturellement être reformulés, en particulier pour être plus efficaces. En guise d'exemple, lors de l'implantation de l'algorithme 1, nul n'est besoin de mémoriser tous les tableaux V^{π_k} ni toutes les politiques π_k. Seuls deux tableaux stockant l'un V courant et l'autre la prochaine valeur de V sont nécessaires, de même pour la politique. Une autre idée pour implanter efficacement certains algorithmes consiste à transformer des boucles en des opérations vectorielles, quand le langage le permet.

En guise d'exercice, le lecteur est invité à :

— calculer la politique optimale pour le problème du chauffeur de taxi pour $\gamma = 0{,}9$.

— Calculer les différentes politiques optimales obtenues en balayant les valeurs de $\gamma \in [0, 1[$.

— Déterminer les valeurs de γ pour lesquelles la politique optimale change.

— Faire de même pour le problème du 21-avec-un-dé.

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 :

3. Cette notation $||v||_\infty$ signifie : v étant un vecteur, $||v||_\infty \equiv \max_i |v_i|$. Cette notion porte le nom de norme infinie. Il existe des tas de normes ; par exemple $||v||_2$ est la norme euclidienne : $||v||_2 \equiv \sqrt{\sum_i v_i^2}$, et $||v||_1 \equiv \sum_i |v_i|$.

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

Nécessite: un PDM : $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$

Nécessite: un seuil de précision ϵ

initialiser π_0 (aléatoirement ou autrement)
$k \leftarrow 0$
répéter
  initialiser $V_0^{\pi_0}$ (aléatoirement ou autrement)
  $i \leftarrow 0$
  // Évaluation de la valeur de π_k.
  répéter
    pour tout état $s \in \mathcal{S}$ **faire**
      $V_{i+1}^{\pi_k}(s) \leftarrow \sum_{s' \in \mathcal{S}} \mathcal{P}(s, \pi_k(s), s')[\mathcal{R}(s, \pi(s), s') + \gamma V_i^{\pi_k}(s')]$
    fin pour
    $i \leftarrow i + 1$
  jusque $||V_i^{\pi_k} - V_{i-1}^{\pi_k}||_\infty \leq \epsilon \frac{(1-\gamma)}{2\gamma}$
  // Amélioration de π_k.
  pour tout état $s \in \mathcal{S}$ **faire**
    $\pi_{k+1}(s) \leftarrow \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s, a, s')[\mathcal{R}(s, a, s') + \gamma V^{\pi_k}(s')]$
  fin pour
  $k \leftarrow k + 1$
jusque $\pi_k = \pi_{k-1}$

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

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).

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

Nécessite: un PDM : $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$

Nécessite: un seuil de précision ϵ

initialiser $V_0 \leftarrow 0$
$k \leftarrow 0$
// Application itérative de l'opérateur d'optimalité de Bellman.
répéter
  pour tout état $s \in \mathcal{S}$ **faire**
    $V_{k+1}(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s,a,s')[\mathcal{R}(s,a,s') + \gamma V_k(s')]$
  fin pour
  $k \leftarrow k + 1$
jusque $||V_k - V_{k-1}||_\infty \leq \frac{\epsilon(1-\gamma)}{2\gamma}$
$V^* \leftarrow V_k$
// On a obtenu une approximation de V^* à ϵ près. On en déduit π^*.
pour tout état $s \in \mathcal{S}$ **faire**
  $\pi(s) \leftarrow \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s,a,s')[\mathcal{R}(s,a,s') + \gamma V^*(s')]$
fin pour

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 avoir 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 [4] de la fonction valeur

4. Pour mes lecteurs pour lesquels la notion d'« estimation » est vague, voir l'annexe C.

de laquelle on déduira une politique ϵ-optimale. Dans la pratique, il est possible que le seuil ϵ choisi soit trop grand et que la politique qui en est déduite ne soit pas 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 avec $\gamma = 0{,}9$, on trouve que $V^*(A) \approx 121{,}65, V^*(B) \approx 135{,}31, V^*(C) \approx 122{,}84$.

3.5 Approche par programmation linéaire

Pour les lecteurs qui ne connaitraient pas la programmation linéaire, lire d'abord l'annexe D. Plusieurs formes de programmes linéaires sont possibles pour exprimer la résolution du problème de planification ; nous en présentons une ci-dessous.

Reprenons l'équation d'optimalité de Bellman :

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

On en déduit l'inégalité suivante :

$$V^*(s) \geq \sum_{s' \in \mathcal{S}} \mathcal{P}(s, a, s')[\mathcal{R}(s, a, s') + \gamma V^*(s')] \forall s \in \mathcal{S}$$

Cette inégalité indique que V^* est plus grand que le terme de droite. Par ailleurs, V^* doit être égal à la valeur maximale que prend le terme de droite pour une action optimale. Aussi, V^* doit être la plus petite fonction qui vérifie les contraintes, donc que la somme des $V^*(s)$ doit être minmale. On en déduit donc le programme linéaire : trouver le vecteur $V \in \mathbb{R}^N$ tel que :

$min \sum_{s=1}^{s=N} V[s]$
vérifiant $V[s] - \sum_{s'} \mathcal{P}(s, a, s')(\mathcal{R}(s, a, s') + \gamma V[s']) \geq 0 \ \forall (s, a) \in \mathcal{S} \times \mathcal{A}$

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

Le problème dual associe une variable duale $\xi(s, a)$ 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 programmes linéaires 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. Ceci est à comparer à la taille de l'espace des politiques, P^N : le gain en efficacité est donc considérable entre les algorithmes que nous présentons et une exploration brutale de l'espace des poitiques.

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 $V^\pi(s) = \mathcal{R}(s, \pi(s), s') + \gamma V^\pi(s')$. Le 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 [5]. 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. On l'appelle aussi *erreur de Bellman*.

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. Prenons l'exemple d'une 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.

5. Il faut aussi que α diminue au fil des itérations. Ce point est précisé plus loin.

On a donc un algorithme qui évalue la valeur d'une politique en utilisant la TD. On le nomme « algorithme TD », et il est spécifié dans l'algorithme 3.

Algorithme 3 L'algorithme TD pour une politique déterministe (et stationnaire). Nous ne spécifions pas de critère d'arrêt : cela dépend de ce que l'on veut faire, de l'application. Dans le cas le plus simple, ce peut être simplement le fait que l'on a exécuté un certain nombre d'itérations ; sinon, ce peut être le fait que les erreurs TD mesurées pendant le dernier épisode sont inférieures à un certain seuil ; ce critère est vraiment à définir en fonction du contexte d'utilisation de l'algorithme.

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

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

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 $\forall s \in \mathcal{S}$:

$$\sum_{\{t \text{ auxquels l'état } s \text{ est visité}\}} \alpha_t(s, n(s)) = +\infty \tag{6}$$

$$\sum_{\{t \text{ auxquels l'état } s \text{ est visité}\}} \alpha_t^2(s, n(s)) < +\infty \tag{7}$$

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

$$\alpha(s, n(s)) \equiv \frac{1}{n(s) + 1} \tag{8}$$

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

Remarque : dans le cas d'un PDM déterministe, on prend $\alpha = 1$. Le lecteur est invité à réfléchir pourquoi.

Comme on l'a fait pour la planification, après avoir évalué une politique, on va chercher à apprendre une politique optimale. Pour l'instant, on a utilisé $\mathcal{P}$ et $\mathcal{R}$ pour obtenir une meilleure politique que la politique courante. Ici nous supposons ces deux fonctions inconnues ; c'est tout l'objet de l'algorithme Q-Learning décrit dans la section suivante que de réussir à apprendre une politique optimale sans connaître ces deux fonctions.

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 $\mathcal{P}$, 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 sans avoir besoin de connaître ni $\mathcal{P}$ ni $\mathcal{R}$. 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 $\mathcal{R}(s_t, a_t, s_{t+1})$ 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 :

$$a_{\epsilon-\text{gloutonne}}(s_t) = \begin{cases} \arg\max_{a\in\mathcal{A}(s_t)}\hat{Q}(s_t,a) \text{ avec probabilité } 1-\epsilon \\ \text{action prise au hasard (uniformément) dans } \mathcal{A}(s_t) \text{ avec probabilité } \epsilon \end{cases} \quad (9)$$

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 définir cette stratégie).

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 1 en début d'apprentissage à une valeur proche de 0.

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. Or, $\widehat{Q}$ 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.

proportionnelle : elle consiste à choisir une action en fonction de sa qualité relativement à la qualité des autres actions dans le même état : 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, la stratégie proportionnelle 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 :

softmax ou **Boltzmann** : 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 2 cas sur 10, et une action parmi les 5 autres actions dans 8 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 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 perturber la 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.

Algorithme 4 L'algorithme *Q-Learning*.

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

1: $\widehat{Q}(s,a) \leftarrow 0, \forall (s,a)$
2: $n(s,a) \leftarrow 0, \forall (s,a) \in (\mathcal{S}, \mathcal{A})$
3: **répéter**
4: initialiser l'état initial s_0
5: $t \leftarrow 0$
6: **tant-que** épisode non terminé **faire**
7: sélectionner l'action a_t et l'émettre
8: observer r_t et s_{t+1}
9: $\widehat{Q}(s_t,a_t) \leftarrow \widehat{Q}(s_t,a_t) + \alpha(s_t,a_t,n(s_t,a_t))[r_t + \gamma \max_{a' \in \mathcal{A}} \widehat{Q}(s_{t+1},a') - \widehat{Q}(s_t,a_t)]$
10: $n(s_t,a_t) \leftarrow n(s_t,a_t) + 1$
11: $t \leftarrow t+1$
12: **fin tant-que**
13: **jusque** critère d'arrêt vérifié.

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 6 et 7 soient respectées.

Comme pour l'algorithme TD, si le PDM est déterministe, on prend $\alpha = 1$.

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

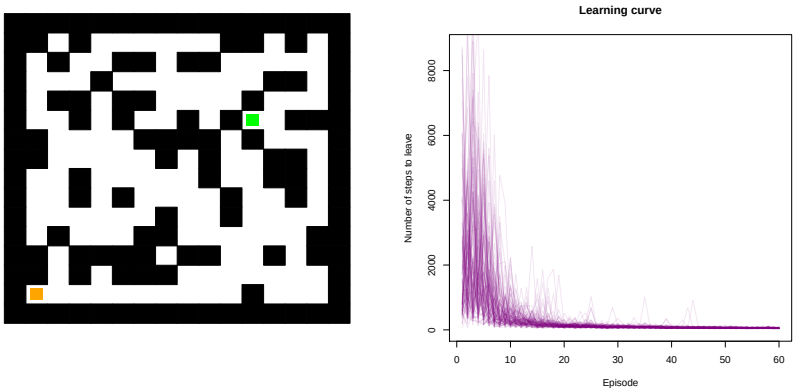


FIGURE 4 – Un labyrinthe et la courbe d'apprentissage du Q-Learning : pour chaque épisode (partir de la case orange en bas à gauche et atteindre la case verte à droite vers le haut), l'ordonnée indique le nombre de pas réalisé par l'agent. Initialement très grand car n'ayant aucune information sur le 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. La figure 5 illustre la capacité d'adaptation du Q-Learning sur le problème du labyrinthe.

Nous invitons le lecteur à implanter le Q-Learning pour les deux tâches présentées plus haut (21-avec-un-dé et chauffeur de taxi) 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. SARSA est spécifié dans l'algorithme 5. Au lieu de $\max_{a' \in \mathcal{A}} \widehat{Q}(s_{t+1}, a')$ (ligne 9 de l'algorithme Q-learning 4), on a $\widehat{Q}(s_{t+1}, a_{t+1})$ (ligne 11 de l'algorithme 5).

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

Nous invitons le lecteur à implanter SARSA pour les tâches présentées plus haut (21-avec-un-dé, chauffeur de taxi et le labyrinthe) et comparer les performances de SARSA avec le Q-Learning.

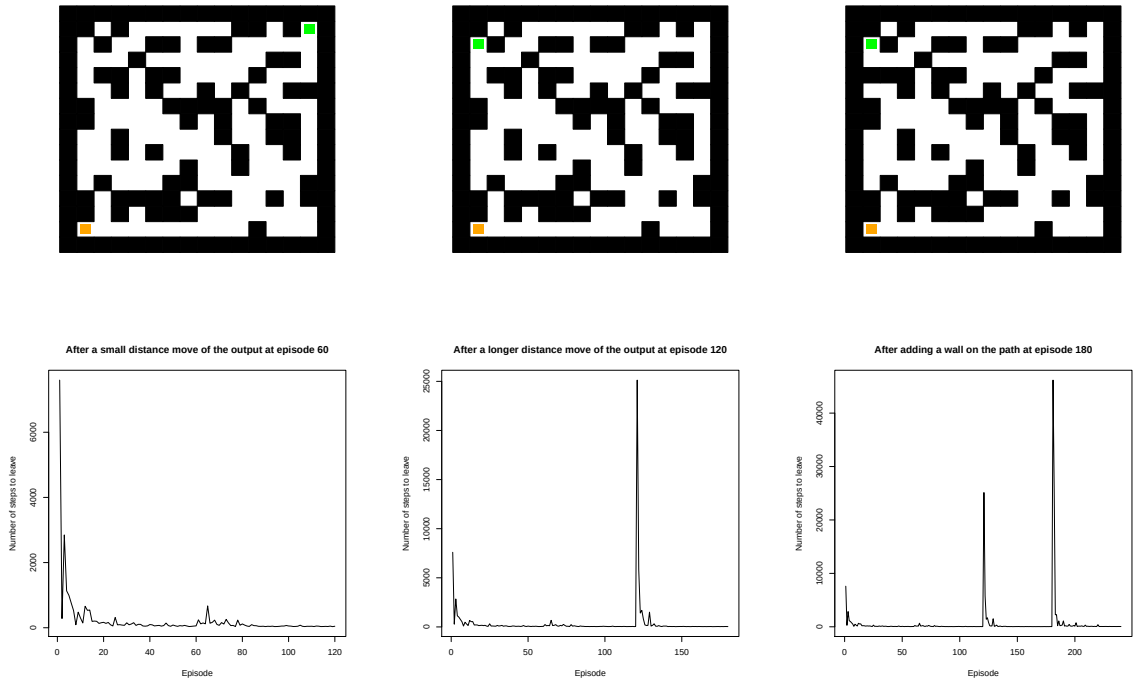


FIGURE 5 – Illustration du comportement du Q-Learning dans un environnement changeant. On reprend le même labyrinthe qu'à la figure 4 pour lequel on a entraîné un Q-learning. La première ligne indique 3 configurations légérement différentes du labyrinthe : la case de départ est toujours la même dans le coin inférieur gauche ; à gauche, la case cible a bougé sur une case proche de sa position précédente, au coin supérieur droit ; au milieu, la case cible a bougé beaucoup plus passant du coin droit au coin gauche ; à droite, cette case cible ne bouge pas mais un mur est ajouté, empêchant le Q-Learning d'atteindre la cible comme il vient d'apprendre à le faire, mais en faisant un grand détour. La seconde ligne indique la courbe d'apprentissage. À gauche, durant les 20 premiers épisodes, il s'agit de la courbe d'apprentissage pour atteindre la position initiale de la cible : c'est la même chose qu'à la figure 4. Ensuite, à l'épisode 60, la position de la cible change : elle est déplacée dans le coin droit, pas très loin de la case cible initiale : au niveau du 60^{e} épisode, on voit un petit pic indiquant que le Q-Learning met un peu plus de temps à trouver la cible ; ce pic n'est pas très élevée car le Q-Learning retrouve rapidement la cible qui n'a pas beaucoup bougé. Au milieu, les 120 premiers épisodes sont les mêmes que la figure de gauche. Au 120^{e} épisodes, la cible bouge loin et on voit cette fois-ci un très grand pic qui indique que le Q-Learning cherche la nouvelle cible : elle a bougé loin de sa position précédente, cela prend donc du temps, d'où la hauteur de ce pic. À droite, les 180 premiers épisodes sont les mêmes qu'au milieu. Au 180^{e} épisode, on bloque la trajectoire suivie par le Q-Learning par un mur ; pour atteindre la cible, il doit apprendre à contourner ce mur et trouver un nouveau chemin vers la cible ; cela lui prend beaucoup de temps, d'où ce pic encore plus haut.

Algorithme 5 L'algorithme SARSA.

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

$\widehat{Q}(s,a) \leftarrow 0, \forall (s,a) \in (\mathcal{S},\mathcal{A})$
$n(s,a) \leftarrow 0, \forall (s,a) \in (\mathcal{S},\mathcal{A})$
répéter
  initialiser l'état initial s_0
  $t \leftarrow 0$
  choisir l'action à émettre a_0 en fonction de la politique dérivée de $\widehat{Q}$ (ϵ-gloutonne par exemple)
  tant-que épisode non terminé **faire**
    émettre a_t
    observer r_t et s_{t+1}
    choisir l'action à émettre a_{t+1} en fonction de la politique dérivée de $\widehat{Q}$ (ϵ-gloutonne par exemple)
    $\widehat{Q}(s_t,a_t) \leftarrow \widehat{Q}(s_t,a_t) + \alpha(s_t,a_t,n(s_t,a_t))[r_t + \gamma \widehat{Q}(s_{t+1},a_{t+1}) - \widehat{Q}(s_t,a_t)]$
    $n(s_t,a_t) \leftarrow n(s_t,a_t) + 1$
    $t \leftarrow t+1$
  fin tant-que
jusque critère d'arrêt vérifié.

4.4 Traces d'éligibilité

Le retour perçu à l'instant t, r_t, est dû à l'action a_t effectuée dans l'état s_t. Cet état est lui-même dû 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.

Algorithme 6 L'algorithme Q (λ).

Nécessite: $\mathcal{S}$, $\mathcal{A}$, γ, λ.

initialiser $\widehat{Q} \leftarrow 0$

pour ∞ **faire**

$e(s,a) \leftarrow 0, \forall (s,a) \in \mathcal{S} \times \mathcal{A}$

$t \leftarrow 0$

initialiser l'état initial s_0

choisir l'action à émettre a_0

répéter

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}

$a^* \leftarrow \arg\max_{a \in \mathcal{A}(s_{t+1})} \widehat{Q}(s_{t+1}, a)$

$\delta \leftarrow r_t + \gamma \widehat{Q}(s_{t+1}, a^*) - \widehat{Q}(s_t, a_t)$

$e(s_t, a_t) \leftarrow e(s_t, a_t) + 1$

pour $(s,a) \in \mathcal{S} \times \mathcal{A}$ **faire**

$\widehat{Q}(s,a) \leftarrow \widehat{Q}(s,a) + \alpha \delta e(s,a)$

$e(s,a) \leftarrow \gamma \lambda e(s,a)$

fin pour

$t \leftarrow t + 1$

jusque critère d'arrêt vérifié

fin pour

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[6], 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.

Algorithme 7 Principe de l'algorithme de Monte Carlo pour estimer la fonction qualité.

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

$\widehat{Q}(s,a) \leftarrow 0, \forall (s,a) \in (\mathcal{S}, \mathcal{A})$
Initialiser n pour que suffisamment de simulations soient réalisées (dépendant du problème)
pour tout état $s \in \mathcal{S}$ **faire**
  pour toute action $a \in \mathcal{A}$ **faire**
    $R \leftarrow 0$
    pour i de 0 à $n-1$ **faire**
      $e \leftarrow s$
      $b \leftarrow a$
      $j \leftarrow 0$
      somme $\leftarrow 0$
      répéter
        effectuer l'action b dans l'état e, observer le retour r et l'état suivant e'
        somme $\leftarrow$ somme $+\gamma^j r$
        $e \leftarrow e'$
        $b \leftarrow \pi(e)$
        $j \leftarrow j+1$
      jusque e est terminal (ou autre critère)
      $R \leftarrow R+$ somme
    fin pour
    $\widehat{Q}^\pi(s,a) \leftarrow R/n$
  fin pour
fin pour

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.

6. *rollout* en anglais.

On laisse le lecteur réfléchir pour compléter cet algorithme pour améliorer la politique obtenue tant que cela est possible.

Une distinction importante à noter entre le Q-Learning et la méthode de Monte Carlo est que le Q-Learning utilise une estimation de Q pour calculer la suivante ; ce n'est pas le cas de la méthode de Monte Carlo. Ce processus qui consiste à construire une succession d'estimations et à utiliser l'estimation courante pour estimer la suivante est qualifié de *bootstrap*. En principe, un processus qui boostrappe va typiquement converger plus vite, mais dans un monde incertain, il risque plus de ne pas converger vers l'optimum attendu, voire de diverger.

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 pour représenter les fonctions valeur et la politique : $V(s)$, $\pi(s)$, et $Q(s,a)$; ces algorithmes sont qualifiés de *tabulaires*. C'est un cas idéal, voire simpliste : la plupart du temps, 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 sensu*, on quitte donc les PDM définis au début du cours qui, par hypothèse, ont un ensemble d'états (et un ensemble d'actions) fini. Cette section s'applique tout autant au problème de planification qu'au problème d'apprentissage.

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 représenter 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

Dans cette section, « non bruitée » s'applique à une fonction dont l'évaluation pour un point donné est toujours la même : $f(x) = C^{\text{ste}}$. Toutes les fonctions ne sont pas ainsi et typiquement, les fonctions qui nous intéressent dans ce cours ne vérifient pas cette propriété. Avant de traiter le cas des fonctions bruitées dans la prochaine section, on présente le cas des fonctions non bruitées.

La valeur est une fonction réelle qui peut être représentée 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 [7] sur $\mathbb{R}$, c'est-à-dire un segment de droite, ou sur $\mathbb{R}^2$, c'est-à-dire un rectangle, ou plus généralement sur $\mathbb{R}^P$, c'est-à-dire un hyper-parallélépipède à P dimensions.

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 polynômial 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 pas la forme de la fonction que nous voulons représenter que nous appellerons 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 6 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 6).

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 réfléchir un réseau de neurones, on peut le faire. La ligne du bas de la figure 6 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

7. De manière très intuitive et non rigoureuse mais suffisante pour comprendre de quoi on veut parler ici avec ce terme, on dira qu'un compact est un sous-ensemble de réels sans trou, par exemple un segment de droite, une surface géométrique sans trou, un volume sans trou dans l'espace, *etc.*

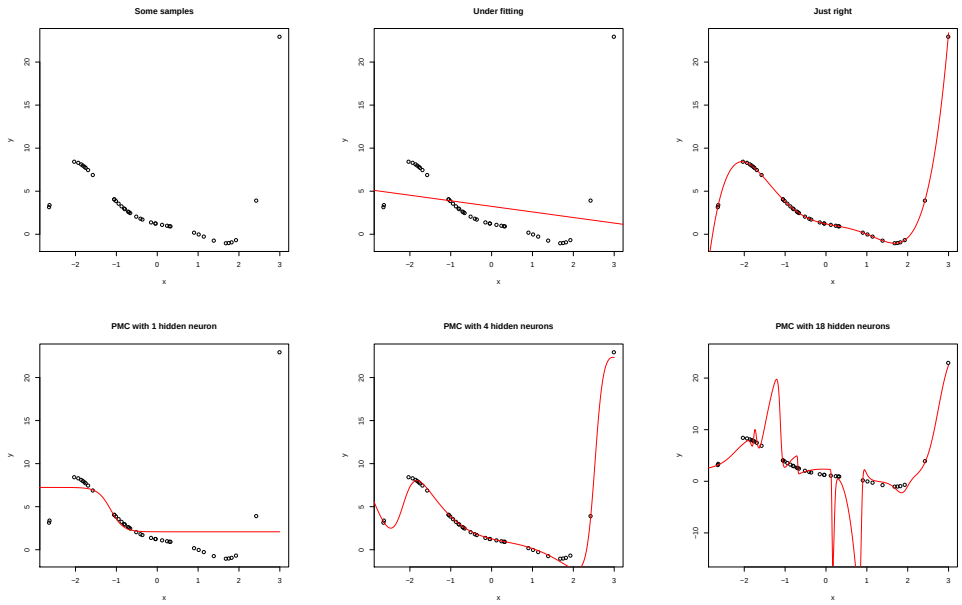


FIGURE 6 – Illustration de l'approximation d'une fonction définie par un ensemble d'échantillons. Voir le texte pour les explications.

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 cachés (en bas au milieu). Avec 18 neurones cachés, 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 a généré 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

« Bruité » signifie qu'à chaque fois que l'on évalue une fonction f en un point fixé x, la valeur de $f(x)$ change. Le problème se complique alors : 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 (approchée) de $f(x_i)$, $f(x_i) + \eta$ où η est un bruit [8]. 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 6, j'ai utilisé une certaine fonction f pour générer les points (x_i, y_i). Pour créer la figure 7, j'ai utilisé

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

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 7, que l'on peut comparer à ceux en haut à gauche de la figure 6 : 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 2e ligne 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 [9]. On peut ajuster un modèle beaucoup plus complexe et on obtient typiquement la figure au milieu de la 2e ligne : 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 [10]. Enfin, si on a de la chance, on peut trouver un modèle qui va bien, comme celui illustré à droite de la 2e ligne : 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 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 3e ligne de la figure 7 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 XIXe 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

9. *under-fitting* en anglais.
10. *over-fitting* en anglais.

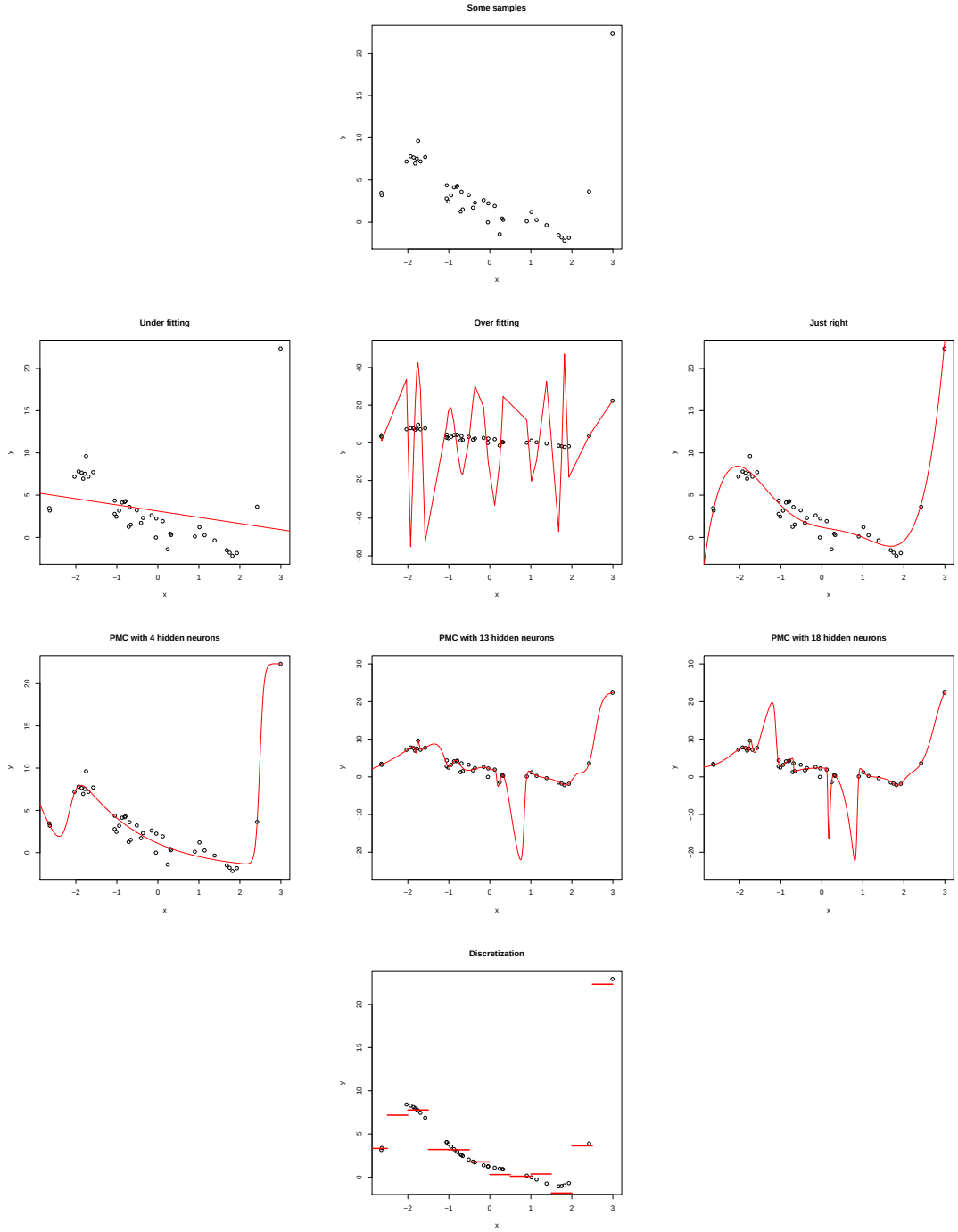


FIGURE 7 – Illustration de l'approximation d'une fonction définie par un ensemble d'échantillons bruités. Voir le texte pour les explications.

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 [11].

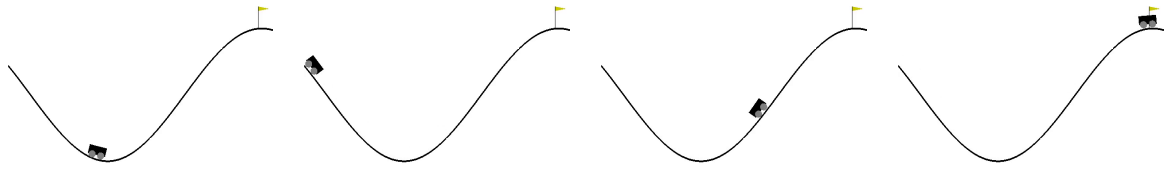
Ensemble d'états très grand, voire infini, non continu : on n'aborde pas ici le cas où l'état est (équivalent à) un numéro : c'est un cas très difficile. On rencontre cette situation par exemple dans les jeux, comme le cube de Rubik ou les échecs : le nombre d'états distincts est très grand, mais demeure fini. On rencontre aussi cette situation dans les problèmes d'optimisation 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 7 : 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.

11. 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.



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

FIGURE 8 – 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).

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. 8). 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 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 $R = \sum_t r_t$.

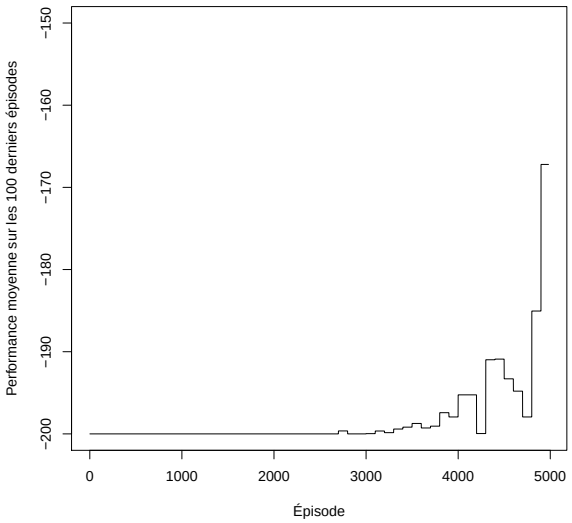


FIGURE 9 – Pour la voiture-sur-la-colline, courbe d'apprentissage typique d'un *Q-Learning* tabulaire utilisant une discrétisation de l'espace d'états : au fil des épisodes, on voit que la performance s'améliore, *i.e.* que la voiture atteint de plus en plus vite le sommet de la colline.

5.2.2 Le *Q-Learning* pour 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 à 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 9 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).

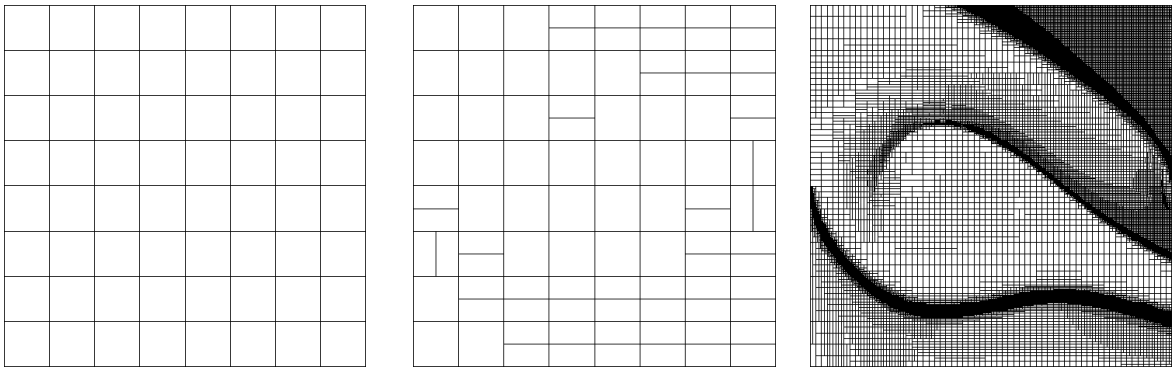
En augmentant la discrétisation, la performance s'améliore (*cf.* table 3). Le temps de calcul augmente.

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 dans 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 pas les états et donc, on ne peut pas représenter la politique optimale pour les dif-

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

férentes 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 petites cases, 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 en apprentissage par renforcement par A. Moore dans l'algorithme *Parti-game* [21], pour lequel R. Munos [23] étudie différentes manières de découper les cellules. Nous illustrons cette idée sur la voiture-sur-la-colline à la figure 10.

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 valeur estimée dans chaque grille ; en combinant plusieurs grilles grossières, on obtient une discrimination entre états relativement fine. Ces techniques doivent être 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.

5.3 Approche avec approximateur de fonctions

Pour dépasser la limite imposée par la mémoire disponible, on va remplacer le tableau `q [s, a]` par une fonction $q(s, a)$. Cette fonction renvoie l'estimation courante de la paire (s, a). Il faut aussi que cette fonction puisse mettre à jour son estimation. Cette fonction (au sens langage de programmation) contient donc une fonction (au sens mathématique) qui calcule cette estimation et dont on peut modifier les paramètres pour que l'estimation soit mise à jour.

Outre la limite imposée par la taille de la mémoire de l'ordinateur utilisée, cette manière de représenter la fonction valeur va permettre la *généralisation* qui est un élément clé de l'apprentissage. En effet, jusqu'à maintenant, étant stockée dans un tableau, $Q(s, a)$ donne de l'information uniquement sur la paire (s, a) ; hors, pour un état s' proche de s, on a généralement raison de penser que $Q(s', a)$ est à peu près égal à $Q(s, a)$: c'est une espèce de principe de continuité qui repose sur la continuité des fonctions de transition $\mathcal{P}$ et de retour $\mathcal{R}$. En représentant Q à l'aide d'une structure qui exhibe une telle propriété de continuité, on permet à l'algorithme de *généraliser* sa connaisance à partir de son expérience : on a observé qu'émettre l'action a dans l'état s produit un retour r et emmène dans l'état s'' ; on en déduit que probablement si on émet a dans un état $s' \equiv s + ds$, le retour observé sera à peu près r et que l'on atteindra un état proche de s''.

Pour cela, l'approche aujourd'hui traditionnelle consiste à utiliser un réseau de neurones qui est entraîné pour représenter une approximation de Q. La fonction `q (s, a)` évoquée ci-dessus est donc une fonction qui calcule la sortie d'un réseau de neurones qui prend s et a en entrée ; la mise à jour de $\widehat{Q}(s, a)$ se fera donc, comme d'habitude pour les réseaux de neurones, par correction des poids. D'autres méthodes d'approximation de fonctions (régression) ont été utilisées mais les réseaux de neurones sont en ce moment la technique préférée.

Ayant dit cela, le principe de la mise en application est immédiat dans les algorithmes de programmation dynamique et dans les algorithmes d'apprentissage par renforcement vus précédemment.

Néanmoins, nous rencontrons une difficulté inédite : contrairement à un élément de tableau dans lequel on stocke une valeur et qu'ensuite, quand on accède à la valeur de cet élément, on retrouve cette même valeur, un réseau de neurones n'est pas un tableau : un réseau de neurones représente une fonction réelle via un ensemble de paramètres (ses poids). Pour modifier sa sortie, on modifie ses poids. Quand on modifie ses poids pour que la sortie du réseau soit celle que l'on veut pour certaines valeurs placées en entrée, on modifie la fonction représentée par le réseau, avec des effets sur l'ensemble de la fonction et non pas seulement sur ces quelques valeurs. C'est cela qui permet la généralisation, mais cela rend son utilisation plus complexe que l'utilisation

d'un tableau. De plus, le réseau est utilisé pour représenter la fonction valeur V ou Q. Celle-ci a une forme que l'on ne connaît pas. Un réseau de neurones peut représenter une certaine famille de fonctions, famille qui dépend de l'architecture du réseau (nombre de couches, nombre de neurones, fonction d'activation). Étant donnée une fonction que l'on souhaite représenter avec un réseau de neurones, on ne sait pas concevoir un réseau qui soit assuré de bien la représenter ; c'est l'expertise du concepteur et des essais qui font que l'on y arrive. Le problème est encore compliqué par le fait que l'on ne dispose que d'échantillons de la fonction, bruités de surcroît. En effet, on ne connait pas la forme de la fonction, on en a seulement des échantillons qui correspondent aux points parcourus par l'algorithme ; on n'a aucune raison de penser *a priori* que cet ensemble de points est suffisant pour bien représenter la fonction. Ainsi, rien ne garantit que le réseau utilisé soit capable de représenter la fonction valeur du problème que l'on essaie de résoudre : le réseau peut être trop simple pour cela. D'une manière générale, le réseau représente une approximation de la fonction valeur. On notera cette approximation $\tilde{V}$ ou $\tilde{Q}$. Il faut bien différencier la notation $\widehat{V}$ et $\tilde{V}$. $\widehat{V}$ est une approximation due au fait que pour connaître la valeur de V, il faudrait réaliser une infinité d'observations de la variable aléatoire. $\tilde{V}$ est une approximation due au fait que l'on essaie de représenter une fonction avec une structure qui n'est pas parfaitement adaptée.

Ajoutons que quand on parle d'approximation, on peut parler de « meilleure » approximation possible d'une fonction réelle étant donné un approximateur de fonction. Pour prendre une image (voir figure 11) : imaginons que nous devions ranger un objet de forme patatoïde dans une boîte parallélépipédique dont on peut déterminer les 3 dimensions : quelle boîte est la « meilleure » ? Il n'y a pas une bonne réponse à cette question, ça dépend ce que l'on veut faire : ça dépend de notre définition de la fonction objectif à optimiser. Pour cette meilleure approximation de la fonction valeur que l'on peut noter $\tilde{V}^*$, c'est un peu la même chose. Dans la pratique, l'algorithme d'apprentissage par renforcement va initialement utiliser une certaine approximation $\tilde{V}_0$ et, itérativement, la mettre à jour, engendrant une suite de $\tilde{V}_k$. On veut que cette suite converge (uniformément [12]) vers la meilleure approximation de V, mais cette volonté est difficile, généralement impossible, à garantir. Autant avec une représentation exacte tabulaire, les algorithmes vus plus haut garantissent cette convergence, autant avec une représentation approchée, en particulier avec un réseau de neurones [13], on sait qu'on ne peut pas garantir cette convergence. Typiquement, au fil des itérations, l'approximation s'approche de l'optimum [14], qu'il ne peut généralement pas atteindre car l'optimum ne peut généralement pas être représenté exactement avec l'approximateur de fonctions utilisé, puis s'en éloigne.

5.4 Programmation dynamique approchée

On met tout d'abord en œuvre cette idée d'approximation de fonction dans le cas de l'algorithme d'itération sur la valeur. Pour celui-ci, l'idée est de simplement remplacer la boucle

12. c'est-à-dire que les éléments de la suite s'approchent du point de convergence d'itération en itération.

13. et plus généralement, quand on utilise un approximateur de fonction non linéaire, c'est-à-dire tous les approximateurs de fonction que l'on utilise dans la pratique.

14. généralement un optimum local.

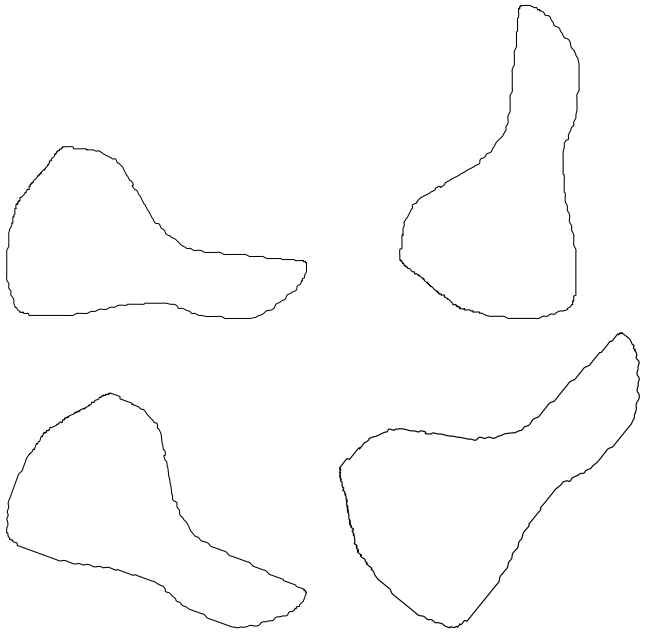


FIGURE 11 – On veut ranger une patate dans une boîte (illustration ici en 2 dimensions) : quelles sont les dimensions de la meilleure boîte ? Cette question très simple est mathématiquement « mal posée ». Prenons l'exemple de la patate représentée en haut à gauche de cette figure : pour la ranger dans une boîte, celle-ci doit être de dimensions $79 \times 45{,}5$; si on la tourne, la dimension de la boîte change : tout le monde le sait ! Quelle est la bonne dimension ? Pour que cette question ait une seule réponse, pour qu'elle soit bien posée au sens mathématique du terme, on doit ajouter des contraintes. Par exemple, on peut chercher la boîte de périmètre minimal, ou maximal ; ou la boîte d'aire minimale/maximale. Le problème a alors une seule solution : il est bien posé : quand on utilise le mot « meilleur », on sait maintenant ce que l'on veut dire. Un problème mathématiquement bien posé (au sens d'Hadamard qui a introduit cette notion au début du XX^{e} siècle) possède une et une seule solution.

pour des lignes 5 à 7 de l'algorithme 2 (page 19) par la collecte de couples (s, v) où s est un état et v et la valeur du membre droit de l'affectation de la ligne 6. Comme par hypothèse il y a maintenant trop d'états pour tous les parcourir et stocker une information pour chacun d'eux, on échantillonne l'ensemble des états. On constitue ainsi un jeu d'entraînement pour résoudre un problème de régression. Ici, on utilise un réseau de neurones, mais on pourrait utiliser n'importe quelle autre méthode de régression.

Algorithme 8 Principe de l'algorithme d'itération sur la valeur approchée. e est un ensemble d'exemples, chaque élément étant un couple (état, estimation de la valeur de cet état). À l'issue de cet algorithme, on a entraîné un réseau de neurones qui, quand on place un état sur son entrée, sa sortie fournit une estimation de sa valeur. Comparer cet algorithme à sa version exacte tabulaire (l'algorithme 2).

Nécessite: un PDM : $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$
Nécessite: un réseau de neurones
1: initialiser les poids de ce réseau de neurones qui représente $\widehat{V}$
2: **répéter**
3: **pour** des états $s \in \mathcal{S}$ échantillonnés indépendemment les uns des autres **faire**
4: $e[i].s \leftarrow s$
5: $e[i].v \leftarrow \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s, a, s')[\mathcal{R}(s, a, s') + \gamma \widehat{V}(s')]$
6: **fin pour**
7: entraîner le réseau de neurones avec les exemples contenus dans e
8: **jusque** critère d'arrêt vérifié

Cette régression fournit une estimation de V ; en itérant, cette approximation s'améliore ... pendant un certain nombre d'itérations. On n'a aucune garantie sur l'approximation ; en fait tout dépend du réseau de neurones utilisé (est-il capable d'approximer correctement la fonction valeur ?) et de l'échantillonnage d'états. Si on s'y prend mal, en continuant d'itérer, l'approximation peut diverger.

Si on peut démontrer certains résultats de convergence pour les algorithmes exacts vus précédemment, ce n'est plus le cas pour les algorithmes approchés qui utilisent un approximateur de fonctions non linéaire, tel un réseau de neurones. La fonction valeur étant typiquement non linéaire, on utilise un approximateur non linéaire.

En guise d'exercice, on encourage le lecteur à réfléchir à l'adaptation de l'algorithme d'itération sur les politiques dans une version approchée.

5.5 Apprentissage par renforcement approché

L'idée de représenter $\widehat{V}$ ou $\widehat{Q}$ par un réseau de neurones a été formulée très tôt, dès la fin de la décennie 1980, peu après la diffusion de l'algorithme de rétropropagation du gradient de l'erreur à partir de sa publicaiton en 1986. Pour que cela devienne facile à utiliser, il a fallu accumuler de l'expertise sur l'utilisation des réseaux de neurones dans le contexte de l'apprentissage par renforcement, et un tas de trucs qui sont ébauchés ici. Mais l'idée est extrêmement simple.

5.5.1 *Fitted Q-iteration* : FQI neuronal

Dans l'algorithme 4 du Q-Learning, la mise à jour de $\widehat{Q}(s_t, a_t)$ se fait en utilisant la transition courante (s_t, a_t, r_t, s_{t+1}). Remplaçant la table par un réseau de neurones représentant $\widehat{Q}$, on peut mettre à jour les poids du réseau de neurones pour que la sortie s'ajuste à la nouvelle valeur. Cette idée a été mise en œuvre et a notamment mené très tôt à des succès impressionnants (TD-Gammon au début de la décennie 1990). Néanmoins, agir ainsi rend l'algorithme instable. Il est beaucoup mieux de réaliser la mise à jour des poids du réseau en utilisant plusieurs transitions ; cela revient à résoudre un problème de régression classique. L'idée est précisée dans l'algorithme 9.

Algorithme 9 Principe de l'algorithme *Fitted Q-Iteration*.

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

1: initialiser les poids du réseau de neurones représentant $\widehat{Q}$
2: **répéter**
3: initialiser l'état initial s_0
4: $t \leftarrow 0$
5: $\mathcal{D} \leftarrow \emptyset$
6: **répéter**
7: choisir l'action à émettre a_t et l'émettre
8: observer r_t et s_{t+1}
9: ajouter le couple $(e_t \equiv (s_t, q_t), q_t \equiv r_t + \gamma \max_{a' \in \mathcal{A}} \widehat{Q}(s', a'))$ à $\mathcal{D}$
10: $t \leftarrow t + 1$
11: **jusque** fin de l'épisode
12: mettre à jour les poids du réseau de neurones avec les données stockées dans $\mathcal{D}$
13: **jusque** critère d'arrêt vérifié.

À la ligne 7, le choix est réalisé par exemple en suivant une stratégie ϵ-décroissant glouton. Les estimations de la qualité Q sont obtenues via le réseau de neurones.

À la ligne 12, la régression a pour objectif de minimiser l'erreur quadratique moyenne : $\mathcal{D}$ contient des couples (entrée du réseau e, valeur attendue en sortie pour cette entrée q) ; pour l'un de ces couples, en plaçant e en entrée du réseau de neurone, celui-ci produit la sortie o ; typiquement $o \neq q$ et le but est de modifier les poids du réseau pour diminuer l'écart entre la sortie o produite par le réseau et la vaeur de sortie attendue q. Plus précisément, pour minimiser la MSE, on cherche à minimiser $\frac{1}{|\mathcal{D}|} \sum_{i=1}^{i=|\mathcal{D}|} (q_i - o_i)^2$. Pour un réseau de neurones, on utilise généralement un algorithme de rétropropagation du gradient de l'erreur pour minimiser cette quantité, autrement dit, une descente de gradient (*cf.* appendice A.1 pour un rafraîchissement concernant cette méthode d'optimisation).

Notons qu'en principe, n'importe quelle méthode de régression peut être utilisée. Actuellement, les réseaux de neurones sont utilisés. D'autres méthodes ont été explorées dans le passé (en particulier forêts aléatoires et méthodes à noyau).

5.5.2 *Deep Q-Network* : DQN

Aujourd'hui, les réseaux de neurones sont qualifiés de profonds (*deep*), même si les réseaux utilisés en apprentissage par renforcement ne sont pas très profonds : avec quelques couches, cela reste des perceptrons multi-couches tout à fait traditionnels. Néanmoins, l'algorithme *Deep Q-network* (DQN) est devenu un nom populaire pour qualifer ce type d'algorithmes, qui ressemblent beaucoup au FQI neuronal vu précédemment, qui est lui-même un Q-Learning non tabulaire. Notons que ce que l'on nomme DQN d'une manière générale varie un peu d'un auteur à un autre : DQN peut signifier l'algorithme basique présenté ici (on remplace la table `Q` par un réseau de neurones) ou l'un des algorithmes qui en dérivent qui sont présentés dans les prochaines sections.

La figure 12 compare une version synthétisée de l'algorithme 4 décrivant le *Q-Learning* tabulaire et DQN. Les différences sont surlignées en jaune : on voit qu'elles sont très peu nombreuses : l'essentiel de l'algorithme est le même.

Le réseau de neurones prend ici en entrée un état et une action et produit en sortie une estimation de sa qualité.

Ce qui a vraiment évolué est la maîtrise des réseaux de neurones dans le contexte de l'apprentissage par renforcement ; un ensemble de trucs sont maintenant connus qui assurent de bien meilleures performances et une plus grande stabilité d'apprentissage, donc une plus grande facilité d'utilisation. De même, il y a eu de nombreux progrès quant à l'algorithme d'entraînement des réseaux de neurones à plusieurs couches cachées. La facilité d'utilisation des réseaux de neurones est très amplifiée par la disponibilité de deux bibliothèques de réseaux de neurones ouvertes (pytorch et tensorflow) et les très nombreux exemples aujourd'hui disponibles sur Internet qui en illustrent de multiples facettes et de multiples applications. Dans la suite de cette section, on présente très brièvement quelques points essentiels pour une mise en œuvre réussie d'un algorithme d'apprentissage par renforcement. Tout est basé sur l'utilisation d'un réseau de neurones pour représenter $\widehat{Q}$ mais est généralement transposable à l'utilisation d'un autre type d'approximateur de fonctions.

5.5.3 *Replay buffer*

Dans DQN, un réseau de neurones est entraîné pour représenter la fonction valeur Q. Chaque transition (s_t, a_t, r_t, s_{t+1}) est utilisée pour mettre à jour $\widehat{Q}$, donc les poids du réseau, selon l'équation de la différence temporelle. Si on a initialement mis à jour les poids après chaque transition avec les informations correspondant à cette dernière transition, on a maintenant compris que c'est une mauvaise idée dans la pratique. En effet, on sait que pour entraîner efficacement un réseau de neurones, il vaut mieux l'entraîner sur un ensemble de données, soit un ensemble de transitions ; il faut aussi que les données soient i.i.d., ce qui n'est pas le cas si on considère les transitions d'une trajectoire : constituant une trajectoire, ces transitions ne peuvent pas être i.i.d : elles ne sont pas échantillonnées uniformément au hasard dans l'ensemble des transitions mais se suivent. Dans une tâche épisodique, il faut mélanger des transitions issues de plusieurs trajectoires ; d'une manière ou d'une autre, il faut que les échantillons soient les plus indépendants les uns des autres qu'il est possible. Cela se traduit dans la notion de *replay buffer* où

Q-Learning tabulaire	DQN
Initialiser le tableau $\widehat{Q}$ $n(s,a) \leftarrow 0, \forall (s,a) \in (\mathcal{S}, \mathcal{A})$ **répéter** initialiser l'état initial s_0 $t \leftarrow 0$ **tant-que** épisode non terminé **faire** • sélectionner a_t et l'émettre • observer r_t et s_{t+1} • $\widehat{Q} \leftarrow \widehat{Q} + \alpha[r_t + \gamma \max_{a'} \widehat{Q} - \widehat{Q}]$ • $t \leftarrow t+1$ **fin tant-que** **jusque** critère d'arrêt vérifié.	initialiser les paramètres θ de $\widehat{Q}$ **répéter** initialiser l'état initial s_0 $t \leftarrow 0$ **tant-que** épisode non terminé **faire** • sélectionner a_t et l'émettre • observer r_t et s_{t+1} • mettre à jour les θ pour minimiser l'erreur de prédiction (la différence temporelle) pour cette paire (s_t, a_t) • $t \leftarrow t+1$ **fin tant-que** **jusque** critère d'arrêt vérifié.

FIGURE 12 – Les différences essentielles entre le *Q-Learning* tabulaire (à gauche) et DQN (à droite) sont surlignées en jaune dans DQN. On considère ici une tâche épisodique.

on stocke dans un sac $\mathcal{D}$ les transitions et où on entraîne le réseau sur un échantillon de ces interactions.

La figure 13 indique l'intégration d'un *replay buffer* dans DQN.

5.5.4 *Target network*

Utiliser le réseau de neurones pour choisir l'action à émettre tout en le mettant à jour rend l'algorithme instable. Pour diminuer ce problème, on utilise deux réseaux : l'un est utilisé pour choisir l'action ; l'autre est mis à jour avec les données d'interaction stockées dans $\mathcal{D}$. De temps en temps, on échange le rôle de ces deux réseaux. Cet échange a lieu lorsque l'on estime que le second réseau a été entraîné avec suffisamment de nouvelles données d'interaction.

La figure 14 indique l'intégration de cette technique dans le DQN avec *replay buffer* précédent : les différences sont surlignées en jaune.

5.5.5 *Double DQN*

À cause du max dans l'équation de mise à jour, le Q-Learning a tendance à sur-estimer la valeur des paires (état, action). Ce phénomène est connu depuis le début des années 1990 quand on utilise un approximateur de fonctions dans le Q-Learning [46].

Reprenons le Q-Learning tabulaire et les versions de DQN vues jusqu'à présent : pour tous ces algorithmes, la mise à jour des paramètres du modèles repose sur une estimation de l'erreur de prédiction (la différence temporelle) de Q de la forme : $r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)$.

Pour les DQN vus jusque maintenant, paramétrés par un certain θ, on peut écrire plus précisément : $r_t + \gamma \max_a Q^\theta(s_{t+1}, a) - Q^\theta(s_t, a_t)$.

DDQN consiste à estimer la différence temporelle en distinguant les deux utilisations de Q :

DQN	DQN avec *replay buffer*
initialiser les paramètres θ de $\widehat{Q}$ **répéter** initialiser l'état initial s_0 $t \leftarrow 0$ **tant-que** épisode non terminé **faire** • sélectionner a_t et l'émettre • observer r_t et s_{t+1} • mettre à jour les θ ... • $t \leftarrow t+1$ **fin tant-que** **jusque** critère d'arrêt vérifié.	initialiser les paramètres θ de $\widehat{Q}$ **répéter** $\mathcal{D} \leftarrow \emptyset$ **pour** $k \in \{1, ...K\}$ **faire** $t \leftarrow 0$ initialiser l'état initial s_0 **tant-que** épisode non terminé **faire** • sélectionner a_t et l'émettre • observer r_t et s_{t+1} • ajouter (s_t, a_t, r_t, s_{t+1}) à $\mathcal{D}$ • mettre à jour les θ pour minimiser la MSE sur un échantillon des données contenues dans $\mathcal{D}$ • $t \leftarrow t+1$ **fin tant-que** **fin pour** **jusque** critère d'arrêt vérifié.

FIGURE 13 – DQN avec *replay buffer* pour une tâche épisodique : les différences avec le DQN basique sont surlignées.

DQN avec *replay buffer*	DQN avec *replay buffer* et *target network*
initialiser les paramètres θ de $\widehat{Q}$ **répéter** $\mathcal{D} \leftarrow \emptyset$ **pour** $k \in \{1, ...K\}$ **faire** $t \leftarrow 0$ initialiser l'état initial s_0 **tant-que** épisode non terminé **faire** • sélectionner a_t et l'émettre • observer r_t et s_{t+1} • ajouter à $\mathcal{D}$ la transition (s_t, a_t, r_t, s_{t+1}) • mettre à jour les θ pour minimiser la MSE sur un échantillon des données contenues dans $\mathcal{D}$ • $t \leftarrow t + 1$ **fin tant-que** **fin pour** **jusque** critère d'arrêt vérifié.	initialiser deux jeux de paramètres θ' et θ'' **répéter** $\mathcal{D} \leftarrow \emptyset$ **pour** $k \in \{1, ...K\}$ **faire** $t \leftarrow 0$ initialiser l'état initial s_0 **tant-que** épisode non terminé **faire** • sélectionner a_t en utilisant θ' et l'émettre • observer r_t et s_{t+1} • ajouter à $\mathcal{D}$ la transition (s_t, a_t, r_t, s_{t+1}) • mettre à jour les θ'' pour minimiser la MSE sur un échantillon des données contenues dans $\mathcal{D}$ • $t \leftarrow t + 1$ **fin tant-que** **fin pour** de temps en temps, échanger θ' et θ'' **jusque** critère d'arrêt vérifié.

FIGURE 14 – DQN avec *replay buffer* et *target network* pour une tâche épisodique : les différences avec le DQN avec *replay buffer* sont surlignées.

$r_t + \gamma \max_a Q^{\theta'}(s_{t+1}, a) - Q^{\theta''}(s_t, a_t)$, en reprenant les notations de l'algorithme DQN avec *target network* vu ci-dessus.

Nous laissons le lecteur écrire et implanter un algorithme correspondant à cette idée.

5.5.6 *Prioritized Replay Buffer*

Le dernier truc dont nous parlons ici consiste en une amélioration du principe de *Replay Buffer*. Dans celui-ci, on sélectionne un sous-ensemble des données pour entraîner le réseau. Cette sélection est uniforme : toutes les données ont la même probabilité d'être sélectionnées. Or, certaines données sont mieux prédites que d'autres. Celles qui sont mal prédites mériteraient que l'entraînement les favorisent pour améliorer leur prédiction. C'est toute l'idée du *Prioritized Replay Buffer* : ne pas sélectionner les données uniformément mais en fonction de l'erreur de prédiction commise par le réseau. Nous laissons le lecteur écrire un algorithme correspondant à cette idée.

5.5.7 Illustration de DQN

Pour illustrer DQN, on en applique la version la plus simple à la voiture-sur-la-colline. La figure 15 illustre quelques points :

— à gauche de la figure, une courbe d'apprentissage typique en utilisant un réseau de neurones avec une seule couche cachée de 128 neurones ;

— au milieu de la figure, les 3 couleurs indiquent plusieurs courbes d'apprentissage, en rouge la même qu'à gauche avec 128 neurones, en bleu avec 64 neurones, en vert avec 32 neurones. On voit qu'ici c'est la version avec 32 neurones cachés qui apprend le plus vite et que dans les 3 cas, la performance atteinte est à peu près la même ;

— à droite on retrouve ces trois courbes superposées à la courbe d'apprentissage du Q-Learning tabulaire de la figure 9.

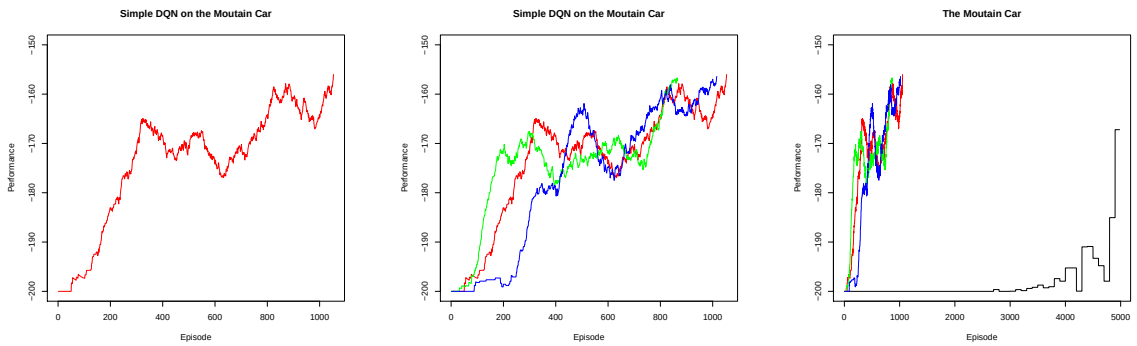


FIGURE 15 – Courbes d'apprentissage de DQN utilisant 3 réseaux de neurones de tailles différentes sur la voiture-sur-la-colline. Voir le texte pour la description de cette figure.

5.5.8 Application de DQN

On introduit un nouveau problème continu, également très utilisé pour tester et comparer les algorithmes. Le *cartpole* est constitué d'un chariot qui peut translater sur des rails horizontaux sur lequel un piquet est posé, fixé à un axe qui lui permet de tourner librement (*cf.* Fig. 16). Quand le chariot subit une accélération d'un côté, le piquet bouge par réaction et se met à osciller. Les actions sont ± 1, qui correspondent respectivement à une accélération d'une certaine amplitude fixée du chariot vers la gauche ou vers la droite. Initialement, le piquet est pointé vers le haut. L'objectif est de le maintenir dans cet l'équilibre instable, pointé vers le haut. À chaque pas de temps où le piquet reste pointé vers le haut, l'agent perçoit un retour de $+1$. L'épisode se termine lorsque le piquet s'est écarté de plus de 15° de sa position d'équilibre ou si le chariot s'est déplacé de plus de 2,4 unités de sa position initiale. La tâche est considérée comme résolue quand l'agent obtient un score supérieur ou égal à 195 en moyenne sur 100 épisodes consécutifs.

La dynamique de ce système est relativement compliquée, voir [12].

Notons que c'est là l'une des définitions du problème du *cartpole*. D'autres définitions existent, notamment dans lesquelles le piquet est initialement pendant vers le bas et qu'il faut apprendre à le maintenir à l'équilibre vers le haut. On recommande la visualisation des vidéos des travaux de M. Riedmiller sur ce point qui contrôle un véritable *cartpole* physique. Son algorithme utilise un réseau avec deux couches cachées de 5 neurones chacune. Partant de l'équilibre stable, piquet vers le bas, la vidéo montre comment son algorithme apprend tout d'abord à faire pointer son piquet vers le haut, puis maintenir l'équilibre instable et est alors capable de résister à de petites perturbations et, s'il perd l'équilibre, le rétablir immédiatement.

En guise d'exercice, on résoudra le cartpole et la voiture-sur-la-colline avec DQN.

6 Autres algorithmes importants

Toutes les méthodes vues jusqu'à maintenant sont basées sur la différence temporelle. Dans cette section, nous introduisons un nouveau type de méthodes très différentes pour résoudre un problème d'apprentissage par renforcement, qui apprennent directement une politique. Ce chapitre présente des algorithmes importants d'apprentissage par renforcement. Au fil de ce chapitre, nous abordons des sujets de plus en plus récents. Pour les sections 6.4 et 6.5, nous sommes au cœur d'importants sujets de recherche en cours d'étude.

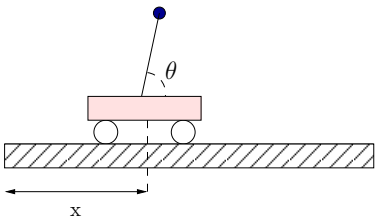


FIGURE 16 – Illustration du problème du *cartpole*.

6.1 Apprentissage direct de politique

Jusqu'à présent, nous avons toujours utilisé une fonction valeur pour en déduire une politique puisque pour les PDM que nous considérons, nous savons qu'une politique gloutonne par rapport à la fonction valeur est optimale. On peut aussi ne pas utiliser de fonction valeur et apprendre directement une politique. Ce sont les méthodes dites d'apprentissage direct de politique.

Voyons d'abord quels sont les problèmes que la première approche (dite « approche de Bellman » ou par programmation dynamique ou basée sur la valeur) posent. Tout irait bien si l'on calculait effectivement exactement la fonction valeur. Or, à tout instant, on n'en a qu'une approximation $\widehat{V}$ car la convergence vers la fonction valeur est asymptotique. En général, on ne sait pas quand il faut arrêter les itérations : il faut arrêter les itérations quand la politique gloutonne par rapport à $\widehat{V}$ est optimale mais c'est précisément ce que l'on ne sait pas déterminer ou en tout cas, c'est difficile à faire. Donc, en général la politique gloutonne par rapport à $\widehat{V}$ n'est pas optimale. Cela entraîne le risque de prendre de mauvaises décisions. Il est courant que la fonction valeur d'un PDM soit complexe. La figure 17 donne un exemple typique. Mesurer et minimiser l'erreur d'approximation de la fonction valeur par une métrique classique (MSE) est utile pour obtenir une forme générale de la fonction valeur, mais insuffisant pour en obtenir tous les détails ; hors, les détails sont cruciaux.

Il s'avère que la politique peut être très simple pour une fonction valeur complexe comme celle de la figure 17. En outre, la solution à notre problème est bien une politique, pas la fonction valeur.

Pour toutes ces raisons, des méthodes qui optimisent directement la politique d'un PDM méritent d'être étudiées. Plusieurs approches sont possibles. L'approche classique en apprentissage par renforcement va être décrite ci-dessous. Auparavant, mentionnons que des algorithmes à base de population (évolutionnaires, stratégies d'évolution) ont été utilisés avec un certain succès [22, 14, 15, 31, 38].

Nous présentons ces méthodes telles qu'elles sont aujourd'hui implantées avec un réseau de neurones. Ces méthodes peuvent parfaitement s'utiliser sans réseau de neurones. Les principes donnés ci-dessous s'adaptent : il faut garder à l'esprit que dans ce cas, les paramètres de la politique notés θ sont les poids d'un réseau de neurones.

6.1.1 Idée générale de l'apprentissage direct de politique

L'idée générale est de travailler avec des politiques stochastiques $\pi^\theta(s, a)$ paramétrées par un certain ensemble de paramètres θ. Actuellement, une politique est représentée par un réseau de neurones dont les poids θ constituent les paramètres de la politique. Le réseau de neurones possède P sorties. Quand on place un état s en entrée du réseau de neurones, la sortie i fournit la probabilité d'effectuer l'action a_i $\pi^\theta(s, a_i)$ ou une information relative à cette quantité (donc toujours une valeur par action).

On cherche donc les paramètres θ de la politique qui optimise la fonction objectif R, donc une politique dont la valeur est maximale. On cherche donc θ^* qui correspond à une politique optimale π^*. Il nous faut donc exprimer un lien entre la valeur d'une politique V^π et les paramètres de

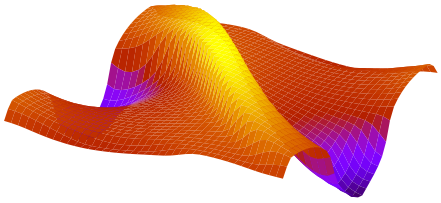


FIGURE 17 – La fonction valeur d'un certain PDM. On note une forme complexe faite de creux et de bosses. Pour avoir un comportement optimal, l'agent doit typiquement rejoindre le sommet à partir de l'un des creux. Il doit notamment passer sur une arête : une petite erreur d'approximation de la fonction valeur peut entraîner un mauvais choix d'action, qui entraîne une chute de l'agent du haut d'une falaise : c'est un peu comme marcher sur une arête de montagne dans un brouillard très épais.

cette politique pour pouvoir ensuite rechercher les paramètres de cette politique qui optimisent la valeur.

On peut montrer que maximiser R revient à maximiser la fonction suivante :

$$\mathcal{L}(\theta) \equiv -\sum_{e,t} Q(s_{e,t}, a_{e,t}) \log\left(\pi^{\theta}(s_{e,t}, a_{e,t})\right) \tag{10}$$

où e est un numéro d'épisode et t le numéro du pas dans un épisode (on aurait pu le noter t_e ce qui aurait plus précis et rigoureux, mais aussi plus lourd, d'où le simple t).

On peut écrire cette fonction de manière plus compacte :

$$\mathcal{L}(\theta) \equiv -Q \log\left(\pi^{\theta}\right).$$

Le lien entre la fonction objectif R et cette fonction $\mathcal{L}$ ne saute pas aux yeux, mais il se démontre mathématiquement. On peut néanmoins se dire que maximiser la fonction objectif a bien un lien avec la maximisation de ce produit : Q représente la valeur d'une action dans un état donné (comme on cherche une politique, on cherche bien la meilleure action dans chaque état), donc celle qui maximise $Q(.,.)$. π étant une probabilité, son logarithme est négatif, d'où ce signe $-$ qui donne une valeur positive à ce produit, que l'on cherche donc à maximiser.

6.1.2 L'algorithme REINFORCE

Nous présentons ici l'algorithme REINFORCE qui est la méthode historique, et toujours d'actualité, d'apprentissage direct de politique en apprentissage par renforcement.

L'optimisation de $\mathcal{L}$ peut être réalisée à l'aide de différentes méthodes. La méthode habituelle actuellement en apprentissage par renforcement est la descente de gradient stochastique (*cf.*

appendice A.1 pour un rafraîchissement concernant cette méthode d'optimisation). Le principe général est de mettre à jour itérativement les paramètres $\theta_{k+1} \leftarrow \theta_k + \alpha \nabla_\theta \mathcal{L}$. α est un coefficient positif à ajuster, dénommé coefficient ou taux d'apprentisage. $\nabla_\theta \mathcal{L}$ est le gradient de $\mathcal{L}$ par rapport à chacun des paramètres du vecteur θ : c'est un vecteur de même dimension que θ. Q est un scalaire réel ; donc, le membre droit est le produit d'un réel par un vecteur, et le résultat donne un vecteur de même dimension que θ, ce qui est bien cohérent avec $\nabla_\theta \mathcal{L}$ qui est le gradient de $\mathcal{L}$ par rapport au vecteur θ.

Il faut donc calculer, du moins estimer $\nabla_\theta \mathcal{L} = \nabla_\theta(-Q \log(\pi^\theta))$. Cette estimation se réalise à partir de données collectées le long d'une trajectoire de l'agent effectuant la politique π^θ. On montre que l'on peut approximer ce terme par $\widehat{Q} \nabla_\theta \log(\pi^\theta)$ où $\widehat{Q} \approx \sum_{j \geq t} \gamma^{j-t} r_j$ est la somme des retours collectés à partir de l'instant t : la notation est cohérente : cette quantité représente bien une estimation de la qualité de la paire (état, action) rencontrée à l'instant t.

Le terme, un peu effrayant, $\nabla_\theta \log(\pi^\theta)$ est calculé soit analytiquement, soit numériquement : tout dépend de la forme de π^θ.

Pour un réseau de neurones, le gradient est calculé automatiquement par les bibliothèques de réseaux de neurones modernes (pytorch, tensorflow). Il suffit de spécifier la fonction que le réseau doit optimiser $\mathcal{L}$ et son gradient est calculé automatiquement [15].

Avec tous ces éléments, on peut maintenant exprimer REINFORCE avec l'algorithme 10. Même si la théorie sous-jacente est assez compliquée, cet algorithme est très simple. Les itérations se poursuivent soit un certain nombre de fois fixé *a priori*, soit jusqu'à ce qu'un certain critère soit rencontré (par exemple que la politique ne change plus beaucoup entre deux itérations successives).

Algorithme 10 Squelette de l'algorithme d'apprentissage direct de politique REINFORCE. Pour simplifier son écriture, on note $\widehat{Q}_{k,t}$ l'estimation de $\widehat{Q}(s_{k,t}, a_{k,t})$ définie dans le texte.

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

```
initialiser les poids du réseau de neurones θ
répéter
  effectuer E épisodes en collectant les transitions dans D.
  pour chaque épisode e faire
    pour chaque pas t de l'épisode e (en utilisant les données stockées dans D) faire
      Q̂_{e,t} ← Σ_{j=t}^{j=|τ_k|-1} γ^{j-t} r_{e,j}
      mettre à jour les poids du réseau θ pour maximiser L
    fin pour
  fin pour
jusque critère d'arrêt vérifié
```

En guise d'exercice, on appliquera REINFORCE au *cartpole*.

15. Ce point rend ces bibliothèques de fonctions très faciles à utiliser ; avant, on devait programmer cela à la main, source d'innombrables bugs... difficiles à détecter... Ces techniques se nomment « différenciation automatique ».

6.1.3 Conclusion sur REINFORCE

Comme il a été exprimé, REINFORCE souffre d'un certain nombre de problèmes. Il nécessite des épisodes complets ; les corrections appliquées aux paramètres peuvent être d'amplitude très variable, éventuellement très grande, ce qui rend l'algorithme instable ; il n'y a pas explicitement d'exploration. Nous traitons chacun de ces points ci-dessous.

Concernant les épisodes complets, rappelons-nous qu'ils sont utilisés pour estimer $\widehat{Q}(s,a)$ la qualité des paires (état, action) (s_t, a_t) par $\sum_{j=t}^{j=|\tau_k|-1} \gamma^{j-t} r_{k,j}$. On constate que cette somme peut être tronquée dès lors que $\gamma^{j-t} r_{k,j}$ est suffisamment petit, ce qui dépend de γ.

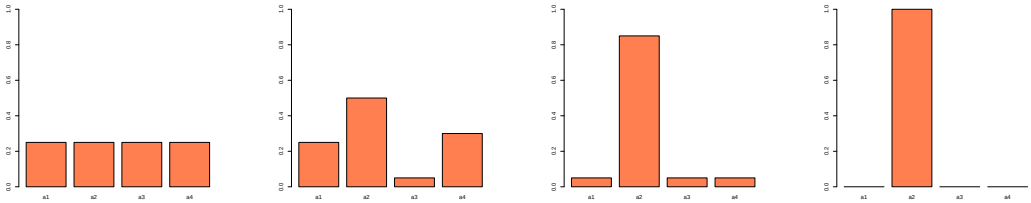
Concernant l'amplitude très variable des corrections, reprenons l'équation de mise à jour des paramètres de l'algorithme REINFORCE : $\widehat{Q}(s_{k,t}, a_{k,t}) \nabla_\theta \log\left(\pi^\theta(s_{k,t}, a_{k,t})\right)$. Par exemple dans le *cartpole*, en début d'apprentissage, l'agent peut maintenir l'équilibre pendant quelques pas de temps : $\widehat{Q}$ sera petit, quelques unités ; plus tard, il va le maintenir plus de 100 pas de temps successifs à l'équilibre : $\widehat{Q}$ va prendre une valeur supérieure de plusieurs ordres de grandeur ; le gradient va ainsi être multiplié par un facteur parfois petit, parfois grand : ce n'est pas une bonne chose pour un apprentissage serein. Par ailleurs, sur un autre plan, il faut bien toujours garder à l'esprit qu'en général, la fonction retour $\mathcal{R}$ est définie arbitrairement : on la spécifie dans le modèle, elle n'est pas imposée par le problème (*cf.* section 7.6) ; pour poursuivre sur le *cartpole*, au lieu de récompenser par +1 le maintien à l'équilibre, pourquoi pas 0,1 ou 10 ? Aussi, $\widehat{Q}$ peut varier considérablement selon le choix qui a été fait de la fonction de retour $\mathcal{R}$; même son signe peut varier. Ces variations ne sont pas réellement liées au fait que la politique est bonne ou pas, mais à des choix arbitraires quand à la définition de $\mathcal{R}$. La solution traditionnelle à ce problème consiste à retirer une certaine quantité, dénommée *ligne de base*, à $\widehat{Q}$ dans l'équation de mise à jour, ce qui donne : $(\widehat{Q}(s_{k,t}, a_{k,t}) - b(s_{k,t})) \nabla_\theta \log\left(\pi^\theta(s_{k,t}, a_{k,t})\right)$. Cette ligne de base b peut être n'importe quelle fonction du moment qu'elle ne dépend pas de l'action. L'utilisation de la valeur de l'état est très répandue, même si ce choix a été remis en cause récemment. On tombe sur l'idée de l'algorithme acteur-critique qui est présenté dans la section suivante.

Enfin concernant l'exploration, la politique π étant stochastique, la capacité d'exploration est liée directement au fait que pour un état donné s, la probabilité de choisir chacune des actions est significative (pas trop petite). L'*entropie* d'une politique est définie par $H(\pi) = -\sum_{(s,a)} \pi(s,a) \ln\left(\pi(s,a)\right)$ (en posant $0 \ln 0 = 0$). C'est une quantité toujours positive, maximale quand π est uniforme (c'est-à-dire, $\pi(s,a) = \frac{1}{P} \forall (s,a)$), minimale lorsque certaines actions sont certaines dans certains états (*cf.* une illustration à la fig. 18). On combine alors la maximisation de cette entropie avec l'optimisation de la fonction objectif pour à la fois trouver une bonne politique et pour que cette politique explore son environnement. La fonction à optimiser devient donc :

$$\mathcal{L}(\pi) = R(\pi) + \lambda H(\pi) \tag{11}$$

où λ est un coefficient qui doit être ajusté [16]. Pour mes lecteurs qui connaissent cette notion,

16. « à la main ».



$H = 1{,}39$ $H = 1{,}20$ $H = 0{,}59$ $H = 0$

FIGURE 18 – Illustration de la notion d'entropie d'une distribution de probabilités. De gauche à droite, on passe d'une distribution uniforme à une distribution piquée sur une seule valeur. L'entropie décroît de gauche à droite, maximale pour une distribution uniforme, nulle pour la distribution piquée.

l'entropie agît comme un terme de régularisation ; pour ceux qui ne connaissent pas cette notion essentielle en optimisation, voir l'annexe B.

Dans cette équation, j'écris que $\mathcal{L}$, R et H sont des fonctions de la politique π. Il faut bien avoir à l'esprit que dans un programme d'ordinateur, la politique est définie, et implantée, par la fonction qui la représente, qui est paramétrée par un vecteur de paramètres θ. Aussi, il est courant d'écrire $\mathcal{L}(\theta) = R(\theta) + \lambda H(\theta)$: dans l'esprit du rédacteur, c'est exactement la même chose (du moment que l'on sait comment la politique est représentée).

Les méthodes d'apprentissage direct de politique ne sont pas utilisées telles quelles aujourd'hui, ou très rarement. Mentionné plus haut, l'algorithme acteur-critique leur est préféré. Il est décrit dans la section suivante. Attention, dans les articles de recherche, on parle souvent d'algorithme d'apprentissage direct de la politique alors qu'il s'agît en réalité d'un algorithme acteur-critique.

6.2 Acteur-critique

On l'a écrit un peu plus haut : pour mettre à jour les paramètres dans une méthode de gradient pour l'apprentissage direct de la politique, on utilise la correction $(\widehat{Q^\theta}(s,a) - b(s))\nabla_\theta \log(\pi^\theta(s,a))$ dans laquelle un choix judicieux pour $b(s)$ est la valeur de s, $V(s)$. Cette valeur est inconnue, il faut donc l'apprendre également. Et il faut aussi une représentation de cette fonction, donc l'utilisation d'un approximateur de fonctions ; celui-ci est paramétré par un certain ensemble de paramètre que nous noterons θ_V [17], l'indice V indiquant qu'il s'agît des paramètres de la fonction valeur. Cela conduit donc naturellement à l'idée d'un algorithme qui apprend à la fois une politique et la fonction valeur : un algorithme acteur-critique. Cette dénomination vient du fait que cet algorithme est composé de deux éléments, un acteur qui apprend une politique et agit et un élément qui critique ce que fait l'acteur en fonction de sa connaissance (son estimation) de la fonction valeur.

17. on pourrait noter θ_π les paramètres de la politique à la place de θ, mais pour simplifier l'écriture on ne précisera pas l'indice π, à moins que cela ne soit nécessaire pour lever une ambiguité.

Notons que la fonction $Q(s,a) - V(s)$ est nommée la fonction avantage $A(s,a) : V(s)$ étant la moyenne des $Q(s,a)$ pour un état s donné, l'avantage indique si une action est meilleure ou moins bonne que la moyenne dans cet état. L'algorithme décrit ici s'appelle aussi A2C pour *Advantage Actor-Crtic*. Il optimise la fonction $\mathcal{L}_{AC} \equiv A \log \pi^\theta$.

Le principe d'un algorithme acteur-critique est très simple : on prend l'algorithme REINFORCE et on y ajoute l'apprentissage de la fonction valeur par minimisation de l'erreur quadratique moyenne. Nous indiquons cela dans l'algorithme 11.

Un algorithme acteur-critique nécessite donc un ensemble de paramètres pour représenter la politique et un ensemble de paramètres pour représenter la valeur. Actuellement, politique et valeur sont représentées par des réseaux de neurones. Au lieu d'avoir deux réseaux séparés, on peut n'utiliser qu'un seul réseau de neurones qui possède deux « têtes », une tête fournissant π, l'autre une approximation de V. Chaque tête est constituée d'une ou quelques couches de neurones, s'appuyant sur un plus gros réseau. Ce « gros » réseau calcule une représentation interne de l'entrée, c'est-à-dire de l'état ; cette représentation interne est alors utilisée par chacune des têtes. Ainsi, les poids du gros réseau sont partagés par les deux têtes et seuls les poids de chacune des têtes sont spécifiques à π ou à V (*cf.* fig. 19 pour une illustration du principe). Outre le fait que ce partage de poids permet de calculer une représentation qui soit pertinente à la fois pour approximer la valeur et la politique, il y a donc moins de poids à entraîner, donc l'entraînement est plus rapide et, on l'espère, de meilleure qualité. Si cette approche à deux têtes a été très en vogue, l'utilisation de deux réseaux distincts est très aujourd'hui fréquente.

Algorithme 11 Squelette d'un algorithme acteur-critique.

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

initialiser les paramètres θ_V et θ de V et π

répéter

effectuer K épisodes en collectant les transitions dans T.

mettre à jour les θ_V par minimisation MSE de la différence temporelle mesurée sur ces transitions

pour chaque épisode k **faire**

pour chaque pas t de l'épisode k **faire**

$\widehat{Q}_{k,t} \leftarrow \sum_{j=t}^{j=|\tau_k|-1} \gamma^{j-t} r_{k,j}$

$\theta \leftarrow \theta + \alpha\gamma^t(\widehat{Q}_{k,t} - \widehat{V}_{k,t})\nabla_\theta \log\left(\pi^\theta(s_{k,t}, a_{k,t})\right)$

fin pour

fin pour

jusque critère d'arrêt vérifié

6.3 Pour terminer, avant d'aller plus loin

Ces dernières années, de multiples variantes de l'algorithme acteur-critique ont été proposées. Beaucoup de ces variantes consistent à modifier la fonction $\mathcal{L}$ qui est optimisée. On va en voir des exemples dans les sections qui suivent.

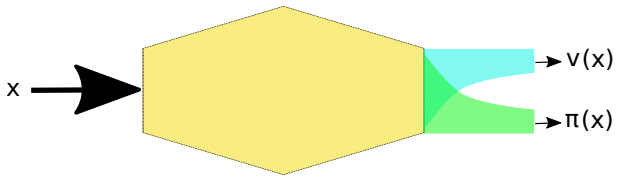


FIGURE 19 – Illustration du principe d'un réseau de neurones à deux têtes classiquement utilisé dans les algorithmes acteur-critique. En entrée du réseau (à gauche) sont placés les propriétés d'un état. Le « gros » réseau (en jaune) calcule une représentation interne. À droite, deux têtes sont connectées qui prennent cette représentation interne en entrée pour fournir une estimation de $V(s)$ (en bleu), l'autre (en vert) une distribution de probabilités sur $\mathcal{A}$: $\pi(s)$. Contrairement à ce que le schéma pourrait laisser penser, les parties vertes et bleues sont totalement disjointes.

Un autre principe consiste à ajouter des têtes au réseau de neurones, têtes qui sont entraînées pour prédire des quantités dites auxiliaires qui aident le réseau à mieux apprendre, c'est-à-dire à atteindre un meilleur optimum. Par exemple, on peut apprendre à prédire l'état qui va être atteint au pas suivant, ou le retour immédiat qui va être observé : apprendre à prédire ces autres quantités constitue la résolution de tâches dites auxiliaires. Le concept caché derrière ce type d'approches, et de bien d'autres, et d'ajouter de la structure dans le problème, ou plutôt, d'identifier et exploiter la structure qui existe dans le problème. Il y a beaucoup de structure non exploitée dans le PDM : l'espace d'états n'est pas un simple sac d'états : certains états sont plus similaires que d'autres ; pour certains états, se tromper un peu et atteindre un état proche ne change pas grand chose ; pour d'autres, ça peut changer beaucoup de choses (rappelez-vous la fonction valeur de la voiture-dans-la-colline). De même, aucune structure n'est exploitée pour les actions : les actions sont simplement numérotées et le numéro associé à une action particulière est tout à fait arbitraire ; par ailleurs, dans cette numérotation, il n'y a aucune information quant à l'impact d'une action sur l'état courant. Or, il peut exister des relations de symétries entre des actions (par exemple, dans le problème du labyrinthe, aller en avant et l'opposé d'aller en arrière : cela semble évident mais cette information n'est pas exploitée ce qui a pour conséquence qu'en début d'entraînement, le Q-Learning présenté plus haut réalise des tas de séquences d'actions qui s'annulent les unes les autres, réalisant ainsi des tas de pas pour rien). Ceci n'illustre que quelques exemples de ce que nous entendons par le mot structure. Le point clé est que l'utilisation de cette structure peut considérablement aider l'algorithme à apprendre. En guise d'illustration, le lecteur est invité à reprendre le problème du labyrinthe et à transformer sa résolution en empêchant les actions annulant la dernière action réalisée, tant que l'agent n'est pas bloqué (parfois, il faut quand même revenir en arrière, si on est bloqué). Évidemment, l'apprentissage va être beaucoup plus rapide.

Tenir compte de la structure correspond à ajouter des contraintes lors de l'optimisation. La structure conditionne la trajectoire effectuée par l'optimiseur. De manière imagée, sans utiliser cette structure, l'optimiseur a de multiples manières de réaliser son pas suivant ; l'utilisation de la structure du problème permet de réaliser un choix plus adéquat. C'est donc une forme de

régularisation (*cf.* annexe B si ce terme vous est inconnu).

6.4 A continu

Jusqu'à maintenant, nous avons considéré uniquement le cas où les actions sont en nombre fini. On ne l'a pas dit, mais le nombre d'actions doit être petit. On va s'intéresser ici au cas où $\mathcal{A}$ est infini et, plus précisèment, un sous-ensemble compact (un intervalle de valeurs) sur $\mathbb{R}$, voire $\mathbb{R}^B$. Cette situation arrive naturellement quand on veut contrôler un système physique : l'action est une accélération qui peut prendre une valeur dans un intervalle fixé. Ainsi, précédemment dans le problème du *cartpole*, on a considéré qu'il y avait 3 actions possibles : ne rien faire et ± 1 ; on pourrait aussi considérer que l'action prend sa valeur dans l'intervalle $[-1, 1]$. Si l'on considère un robot muni de B articulations contrôlables, alors une action revient à appliquer une certaine accélération à chaque articulation, donc un ensemble de B nombres réels, chacun appartenant à son intervalle de valeurs possibles, par exemple $[-1, 1]^B$.

On suppose que partant d'un état, deux actions d'amplitudes proches (a et $a + \epsilon$) amènent le système dans des états proches à l'instant suivant (notion de continuité de la dynamique).

Les algorithmes de type *Q-Learning* qui stockent une valeur pour chaque action possible sont inadaptés à cette situation où le nombre d'actions est infini.

La principale question est alors de représenter une politique. Précédemment, pour une action pouvant prendre un certain nombre P de valeurs différentes, on l'a représentée par une distribution de probabilités sur ces P valeurs. Ici, P est infini. Une hypothèse courante est de considérer que la probabilité d'émettre une action avec une certaine amplitude est décrite par une certaine distribution de probabilités dont il faut apprendre les paramètres. Dans le cas le plus simple, on considère que cette distribution est normale et qu'il faut donc en apprendre la moyenne et l'écart-type. Si l'action possède B dimensions, on apprend donc B distributions de probabilités.

On décrit maintenant très brièvement deux algorithmes assez récents qui constituent l'état de l'art en méthodes *on-policy* et *off-policy* dans le cas d'espaces d'actions continus. Au moment où j'écris ces lignes (2020-21), ce sujet est très étudié et très mouvant.

6.4.1 PPO

Nous décrivons briévement ici une variante de l'acteur-critique qui constitue l'un des algorithmes *on-policy* les plus performants actuellement, l'algorithme PPO pour *Proximal Policy Opimization*.

Proposé en 2017, PPO est un descendant d'un autre algorithme dénommé TRPO proposé en 2015. Ce sont tous deux des algorithmes acteur-critique. L'idée centrale de TRPO est d'essayer de garantir qu'à chaque mise à jour de la politique, celle-ci s'améliore. Cette attente semble évidente, mais A2C ne le garantit pas. En effet, la politique étant apprise par descente de gradient (voir appendice A.1), si on n'y prête pas attention, la politique peut empirer lors d'une mise à jour de ses paramètres ; garantir son amélioration à chaque itération requiert une attention particulière. Pour cela, l'algorithme TRPO ajoute un mécanisme empêchant les modifications trop importantes de la politique lors de sa mise à jour. TRPO est assez complexe et assez lent,

à la fois chaque itération est assez lourde en temps de calcul et TRPO est assez peu efficace (le nombre de transitions pour obtenir une politique d'une certaine qualité est assez élevé).

Pour sa part, PPO est un algorithme beaucoup plus simple qui essaie lui aussi de faire en sorte que la nouvelle politique ne soit pas trop différente de la précédente et qu'elle l'améliore. En pratique, PPO est plus performant que TRPO. Pour cela, PPO optimise la fonction suivante : $\mathcal{L}_{PPO} \equiv \text{clip}(\frac{\pi_{new}}{\pi_{old}}, A, \delta)$, à comparer à $\mathcal{L}_{A2C} \equiv A \log \pi$.

On définit :

$$\text{clip}(\alpha, A, \delta) \equiv \begin{cases} A \min(\alpha, (1+\delta)) \text{ si } A \geq 0, \\ A \min(\alpha, (1-\delta)) \text{ sinon.} \end{cases}$$

Cette fonction empêche les variations de la politique qui entraînent une trop forte variation de la fonction avantage. Ça semble compliqué, mais le principe est très simple.

Détaillons $\mathcal{L}_{PPO}$: $\mathcal{L}_{PPO} \equiv \sum_{(s_t,a_t)} \text{clip}(\frac{\pi^{\theta_{k+1}}(s_t,a_t)}{\pi^{\theta_k}(s_t,a_t)}, A(s_t,a_t), \delta)$.

Dans cette équation, π^{θ_k} est la politique suivie actuellement, $\pi^{\theta_{k+1}}$ est la prochaine politique, celle que l'on veut calculer : θ_{k+1} constitue l'inconnue du problème de maximisation de $\mathcal{L}_{PPO}$.

La somme est réalisée sur les échantillons collectés sur une ou des trajectoires avec la politique π^{θ_k}. Considérant un échantillon (s_t, a_t) : soit l'avantage $A(s_t, a_t)$ est positif, soit il est négatif. S'il est positif, cela signifie que a_t est (estimée comme) une bonne action dans l'état s_t. Dans ce cas, on a envie d'augmenter sa probabilité d'émission, donc que $\pi^{\theta_{k+1}}(s_t, a_t) > \pi^{\theta_k}(s_t, a_t)$. S'il est négatif, le raison est inverse : c'est une mauvaise action dont la probabilité d'émission doit diminuer. Si A est positif, $\frac{\pi^{\theta_{k+1}}(s_t,a_t)}{\pi^{\theta_k}(s_t,a_t)}$ doit être supérieur à 1 (pour que $\pi^{\theta_{k+1}}(s_t, a_t) > \pi^{\theta_k}(s_t, a_t)$) ; donc, le terme $\min(\pi^{\theta_{k+1}}(s_t, a_t) > \pi^{\theta_k}(s_t, a_t), (1+\delta))$ vaudra $\pi^{\theta_{k+1}}(s_t, a_t) > \pi^{\theta_k}(s_t, a_t)$ si celui-ci est plus petit que $1+\delta$, ou $1+\delta$: ainsi, on limite la variation de la probablitié d'émission de a_t. Le raisonnement est symétrique si A est négatif. δ est un paramètre très important de PPO.

Remarque : π étant une distribution de probabilités, on pourrait exprimer cette contrainte autrement, en limitant la différence entr π^{θ_k} et $\pi^{\theta_{k+1}}$ en utilisant une mesure appropriée, comme la divergence de Kullback-Leibler ou une distance de Wasserstein. Cela a aussi été proposé dans d'autres algorithmes, mais expérimentalement, c'est PPO qui obtient les meilleures performances. Il s'agit de résultats empiriques dont nous ne comprenons pas la raison.

6.4.2 SAC

PPO est un algorithme *on-policy* qui ne peut s'entraîner que sur des trajectoires collectées avec la politique courante. De son côté, SAC est un algorithme acteur-critique *off-policy*, considérer comme le meilleur d'un point de vue expérimental actuellement. Aussi le décrit-on briévement.

L'idée majeure dans SAC est celle de régularisation de l'entropie de la politique : on cherche une politique qui maximise $\mathcal{L}_{SAC}(\pi) \equiv R(\pi) + \lambda H(\pi)$. On a déjà rencontré cette expression pour REINFORCE (*cf.* section 6.1.2). Mais par rapport à REINFORCE, SAC utilise les meilleures idées introduites depuis : double DQN, acteur-critique, et prédiction par le réseau de neurones outre de π et V, mais aussi de deux Q (à la double DQN).

6.5 *Model-based*

L'apprentissage par renforcement est lent ; il faut essayer de tirer un maximum d'information, et l'exploiter, de chaque interaction entre l'agent et son environnement. Jusqu'à présent, on a toujours considéré que l'on obtient de l'information sur les conséquences de l'action qui est émise et elle seule. On peut envisager doter l'agent d'un mécanisme lui permettant d'anticiper les conséquences de ses actions, comme un humain essaie d'anticiper ce qui va se passer s'il effectue une certaine action ; ainsi, au lieu d'essayer et voir ce qui se passe, on peut choisir la meilleure action ou une action qui semble la meilleure étant donnée les informations dont on dispose. Pour cela, il faut disposer d'un « modèle » de l'environnement qui nous permet de représenter les conséquences attendues des actions.

Plusieurs situations existent. Par exemple, dans le cas de jeux déterministes dont les règles sont bien définies (échecs, go, ...), un simulateur du jeu peut être utilisé pour construire un arbre de jeu pour les prochains coups, à partir de l'état courant. Construire un tel arbre de jeu est une technique très classique en intelligence artificielle. Des techniques « modernes » essaient de construire seulement une partie de l'arbre qui est utile pour la prise de la prochaine décision, ce qui se révèle indispensable lorsque le facteur de branchement est important. Il n'est pas possible de présenter ces techniques ici ; le lecteur intéressé est invité à consulter les travaux sur les recherches arborescentes de Monte Carlo ([7] pour une présentation pas trop ancienne dans ce domaine actuellement en plein développement) qui applique l'idée présentée plus haut de méthodes de Monte Carlo (*cf.* section 4.5).

Pour des jeux non déterministes (*i.e.* avec un lancer de dé), cette idée est encore valable et plus ou moins applicable dans la pratique. En effet, ne connaissant pas le résultat du lancer de dé qui sera observé, on ne peut qu'explorer les différentes alternatives à ce lancer, ce qui entraîne la construction de nombreuses branches qui seront inutiles par la suite, une fois que le dé aura été lancé.

Pour des situations plus complexes dans lesquelles on ne peut pas spécifier précisèment ce qui va se passer, on peut essayer d'utiliser un modèle approché, voire d'apprendre un modèle de la dynamique.

Après la proposition du Q-Learning, il a été proposé un algorithme qui apprend un modèle de l'environnement, soit des fonctions $\mathcal{P}$ et $\mathcal{R}$: l'algorithme Dyna. Celui-ci, au fil des interactions, construit donc une estimation de ces deux fonctions. Cette estimation peut ensuite être utilisée pour améliorer l'estimation de la fonction $\widehat{Q}$ en faisant des « expériences de pensée » : l'agent simule l'exécution d'une action dans un état, observe le retour et l'état suivant et utilise cette information comme une transition observée pour mettre à jour son estimation de $\widehat{Q}$. L'idée est bonne mais telle qu'elle vient d'être décrite, elle reste bien simpliste pour l'application sur des tâches qui ne sont pas déterministes, et de petite taille. Néanmoins, on peut poursuivre la réflexion et obtenir des algorithmes qui peuvent être utilisés.

Il y a beaucoup de travaux dans cette direction, en particulier pour y voir plus clair quant au fait que le modèle étant une approximation de l'environnement, il faut l'utiliser quand il est utile, et ne pas l'utiliser quand il induit l'agent en erreur.

À mon avis, la résolution par apprentissage par renforcement de problèmes de taille significative passe forcément par l'utilisation de ce genre d'idées. On pourra lire [8].

7 Quelques autres sujets actuellement étudiés

Dans cette section, on indique des sujets d'actualité qui nous semblent importants. Nous ne faisons que les décrire brièvement pour vous mettre en appétit. Sur chacun, il y a déjà des travaux de recherche qui ont été menés, mais il reste encore beaucoup à faire : plusieurs thèses au moins sur chaque sujet !

7.1 Représentation

En informatique, on sait que la représentation des informations est un sujet central : une bonne représentation diminue les temps de traitement, par exemple en rendant plus efficace la recherche d'éléments. La conception de structures de données est l'un des points les plus importants dans la réalisation des programmes.

De même, en apprentissage automatique, l'utilisation d'une représentation bien choisie des données peut tout changer dans les performances d'une méthode. Une « bonne » représentation contient ici l'information pertinente, et seulement elle. Qu'elle contienne l'information pertinente permet de trouver la solution ; qu'elle ne contienne que l'information pertinente est un gage d'efficacité dans la recherche de la solution. Ceci dit, comment savoir que la représentation utilisée contient bien toute l'information pertinente ? Il n'y a pas de réponse bien nette à cette question : un expert sait, ou sent, qu'il utilise une telle bonne représentation, confirme son intuition en analysant le problème qu'il cherche à résoudre, et constate que sa représentation est bonne ou suffisamment bonne, car l'algorithme donne le résultat attendu.

En apprentissage par renforcement, la donnée est un état ou une paire (état, action). Lorsqu'un réseau de neurones est utilisé pour représenter la fonction valeur ou la politique, voire d'autres fonctions, celui-ci exploite implicitement des propriétés mathématiques, comme la continuité de la fonction qui est représentée. Si la valeur de la donnée en entrée du réseau varie un peu, la sortie varie un peu. Aussi, si la donnée en entrée est un état, si l'état change un peu, la sortie, la valeur de l'état change un peu. Cela a deux conséquences : il n'est pas facile de représenter des variations très brutales de la fonction représentée ; il faut que la manière dont sont codés les entrées tiennent compte de cette remarque. Ainsi, dans un labyrinthe bidimensionnel, si l'état est le numéro d'une case, cette numérotation ne peut pas tenir compte des notions de proximité des cases en deux dimensions ; donc, avec cette représentation, la fonction valeur n'est pas « continue » [18] ; donc la représentation de la fonction valeur par un réseau est difficile. Ça ira

18. de toute façon, si on code la position de l'agent avec le numéro de la case du labyrinthe où il se trouve, le domaine de définition est discret et donc la fonction ne peut pas être continue (une fonction doit être réelle pour pouvoir être continue). Néanmoins, on voit bien que deux états numérotés i et $i+1$ peuvent être voisins pour certains i, et pas pour d'autre (passage à la ligne). Et en fait, quand on utilise une simple numérotation des cases d'un labyrinthe, on peut les numéroter dans n'importe quel ordre car un numéro n'est pas un vrai nombre : il n'y a pas d'ordre, on pourrait utiliser des symboles à la place des numéros, ça ne changerait rien pour l'algorithme

beaucoup mieux si un état est représenté par ses coordonnées cartésiennes : (i, j) et $(i+1, j)$ sont voisins, donc leurs valeurs sont proches [19] ... sauf si l'un des deux est un mur, qui va créer une discontinuité à nouveau. En fait, un « simple » labyrinthe pose des problèmes de représentation complexes. Considérons le *cartpole* : c'est un système dynamique physique et on sait que l'état d'un tel système, considéré ponctuel, est exactement représenté à chaque instant par sa position et sa vitesse. Donc, quand on veut contrôler ce type de systèmes, utiliser la position et la vitesse instantanées est une bonne représentation de l'état. Si on considère un jeu comme les échecs, à nouveau, une bonne représentation du jeu qui préserve la continuité de la fonction valeur n'est pas simple. On peut aussi considérer des tâches où l'état est défini par une image (un jeu vidéo par exemple) ; pour ce type d'état d'entrée, on connait des techniques très performantes qui en construisent une bonne représentation interne (réseaux de neurones convolutionnels). Si on considère une tâche définie sur un graphe, à nouveau, on ne sait pas construire une « bonne » représentation pour un graphe.

On l'a vu, dans un algorithme acteur-critique utilisant un réseau de neurones, les premières couches de celui-ci calculent une représentation de l'entrée du réseau (s) et cette représentation est ensuite utilisée pour calculer les sorties ($\widehat{V}(s)$, $\pi(s)$, ...). Du fait du fonctionnement de l'algorithme d'entraînement des poids du réseau, une représentation combinant l'entrée et la fonction objectif à optimiser est produite lors de l'entraînement.

On peut aussi réfléchir à la représentation des actions. Si on reprend le cas du labyrinthe, il est de coutume d'avoir 4 ou 5 actions qui permettent de se déplacer d'une case dans chacune des 4 directions cardinales et une 5^{e} qui consiste à ne pas bouger. À nouveau, ce sont juste des numéros ou des symboles, pas des vrais nombres. Il serait bien plus pertinent que les actions soient représentées d'une manière qui contienne de l'information sur ce qu'elles font vraiment, c'est-à-dire comment elles transforment l'état courant en l'état suivant. Par exemple, si on utilise une représentation de la position de l'agent par ses coordonnées cartésiennes dans le labyrinthe, l'action « aller à droite » dans un labyrinthe a pour effet d'ajouter 1 à son abscisse ou la laisser inchanger, selon que l'agent peut se déplacer à droite ou pas (absence ou présence d'un mur). Si l'action « aller à droite » est simplement codée par disons 2 (ou un caractère $\rightarrow$), quelle que soit l'effet de l'action sur l'état, la représentation est la même. Encore peu nombreux, des travaux commencent à s'intéresser à cette question : avoir une représentation pertinente des actions rendrait fort probablement les algorithmes plus efficace ; tout comme un état doit permettre de déterminer l'action optimale à exécuter dans cet état, une action devrait informer sur ses effets, en termes de changement d'état et de retour perçu.

7.2 Apprentissage par transfert

L'apprentissage par renforcement étant lent, il faut tirer partie de la moindre information et tirer partie aussi de tout ce qui a été appris à un moment donné. Il est courant que l'on

(d'ailleurs, l'algorithme ne manipule que des symboles sans relation entre-eux, pas des nombres).

19. suite de la note précédente : dans ce cas i et j sont maintenant bien des nombres : il y a un ordre : plus la coordonnée est petite, plus la case est à gauche ; la différence des coordonnées selon un certain axe indique une certaine distance.

souhaite résoudre non pas une tâche, mais une famille de tâches. Par « famille de tâches » on veut dire un ensemble de tâches qui se ressemblent beaucoup mais qui ont des caractéristiques légérement différentes. Par exemple, imaginons que nous ayons entraîné un agent pour résoudre la voiture-dans-la-colline et que la pente de la côte varie légérement, l'agent doit pouvoir adapter sa politique sans devoir la ré-apprendre depuis le début. Dans cette situation, on veut essayer de tirer partie de ce qui a été appris sur une tâche et transférer cela vers la nouvelle tâche. C'est l'idée de l'apprentissage par transfert [20]

Le cas le plus simple est celui où le nouveau PDM à résoudre est décrit par les mêmes ensemble d'états et d'actions, la même fonction de retour $\mathcal{R}$ et la même fonction objectif à optimiser R, et que seule $\mathcal{P}$ change un peu.

Ce problème du transfert d'apprentissage devient essentiel dans des applications réelles de l'apprentissage par renforcement. Par exemple, en robotique, il est courant d'entraîner un algorithme sur des simulations du robot et d'ensuite transférer ce qui ainsi été appris dans le contrôleur du vrai robot. Lors de ce transfert, on passe d'un monde virtuel au monde réel ; dans le monde virtuel, le réel a été encodé, de manière approximative. Ce type de transfert qui semble très simple et très naturel est en fait difficile à réaliser car les petits écarts entre le virtuel et le réel ont des conséquences énormes sur l'algorithme d'apprentissage. Sans prendre de très grandes précautions, cette approche ne marche tout simplement pas.

Le transfert prend de multiples formes dans la littérature. On peut essayer de transférer des comportements d'humains vers l'algorithme : c'est l'apprentissage par démonstration. On distingue le cas où l'humain est expert ou non de la tâche, c'est-à-dire la confiance que l'algorithme peut mettre dans la démonstration.

On peut vouloir qu'un algorithme d'apprentissage par renforcement apprenne à résoudre une tâche qu'il a spontanément une probabiltié très faible de réaliser. Dans ce cas, comme avec un animal auquel on veut apprendre à réaliser une tâche qui ne lui est pas naturelle (un ours qui fait du vélo par exemple), on peut manipuler la fonction de retour au fil de l'apprentissage : au lieu de ne récompenser que le comportement hautement improbable, on commence par récompenser des comportements beaucoup plus simples, beaucoup plus probables, et progressivement, on récompense des comportements de plus en plus spontanément improbables mais qui deviennent beaucoup plus probables du fait de cete procédure. C'est du « domptage », technique connue sous le nom de façonnage en psychologie ou en éthologie [21]. Du fait de sa similarité avec la pédagogie, on parle aussi maintenant de *curriculum learning*. Ce terme est apparu en apprentissage par renforcement assez récemment pour faire comme si c'était une idée nouvelle, alors qu'il s'agît exactement de façonnage du comportement, technique étudiée depuis au moins la décennie 1990.

Une autre manière de faire du transfert, à mon avis obscure, mais « ça marche ! », est d'entraîner l'agent à résoudre plusieurs tâches en même temps. L'exemple le plus connu est l'entraînement de DQN sur 150 jeux vidéos Atari. On ne comprend pas dans le détail ce qui se passe, mais on imagine que DQN apprend des choses communes à différents jeux et qu'il utilise

20. *transfer learning* en anglais.
21. *shaping* en anglais.

des choses apprises sur un, ou des jeux, pour en résoudre d'autres [22].

Un dernier cas de transfert que nous mentionnons ici concerne la représentation tabulaire d'espace d'états continus (*cf.* la section 5.2 sur l'approche tabulaire approchée). Comme on ne connait pas la bonne résolution de grille utiliser et comme plus la grille est fine, plus les temps de calcul sont longs, on peut adopter une approche qui consiste à nouveau à rendre le problème de plus en plus lourd à résoudre. On commence par une grille grossière dont on se doute que la résolution est insuffisante pour représenter avec suffisamment de précision la fonction valeur et progressivement, on raffine cette grille en transférant de la grille grossière vers la grille raffinée. On peut aussi raffiner certaines cellules et pas toutes, selon ce qui observé par l'algorithme. Nous retombons sur la dicrétisation adaptive introduite plus haut.

En guise d'exercice, on encourage le lecteur à concevoir des expériences de transfert d'apprentissage sur des tâches que nous avons déjà rencontrées.

7.3 Passage à l'échelle

Cette section va être très brève mais elle est présente car il s'agît d'un sujet majeur pour le déploiement en conditions réelles de l'apprentissage par renforcement : actuellement, les algorithmes d'apprentissage par renforcement apprennent très lentement. Trouver des moyens de les accélérer et passer à l'échelle pour pouvoir traiter des espaces d'états beaucoup plus grands est un objectif important mais sur lequel butent la communauté scientifique. Les algorithmes d'apprentissage par renforcement demandent beaucoup d'interactions agent-environnement pour apprendre et ils ne se parallélisent pas bien car ce sont des algorithmes séquentiels.

7.4 Apprentissage par renforcement sur des données

On a supposé dans tout ce cours que l'on dispose d'un simulateur de l'environnement. C'est une hypothèse tout à fait pertinente si on veut entraîner un agent à jouer à un jeu comme les échecs ou le tarot, c'est-à-dire des tâches dans lesquelles l'environnement peut être parfaitement et exhaustivement décrit. Dès que l'on quitte ce genre de tâches et qu'on essaie d'aborder de « vrais » problèmes concrets, cette hypothèse n'est plus tenable : impossible de simuler un environnement pertinent pour un véhicule autonome par exemple. Et pas la peine d'aller chercher si loin.

Il y a beaucoup d'applications potentielles de l'apprentissage par renforcement pour lesquelles on peut seulement observer des trajectoires contrôlées par des humains voire même un ensemble d'interactions entre un agent et son environnement, soit des t-uplets (s_t, a_t, r_t, s_{t+1}). Avec ces informations, il faut essayer d'apprendre au mieux une politique. Cela peut passer par l'estimation d'un modèle pour ensuite se replacer dans le cadre de l'apprentissage par renforcement basé sur un modèle, mais ce modèle doit être estimé à partir d'un nombre de données généralement très faible. En outre, ces difficultés se combinent souvent à des problématiques de maîtrise du

22. Exceptionnellement dans ce cours, j'utilise le mot « chose » qui est très imprécis à dessein : effectivement, on ne sait pas bien ce qui se passe dans l'algorithme et on ne sait pas bien ce qu'apprend l'algorithme, ni ce qu'il transfère. On a quand même la nette impression qu'il y a du transfert entre les différentes tâches.

risque qui empêche une exploration à tout-va (*cf.* section 7.7) .

Ce sujet est un autre sujet majeur pour pouvoir faire sortir l'apprentissage par renforcement des laboratoires. On le connait sur le nom de *offline reinforcement learning*.

7.5 S ou A discret très grand

Le cas où l'ensemble des états ou l'ensemble des actions est très grand, voire infini dénombrable, demeure un défi. Pour l'ensemble des états, cela correspond à des problèmes où le nombre d'états est grand (typiquement $> 10^9$ pour donner un ordre de grandeur) et où l'ensemble des états n'est pas muni d'une distance pertinente. Une distance permet de mesurer la différence entre deux états, donc d'ajouter une structure à l'ensemble des états. Quand on n'est pas capable de définir une distance entre états, les états sont identifiés par un numéro et il n'existe aucune relation particulière entre un état i et l'état $i+1$ ou quelqu'autre état. Cela a pour conséquence que la fonction valeur n'a aucune régularité entre états proches (on ne peut pas parler de continuité puisque le domaine est ici discret mais on imagine bien ce que l'on veut dire). Avec un ensemble d'états de ce type, on peut seulement apprendre la valeur de chaque état et utiliser une représentation tabulaire pour la stocker.

Il y a des tas de problèmes intéressants qui souffrent de cette situation. L'optimisation combinatoire en est un exemple. Des tas de problèmes peuvent être dérivés de problèmes d'optimisation combinatoire en y ajoutant de l'incertitude. Des tas de jeux souffrent aussi de ce problème.

7.6 Le retour

Souvent, quand on modélise un vrai problème comme un PDM, la fonction de retour n'est pas dictée par la solution du problème : le modélisateur possède une grande latitude pour la définir. Par exemple pour un jeu comme les échecs, on peut facilement définir la fonction retour pour les combinaisons terminales d'une partie : je gagne, je perds, ou on fait match nul : $+1, -1, 0$. Et pour tous les autres coups de la partie, on ne sait pas, donc on retourne 0. Le retour est censé guider l'algorithme mais avec une telle définition, rien ne guide l'algorithme vers une stratégie qui lui permette de gagner ; l'apprentissage va donc être très difficile et très lent. Tout l'art du modélisateur consiste à définir la fonction de retour pour aider l'agent à apprendre une bonne stratégie. On parle de retours éparses pour des tâches dans lesquels l'agent perçoit un retour non nul de temps en temps, assez rarement, l'obtention d'un tel retour ne donnant par d'information sur l'obtention du retour suivant (imaginez une tâche dans laquelle plusieurs étapes clés doivent être franchies successivement : quand l'une est franchie, que doit-on faire ?).

Des travaux récents distinguent retour explicite, fourni par l'environnement, de retour implicite, voire de motivation de l'agent pour atteindre certains états. D'autres travaux récents essaient de se passer de retour. Il reste beaucoup à faire sur ces questions.

7.7 Maîtrise du risque

Quand l'algorithme explore, il prend des risques. Dans certaines applications, ce n'est pas possible de prendre des risques, en tout cas, pas n'importe quel risque. Par exemple, dans de

nombreuses situations dans lesquelles l'agent apprend un comportement dans un environnement anthropisé, l'exploration est limitée : pas question qu'un robot explore en frappant des humains pour voir ce qui se passe ! Pas question qu'un véhicule autonome écrase des piétons, renversent les cyclistes et foncent dans les autres véhicules pour apprendre qu'il ne faut pas le faire. Et quand des intérêts financiers sont en jeu, il n'est pas non plus question de faire n'importe quoi.

Ces questions sont encore relativement peu explorées en apprentissage par renforcement. Si on veut que l'apprentissage par renforcement sorte des laboratoires, c'est un sujet essentiel. Cela rejoint la notion de contrôle robuste en théorie du contrôle. La question a deux côtés : alors qu'il ne faut pas mettre en danger l'environnement, il faut apprendre et donc essayer des choses : si on est trop prudent, on ne prend aucun risque et on n'apprend rien. Dans les véhicules autonomes par exemple, une tâche très difficile est le franchissement d'un carrefour ou l'insertion dans un rond-point : tous les jeunes conducteurs savent qu'à un moment, pour passer un rond-point sur lequel le trafic est relativement dense, il faut oser s'engager alors que des véhicules approchent constamment !

7.8 Environnements non stationnaires

On l'a vu (*cf.* section 4.2), les algorithmes d'apprentissage par renforcement s'adaptent aux changements de leur environnement : c'est une propriété clé de ces algorithmes, héritée directement du fait qu'ils s'inspirent de l'adaptation du comportement des animaux. Or, il y a très peu de travaux concernant l'apprentissage par renforcement dans un environnement qui change au fil du temps. Il y a des travaux expérimentaux qui montrent que « ça marche », en tout cas dans une certaine mesure, mais il n'y a pas de travaux plus fondamentaux pour comprendre cette situation et guider la conception d'algorithmes et leur utilisation. Or, à nouveau, si l'on veut que l'apprentissage par renforcement sorte des laboratoires, la non stationnarité de l'environnement doit êre prise en compte. Nous évoluons dans un environnement qui change constamment ; nos algorithmes doivent être capables de faire cela de manière saine et que nous comprenons et maîtrisons.

7.9 Systèmes hybrides numérique et symbolique

Les algorithmes d'apprentissage par renforcement sont fortement ancrés dans les méthodes de mathématiques continues. Les mathématiques continues sont très différentes des mathématiques dicrètes : les mathématiques continues, c'est le monde de la continuité, des formes géométriques arrondies ; on manipule des entités, des nombres, auxquelles sont associées de nombreuses propriétés qui font que quand on les manipule, on utilise implicitement des propriétés essentielles, comme une notion de distance : si un algorithme manipule x, le résultat est presque tout le temps sensiblement le même que s'il manipule $x + \epsilon$; s'il manipule x et y, le résultat est souvent en relation avec la différence $|x - y|$, ou $(x - y)^2$ ou quelque chose comme cela.

Il est très courant qu'observant un algorithme d'apprentissage par renforcement en train de résoudre une tâche, un humain ait envie de lui donner des conseils : « fais ceci, ou cela » et en général, on ne sait pas comment traduire cette connaissance dans une forme que l'algorithme

puisse intégrer et manipuler, dans le monde des mathématiques continues. Reprenons l'exemple des échecs : voyant une configuration de jeu, l'humain pourrait « dire » à l'algorithme : « fais ça, et fais-le parce que ... » : on ne sait tout simplement pas transmettre cette connaissance à l'algorithme. Typiquement, l'humain fournit une connaissance de nature symbolique sur la situation : l'algorithme ne travaille pas du tout avec de telles connaissances symboliques. De même, il arrive fréquemment qu'un raisonnement logique simple permettrait à l'algorithme de déterminer l'action à réaliser ; mais l'algorithme ne manipule pas de symbole, ne fait pas de raisonnement logique et est donc incapable de cela.

Permettre à un algorithme d'apprentissage par renforcement de mixer des connaissances symboliques et des capacités de raisonnement logiques à son mode de fonctionnement qui a été décrit dans ce cours est un défi majeur pour les années à venir.

7.10 Fiabilité des expériences, leur reproductibilité et leur répétabilité

Cette dernière section n'est pas propre à l'apprentissage par renforcement. Elle concerne de très nombreux domaines dans lesquels sont étudiés des algorithmes stochastiques. Encore trop souvent, on lit des articles dans lesquels la partie expérimentation est dénuée de toute valeur et pourtant, des « conclusions » sur les performances expérimentales des algorithmes sont tirées. Nous voulons ici rappeler un certain nombre de principes nécessaires pour réaliser des études expérimentales un tant soit peu significatives.

Le titre de cette section mentionne les termes de « reproductibilité » et de « répétabilité ». La définition de ces notions n'est pas encore totalement bien établie et généralisée. Dans ce document, je dirai qu'une expérience est répétable si l'auteur fournit le programme avec laquelle elle a été réalisée et que quiconque arrive à reproduire les expérimentations menées par son auteur, en obtenant les « mêmes » résultats. Je dirai qu'une expérience est reproductible si en partant de sa description dans un article, on peut développer le code, réaliser les mêmes expériences que dans l'article et obtenir les « mêmes » résultats. La notion de reproductibilité est donc extrêmement exigeante et difficile à obtenir. Dans la pratique de notre communauté, la répétabilité est déjà un objectif difficile à atteindre qui nécessite beaucoup d'efforts et de précautions. Cela peut sembler bizarre mais le développeur lui-même d'un code qu'il a expérimenté un jour n'est pas toujours, et même rarement, capable de répéter ou reproduire ses expériences un peu plus tard. Déjà atteindre cet objectif nécessite une grande rigueur. Qu'une autre personne utilisant le code développé par le premier, réalise les mêmes expériences et obtienne les mêmes résultats est encore plus difficile : le simple fait d'exécuter le code sur une autre machine, éventuellement une autre version du système d'exploitation, éventuellement une autre version de python (ou autre langage), d'autres versions des paquetages python, ... rend l'obtention des mêmes résultats difficiles. Et pour réaliser les mêmes expériences, si le code qui a été utilisé originalement n'est pas fourni, il est illusoire à partir de la description qui en est faite dans les publications d'y arriver : si ce code n'est pas fourni, on ne peut pas savoir si on effectue strictement les mêmes expériences ou pas. Par ailleurs, les algorithmes dont nous traitons dans ce cours sont non déterministes et ils sont exécutés sur des ordinateurs qui partagent leur temps entre de multiples processus, avec des entrées-sorties qui leur sont propres, ... Aussi est-il illusoire d'obtenir exac-

tement le même résultat au bit près. À partir de là, il faut pouvoir juger ce que signifie obtenir le « même » résultat ; on est en train de comparer des distributions de performances et d'essayer de déclarer que deux de ces distributions sont les mêmes ou pas. C'est une question de statistiques, non résolue dans le cas général, en particulier celui qui nous intéresse. Ainsi, déterminer si deux expériences donnent le même résultat ou si un algorithme est meilleur qu'un autre sont des questions naturelles, pour lesquelles on n'a pas de réponse exacte. Ce sont des questions auxquelles la communauté scientifique, informaticienne en particulier, réflêchit actuellement. Ce qui suit dans cette section donne un ensemble de pistes qu'il faut suivre pour, sinon atteindre la perfection, du moins éviter les pièges les plus grossiers et essayer de tendre vers de tels objectifs.

Ceci étant dit, j'aimerais également insister sur la différence entre algorithme et programme. Un algorithme est une spécification d'une séquence de traitements censée réaliser une certaine tâche. Un agorithme n'est pas exécutable ; pour cela, il doit être transformé et implanté dans un programme. Cette étape peut sembler évidente : elle ne l'est pas. Tout d'abord, même en laissant de côté les algorithmes qui sont faux ou très insuffisamment spécifiés, il n'est pas rare qu'un algorithme soit mal spécifié, que certains « détails » manquent (omission involontaire des rédacteurs ou parce que supposé trivial). Dans ce cas, la personne qui l'implante doit imaginer ce qui manque et cela débouche sur plusieurs, voire une multitude, d'implantations possibles différentes, dans des langages différents, sur des systèmes différents, ... avec des performances en temps et en occupation mémoire qui varient, voire, bien pire, des sémantiques différentes. Ensuite, même un algorithme bien spécifié, que l'on peut exécuter « à la main dans sa tête » en le lisant, peut s'implanter de tas de manières différentes. Pour toutes ces raisons, la bonne pratique à adopter est de mettre à disposition sa propre implantation de l'algorithme, en s'assurant qu'elle pourra s'exécuter sur un grand nombre d'ordinateurs : on favorisera systématiquement le développement sous Linux (Ubuntu est un excellent choix), dans un langage « standard » (python, C, C++).

Ceci étant dit, dans la suite de cette section je parle d'« algorithme » sachant que je veux parler d'une implantation de l'algorithme, si possible celle fournie par ses auteurs.

7.10.1 Pourquoi fait-on des expériences ?

Même si c'est trop souvent le cas, une étude expérimentale ne doit pas avoir pour seul objectif de montrer que l'algorithme que l'on propose est meilleur que d'autres.

Une étude expérimentale doit permettre de comprendre les forces et faiblesses d'un algorithme. On sait (ça se démontre) qu'il n'y a pas d'algorithme qui soit bon/le meilleur pour tous les problèmes. Il faut donc essayer de caractériser le type de problèmes pour lesquels l'algorithme se comporte bien, et ceux sur lesquels il est à la peine.

Pour cela, il est très utile de concevoir des expériences sur des problèmes synthétiques abstraits qui ont exactement la propriété que l'on veut et pour laquelle on veut investiguer le comportement de l'algorithme.

7.10.2 Comparer son algorithme avec lesquels ?

Un algorithme doit faire mieux qu'un algorithme agissant au hasard. Ce n'est pas forcément nécessaire de l'écrire dans son article, mais ça peut être la première chose à vérifier.

On doit comparer son algorithme au meilleur algorithme de l'état de l'art. Si on ne bat pas l'état de l'art, c'est dommage mais ce n'est pas pour cela qu'il faut le mettre à la poubelle. Évidemment pouvoir écrire que son algorithme est le meilleur simplifie l'argumentation ; mais même sans être le meilleur, si on apporte des idées nouvelles, si on est capable d'expliquer pourquoi on ne bât pas l'état de l'art, ... l'idée peut valoir la peine.

Pour se comparer à d'autres algorithmes, le mieux est d'utiliser le code fourni par ses auteurs ; il ne faut pas le ré-écrire car même pour ceux qui sont précis (ce n'est pas toujours le cas), les papiers ne décrivent jamais tous les détails qui ont été implantés dans le programme. Si on n'a pas accès à l'implantation des auteurs, il ne reste qu'à se rabattre sur les performances qu'ils publient ; souvent néanmoins, quand on exécute soi-même le programme, les performances diffèrent : c'est bien ça le problème de la reproductibilité !

7.10.3 Comparer son algorithme sur quelles tâches ?

Beaucoup d'auteurs ont tendance à utiliser une tâche sur laquelle leur algorithme est meilleur, et ne pas parler de toutes les autres où il est moins bon (ou, on ne sait pas). L'objectif d'une étude expérimentale digne de ce nom est de déterminer les conditions dans lesquelles votre algorithme fonctionne bien, et celles où il est moins bon, voire mauvais.

Tout le monde sait (devrait savoir) qu'aucun algorithme ne peut pas être le meilleur pour tous les problèmes. Donc, si votre algorithme est le meilleur sur une tâche donnée, immanquablement, il donne de moins bons, voire de mauvais résultats sur d'autres : lesquelles ? C'est en identifiant ces faiblesses que l'on pourra déterminer ce qui pourrait être amélioré, ou ce qui est de portée de l'algorithme.

7.10.4 Une expérience ne prouve rien

et j'ajoute : des expériences non plus. Mais en multipliant les expérimentations, on peut accumuler des arguments.

7.10.5 Significativité des résultats expérimentaux

Les algorithmes étudiés ici sont stochastiques : exécutés deux fois dans le même contexte, leurs résultats sont différents. Ce sont des algorithmes difficiles à debugger.

Il faut exécuter plusieurs fois le programme dans le même environnement pour observer et quantifier la variabilité de ses performances. Donner seulement la performance moyenne ne veut rien dire. Il faut au minimum donner l'écart-type de la performance ou l'intervalle des performances observées sur un nombre suffisamment grand d'expériences pour que cette variabilité soit stable. Moins connue que la moyenne, la médiane est une mesure qui peut donner une information significative, plus pertinente que la moyenne, notamment quand la moyenne et la médiane

sont significativement différentes. La moyenne (et l'écart-type) n'a vraiment de sens que si une distribution est distribuée normalement. Il est important de comprendre les hypothèses sous-jacentes à des notions très connues que l'on emploie souvent sans se pré-occuper des hypothèses sous-jacentes : par exemple, une moyenne, un écart-type n'ont de sens que pour une distribution normale.

Il faut exécuter plusieurs fois son programme avec des graines de générateurs de nombres pseudo-aléatoires différentes. Le comportement d'un programme peut beaucoup changer d'une graine à une autre.

7.10.6 Quelques choses à faire

Nous listons ci-dessous quelques actions à effectuer pour aller vers la reproductibilité de ses expériences.

— générateur de nombres pseudo-aléatoires : une graine (*seed*) doit être utilisée pour l'initialiser ; la graine doit être mémorisée pour que l'exécution puisse être reproduite au bit près : avec un générateur correctement initialisé, le résultat d'exécutions paramétrées de la même manière doit être strictement identique à chaque exécution, au moins sur une machine donnée, avec un système d'exploitation inchangé. On suppose que cette machine n'exécute pas d'autre tâches sinon cela peut avoir des effets aléatoires sur l'exécution du programme.

L'initialisation du générateur de nombre pseudo-aléatoires se fait avec les instructions suivantes :

```
import numpy as np
# ...
graine = int("ThisIsTheStartOfTheSchoolYear", base=36)%2**31
gnpa = np.random.default_rng (graine)
```

puis on génère un nombre pseudo-aléatoire par l'instruction :

```
nombre_aleatoire = gnpa.random (1)
```

L'initialisation de la graine doit être réalisée le plus tôt possible durant l'exécution, dès que le programme dispose de sa valeur et que la bibliothèque a été importée.

Il faut tester votre programme pour vérifier qu'il génère bien exactement les mêmes résultats lors de deux excéutions dfférentes. Tant que ce n'est pas le cas, votre expérience ne sera pas reproductible, donc essentiellement inutile. Il faut tester le programme sur plusieurs machines : les résultats doivent être identiques au bit près.

— Votre programme possède un argument qui prend une graine en paramètre et cette graine est utilisée ; ainsi, en spécifiant deux fois la même graine, vous aurez deux fois la même exécution, strictement.

— Toujours mémoriser la ligne de commande qui a engendré un certain résultat expérimental, associé avec le numéro de version de votre programme (et quand votre programme est modifié, son numéro de version est modifié).

— Les figures qui illustrent les rapports et articles doivent être générées automatiquement avec des scripts qui permettent de reproduire exactement le figure (à nouveau au bit

près).

— Rendre son code disponible. En combinant ces deux derniers points, n'importe qui doit pouvoir regénérer les mêmes figures en exécutant le code que vous avez mis à disposition.

8 Un bref historique

Le livre de Sutton et Barto [41] donne de très nombreuses précisions concernant l'historique de l'apprentissage par renforcement. Je ne vais pas le copier ici mais seulement donner quelques repères chronologiques. Les livres de Bertsekas (*cf.* la bibliographie) donnent également de nombreuses références historiques.

L'un des premiers longs exposés sur la programmation dynamique est constitué par le livre de R. Bellman publié en 1957 [2]. L'introduction des problèmes de décision de Markov s'intègre dans les travaux menés durant les années 1950 sur la programmation linéaire et la programmation mathématique plus généralement. D'un point de vue social, cela s'insère dans la mécanisation des calculs et la résolution de problèmes par les premiers ordinateurs, en particulier de problèmes d'organisation de la production industrielle. Bellman a proposé l'algorithme d'itération sur les valeurs. L'algorithme d'itération sur les politiques a été inventé par un doctorant de Bellman, R. Howard, et publié dans sa thèse en 1960 [16].

Les méthodes de Monte Carlo ont été introduites pour la simulation de phénomènes aléatoires dès les années 1940 lors de l'avénement des tous premiers ordinateurs, notamment pour la conception de bombes atomiques.

La notion de différence temporelle a été introduite par Samuel [32, 33] dans ses travaux pionniers sur l'apprentissage automatique appliqué au jeu de dames américain. L'algorithme TD a été publié en 1988 par R. Sutton [39]. Il est inspiré des mécanismes d'apprentissage du comportement chez l'animal ; il formalise des lois de la dynamique du comportement, comme la loi de l'effet [45], le conditionnement opérant [37], la loi de Rescorla-Wagner [27] et s'appuie sur la notion de sélection du comportement par ses conséquences, étudiée par B.F. Skinner [36]. La thèse de Sutton demeure une lecture plus que recommandée [40]. Historiquement, le premier modèle informatique du conditionnement opérant repose sur des réseaux de neurones, publié en 1971 [11].

L'algorithme Q-Learning a été publié en 1989 par Ch. Watkins dans sa thèse de doctorat [47]. REINFORCE a été publié par Williams en 1992 [49]. L'algorithme SARSA a été publié en 1994 [30]. Les algorithmes acteurs-critiques sont une idée ancienne ; peut-être bien constituent-ils les premiers algorithmes proposés pour résoudre des problèmes d'apprentissage par renforcement, avant que ce terme n'existe dans le domaine de l'informatique. Ils ont connu un fort développement depuis les années 2010. Un article important sur ces méthodes est [1] en 1983. TRPO a été proposé en 2015 [34] ; PPO a été ensuite proposé en 2017 dans [35] ; SAC l'a été en 2018 dans [13].

Pour leur part, les réseaux de neurones ont connu une avancée considérable en 1986 avec l'introduction de l'algorithme de rétro-propagation du gradient de l'erreur qui a permis d'entraîner des réseaux constitués de plusieurs couches cachées (algorithme publié dès 1974 dans sa thèse

de doctorat par P. Werbos, mais semble-t-il resté ignoré). Très vite, des auteurs ont proposé d'utiliser un réseau de neurones pour représenter la fonction valeur dans l'algorithme TD. Dès 1986, Williams les étudie dans le cadre de l'apprentissage par renforcement [48, 50]. À la même période, G. Tesauro a créé et raffiné petit à petit un programme de jeu de Backgammon très fort (TD-Gammon [42]) capable de rivaliser avec des experts humains qui repose sur l'algorithme TD combiné avec un perceptron à une couche cachée ; à Cambridge autour de Rummery, plusieurs doctorants ont travaillé sur cette idée [30, 43, 44] mais aussi à Carnegie-Mellon [20] et à l'Université du Massachussetts [12]. DQN est donc un nom moderne pour une vieille idée.

D'un emploi délicat, les réseaux de neurones ont donné des résultats expérimentaux spectaculaires (à l'époque en tout cas) dans les travaux de M. Riedmiller en robotique et la thèse de doctorat de R. Coulom [9]. Riedmiller a proposé l'algorithme *neural Fitted Q-Iteration* en 2005 [28].

9 Pour aller plus loin

Ce cours a pour ambition de présenter succinctement et de manière intuitive l'apprentissage par renforcement. Ce n'est qu'un premier pas dans ce domaine fascinant. Pour aller plus loin, il y a le fameux livre de Sutton et Barto, *Reinforcement Learning: an introduction* [41], disponible gratuitement sur Internet. Pour ceux qui veulent mettre en application l'apprentissage par renforcement, il faut lire le livre de Max Lapan [17]. Pour ceux qui trouvent mes notes bien approximatives et qui veulent comprendre la théorie, le livre de Puterman *Markov decision processes* [26] demeure indispensable et les livres de Bertsekas sur la programmation dynamique *Dynamic Programming and Optimal Control* [4], *Neurodynamic programming* [6] et ses récents *Reinforcement learning and optimal control* [5] et *A course on reinforcement learning* [3] sont les indispensables compléments du Puterman. Pour la programmation dynamique approchée, il faut lire le livre de Warren Powell *Approximate dynamic programming* [24]. Pour le problème de planification, le livre de LaValle [18] est une excellente source introductive disponible en ligne.

Outil essentiel de l'apprentissage par renforcement, les réseaux de neurones doivent être compris par ceux qui veulent mettre en œuvre. Pour cela, le *Deep learning book* [10], disponible en ligne, est une lecture recommandée.

Enfin, pour mes lecteurs qui, comme moi, n'ont pas suivi de cursus vraiment sérieux en maths pendant leurs études, il n'est jamais trop tard pour acquérir les connaissances solides qui sont nécessaires pour aborder plus formellement les sujets abordés dans ce cours. Je recommande quelques livres qui m'ont beaucoup aidé et que je conserve à portée de main :

- Walter Appel, *Mathématiques pour la physique et les physciens*, H&K Éditions (mon édition, la 3^{e}, date de 2005)
- Thomas A. Garrity, *All the mathematics you missed*, Cambridge university Press, 2001
- Press *et al.*, *Numerical recipes in C* : indispensable pour connaître et comprendre les algorithmes numériques qui sont utilisés. Plutôt que l'implantation associée à ce livre, je conseille d'utiliser la *GNU Scientific Library* qui contient les mêmes algorithmes, très bien implantés, mais sans les explications concernant le fonctionnement des algorithmes.

Notations

On résume les notations utilisées :

$\mathcal{S}$ est l'ensemble des états d'un PDM

N est le nombre d'états : $N = |\mathcal{S}|$

s est un état ($s \in \mathcal{S}$)

$\mathcal{A}$ est l'ensemble des actions d'un PDM

P est le nombre d'actions : $P = |\mathcal{A}|$

a est une action ($a \in \mathcal{A}$)

$\mathcal{T}$ est l'ensemble des instants de décision d'un PDM

t est un instant de décision ($t \in \mathcal{T}$)

$\mathcal{P}$ est la fonction de transition d'un PDM

$\mathcal{R}$ est la fonction de retour d'un PDM

R est la fonction objectif à optimiser d'un PDM

r est un retour immédiat (variable aléatoire dont la loi est $\mathcal{R}$)

K est le nombre total de mises à jour ou le nombre total d'épisodes à réaliser

k est un indice de mise à jour ou un indice d'épisode

π est une politique

γ facteur de dépréciation d'un PDM déprécié

α taux ou coefficient d'apprentissage

V est la fonction valeur d'un état d'un PDM

Q est la fonction valeur d'une paire (état, action) d'un PDM

A est la fonction avantage d'une paire (état, action) d'un PDM

τ est un paramètre température ou une trajectoire (clair selon le contexte)

Θ est l'ensemble de définition des paramètres

θ est la valeur d'un paramètre : $\theta \in \Theta$

$\mathcal{L}$ est une fonction objectif

E est une fonction d'erreur

$\hat{}$ notation avec un chapeau, telle que $\widehat{Q}$: c'est une estimation d'une variable aléatoire : *cf.* page 22

$\tilde{}$ notation avec un tilde, telle que $\tilde{Q}$: c'est une approximation d'une quantité : *cf.* page 45

$a \equiv b$: dénote une définition : le symbole a est défini par l'expression symbolique b

$a = b$: dénote une égalité : a est égal à b

$a \leftarrow b$: dénote l'affectation des langages de programmation : la valeur de l'expression b est stockée dans la variable dénommée a

Index

Références

[1] A. Barto, R.S. Sutton et C. Anderson. "Neuronlike elements that can solve difficult learning control problems". In : *IEEE Trans. on Systems, Man, and Cybernetics* 13 (), p. 835-846 (cf. p. 75).

[2] R. Bellman. *Dynamic programming*. Princeton University Press, 1957 (cf. p. 75).

[3] D.P. Bertsekas. *A course on Reinforcement Learning*. Athena Scientific, 2023 (cf. p. 76).

[4] D.P. Bertsekas. *Dynamic Programming and optimal control*. Athena Scientific, 2017 (cf. p. 76).

[5] D.P. Bertsekas. *Reinforcement Learning and Optimal Control*. Athena Scientific, 2019 (cf. p. 76).

[6] D.P. Bertsekas et J.N. Tsitsiklis. *Neuro-dynamic programming*. Athena Scientific, 1996 (cf. p. 76).

[7] A.C. Browne et al. "A survey on Monte Carlo Tree Search methods". In : *IEEE Trans. on Computational Intelligence and AI in Games* 4.1 (2012), p. 1-43 (cf. p. 64).

[8] K. Chatzilygeroudis et al. "A survey on policy searh algorithms for learning robot controllers in a handful of trials". In : *IEEE Trans. on Robotics* (2019) (cf. p. 65).

[9] R. Coulom. "Apprentissage par renforcement utilisant des réseaux de neurones, avec des applications au contrôle moteur". Thèse de doct. Institut National Polytechnique de Grenoble, 2002 (cf. p. 76).

[10] I. Goodfellow, Y. Bengio et A. Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016 (cf. p. 76).

[11] S. Grossberg. "On the dynamics of operant conditioning". In : 33 (1971), p. 225-255 (cf. p. 75).

[12] V. Gullapalli. "Reinforcement Learning and its application to control". Thèse de doct. University of Massachussets, 1992 (cf. p. 54, 76).

[13] T. Haarnoja et al. "Soft actor-critic : off-policy maximum entropy deep reinforcement learning". In : *Proc. ICML*. 2018 (cf. p. 75).

[14] V. Heidrich-Meisner et Ch. Igel. "Evolution Strategies for Direct Policy Search". In : *Proc. Parallel Problem Solving from Nature (PPSN X)*. Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2008 (cf. p. 55).

[15] V. Heidrich-Meisner et Ch. Igel. "Similarities and differences between policy gradient methods and evolution strategies". In : *Proc. 16th European Symposium on Artificial Neural Networks (ESANN)*. 2008, p. 149-154 (cf. p. 55).

[16] R. Howard. *Dynamic programming and Markov processes*. MIT Press et John Wiley & Sons, 1958 (cf. p. 75).

[17] M. Lapan. *Deep Reinforcement Learning Hands-on*. 2e éd. Pakt Publishing, 2020 (cf. p. 76).

[18] S. LaValle. *Planning algorithms*. Cambridge University Press, 2006 (cf. p. 76).

[19] Y. Le Cun et al. "Efficient backprop". In : *Neural networks : tricks of the trade*. Sous la dir. de G.B. Orr et K-R. Müller. springer, 1998. URL : http://citeseer.nj.nec.com/lecun98efficient.html (cf. p. 96).

[20] L-J. Li. "Reinforcement learning for robots using neural networks". Thèse de doct. Carnegie-Mellon University, 1992 (cf. p. 76).

[21] A.W. Moore et C. Atkeson. "The parti-game algorithm for variable resolution reinforcement learning in multidimensional state space". In : *Machine Learning Journal* 21 (1995) (cf. p. 43).

[22] D. E. Moriarty, A. C. Schultz et J. J. Grefenstette. "Evolutionary Algorithms for Reinforcement Learning". In : *Journal of Artificial Intelligence Research* 11 (sept. 1999), p. 241-276 (cf. p. 55).

[23] R. Munos et A.W. Moore. "Variable Resolution Discretization in Optimal Control". In : *Machine Learning Journal* (2001) (cf. p. 43).

[24] W.B. Powell. *Approximate Dynamic Programming*. 2e éd. John Wiley et Sons, 2011 (cf. p. 76).

[25] Ph. Preux. *Fouille de données – Notes de cours*. Disponible sur https://philippe-preux.github.io/Documents/notes-de-cours-de-fouille-de-donnees.pdf. 2009 (cf. p. 96).

[26] M.L. Puterman. *Markov decision processes — Discrete Stochastic Dynamic Programming*. Probability, and mathematical statistics. John Wiley & Sons, 1994 (cf. p. 76).

[27] R.A. Rescorla et A.R. Wagner. "A theory of Pavlovian conditioning : Variations in the effectiveness of reinforcement and nonreinforcement". In : *Classical conditionning*. T. 2 : Current Research and theory. Prentice-Hall, 1972 (cf. p. 75).

[28] M. Riedmiller. "Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method". In : *Proc. European Conference on Machine Learning (ECML)*. T. 3720. Lecture notes in Computer Science. Springer, 2005, p. 317-328 (cf. p. 76).

[29] S. Ruder. *An overview of gradient descent optimization algorithms*. arvix 1609.04747. 2017 (cf. p. 91).

[30] G. Rummery et M. Niranjan. *On-line Q-learning using connectionist systems*. Rapp. tech. Technical Report CUED/F-INFENG/TR 166. UK : Department of Engineering, University of Cambridge, 1994 (cf. p. 75, 76).

[31] T. Salimans et al. *Evolution strategies as a scalable alternative to reinforcement learning*. arxiv :1703.03864. 2017 (cf. p. 55).

[32] A. Samuel. "Some Studies in Machine Learning using the game of checkers". In : *IBM J. of Research and Development* (1959), p. 210-229 (cf. p. 75).

[33] A. Samuel. “Some Studies in Machine Learning using the game of checkers II”. In : *IBM J. of Research and Development* (1967), p. 601-611 (cf. p. 75).

[34] J. Schulman et al. “Trust Region Policy Optimization”. In : *Proc. ICML*. 2015 (cf. p. 75).

[35] J. Shulman et al. *Proximal policy optimization algorithms*. arxiv 1707.06347. 2017 (cf. p. 75).

[36] B.F. Skinner. “Selection by consequences”. In : *Science* 213 (1981), p. 501-514 (cf. p. 75).

[37] J. Staddon. *The new behaviorism : Mind, Mechanism, and Society*. Psychology Press, 2001 (cf. p. 75).

[38] F.P. Such et V. Madhavan. *Deep neuroevolution : Genetic algorithms are a competitive alternative fo training deep neural networks for reinforcement learning*. arxiv :1712.06567. 2017 (cf. p. 55).

[39] R.S. Sutton. “Learning to predict by the method of temporal difference”. In : *Machine Learning* 3 (1988), p. 9-44 (cf. p. 75).

[40] R.S. Sutton. “Temporal credit assignement in reinforcement learning”. Thèse de doct. University of Massachussets, 1984 (cf. p. 75).

[41] R.S. Sutton et A.G. Barto. *Reinforcement learning : an introduction*. 2e éd. `http://incompleteideas.net/book/the-book-2nd.html`. MIT Press, 2018 (cf. p. 75, 76).

[42] G. Tesauro. “Temporal Difference Learning and TD-Gammon”. In : *Communications of the ACM* (1995) (cf. p. 76).

[43] C. Tham et R. Prager. “A modular Q-learning architecture for manipulator task decomposition”. In : *Proc. ICML*. 1994 (cf. p. 76).

[44] C.K. Tham. “Modular on-line function approximation for scaling up reinforcement learning”. Thèse de doct. Cambridge University, 1994 (cf. p. 76).

[45] E.L. Thorndike. “Animal Intelligence : An experimental study of the associative process in animals”. In : *Psychology Monographs* 2 (1898) (cf. p. 75).

[46] S. Thrun et A. Schwartz. “Issues in using function approximation in reinforcement learning”. In : *Proc. of the 1993 Connectionist Models Summer School*. 1993 (cf. p. 50).

[47] C. Watkins. “Learning from delayed rewards”. Thèse de doct. Cambridge University, 1989 (cf. p. 75).

[48] R. Williams. *Reinforcement learning in connectionist networks : A mathematical analysis*. Rapp. tech. Technical Report ICS 8605. Institute for Cognitive Science, University of California at San Diego, La Jolla. 1986 (cf. p. 76).

[49] R. Williams. “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In : *Machine Learning* 8.3-4 (1992), p. 229-256 (cf. p. 75).

[50] R. Williams. *Toward a theory of reinforcement-learning connectionist systems*. Rapp. tech. Technical Repordt NU-CCS-88-3, College of Computer Science, Northeastern University, Boston, MA. 1988 (cf. p. 76).

Table des matières

A Optimisation des paramètres d'une fonction

Différentes méthodes existent pour optimiser une fonction, c'est-à-dire déterminer un minimum ou un maximum de cette fonction. Dans les cas les plus simples, l'optimum existe, est unique et peut-être déterminé analytiquement : par exemple, on apprend à faire cela au lycée pour une parabole. Les fonctions pour lesquelles on peut déterminer analytiquement l'optimum sont exceptionnelles ; en pratique, il est impossible de déterminer analytiquement la position de l'optimum et, en général, il n'y a pas un mais des optima locaux. Sauf cas encore exceptionnel, les méthodes ne garantissent pas de trouver l'optimum global en temps fini mais un optimum local, et il est impossible de juger si cet optimum local est plus ou moins de même qualité que l'optimum global. Par le mot « qualité », on entend ici, pour une fonction unidimensionnelle, l'ordonnée : si on optimise la fonction f, notons x^* l'optimum global et x_o un optimum, on s'intéresse à l'écart entre $f(x_o)$ et $f(x^*)$, pas à l'écart entre x^* et x_o. Dans la pratique, la fonction à optimiser est généralement de dimension bien supérieure à 1, ce qui rend la tâche encore plus difficile : on travaille alors dans des espaces que nous ne sommes pas capables de visualiser et qui ont des propriétés bien plus complexes que celles auxquelles les fonctions en 1 ou 2 dimensions nous ont habitué. Dans la suite, on se place dans le cas de l'optimisation d'une fonction f à valeur réelle, de dimension quelconque, définie sur un certain domaine $\mathcal{D}$. On considère la minimisation de f, sachant que la maximisation consiste seulement à inverser le raisonnement et, d'un point de vue algorithmique, remplacer des $<$ par des $>$ ou des min par des max.

Même si leurs détails diffèrent, toutes les méthodes d'optimisation ont pour principe de prendre un point initial x_0 et, en fonction de $f(x_0)$, déterminer un x_1 et ainsi de suite, à partir d'un x_k engendrer un x_{k+1}. Ces points peuvent être un élément du domaine de définition de f ($x_k \in \mathcal{D}$) ou, pour un algorithme à base de populations, un ensemble d'éléments de ce domaine ($x_k \subset \mathcal{D}$). La manière dont est déterminé x_{k+1} à partir de x_k est la différence majeure entre toutes ces méthodes. Ces méthodes ont tendance à réaliser ce que l'on nomme une « descente », c'est-à-dire générer x_{k+1} tel que $f(x_{k+1}) < f(x_k)$. J'ai bien écris qu'elles ont « tendance à » faire cela car certaines d'entre-elles s'autorisent momentanément à prendre un x_{k+1} tel que $f(x_{k+1}) > f(x_k)$.

Notons que dans ce cours et plus généralement en apprentissage automatique, le processus d'optimisation à pour objectif de trouver les meilleurs paramètres d'un modèle pour qu'il représente au mieux des données. Pour illustrer ce point, si on considère un polynôme du second degré $f(x) = ax^2 + bx + c$, alors que souvent on connaît a, b et c et on cherche le x qui minimise f, en apprentissage, on cherche les a, b et c qui permettent de représenter au mieux un ensemble de points pour lesquels on dispose de couple $(x, f(x))$ (problème de régresion, *cf.* la section 5.1).

Dans la suite de cette section, on va présenter rapidement une méthode d'optimisation très utilisée en apprentissage par renforcement, la descente de gradient.

A.1 Descente de gradient

Pour présenter la méthode de descente de gradient, on commence par l'illustrer sur un exemple très simple de fonction unidimensionnelle. Ensuite, on montre son application pour

trouver les meilleurs paramètres d'un modèle, ici un perceptron.

A.1.1 Optimisation d'une fonction réelle

Intuitivement, une descente de gradient est une notion très simple :

1. on prend un point $x_k \in \mathcal{D}$;
2. on calcule $f(x_k)$ et on regarde comment la fonction varie en x_k :
3. si la fonction monte, on prend $x_{k+1} < x_k$ et on recommence en 2 ; si elle descend, on prend $x_{k+1} > x_k$ et on recommence en 2. Si elle est plate, on a atteint un minimum, on s'arrête.

Toute la question est alors de savoir comment on calcule x_{k+1}. Si la fonction monte en x_k, sa dérivée en x_k est positive, on a envie de prendre $x_{k+1} \leftarrow x_k - \alpha f'(x_k)$. Si elle descend, sa dérivée en x_k est négative, on a encore envie de prendre $x_{k+1} \leftarrow x_k - \alpha f'(x_k)$. En résumé, quel que soit le signe de la dérivée de f en x_k, on prend $x_{k+1} \leftarrow x_k - \alpha f'(x_k)$. Reste le coefficient α auquel on va pour l'instant donner une valeur petite. Et il faut aussi savoir s'arrêter ; en pratique, on n'attend jamais exactement l'optimum, donc la dérivée n'est jamais exactement nulle ; on arrête donc les itérations lorsque la dérivée est suffisamment petite. Tout cela mis bout à bout donne l'algorithme 12.

Algorithme 12 Principe de l'algorithme de descente de gradient. À l'issue de son exécution, x_k contient un minimum (local).

Nécessite: une fonction $f(x)$ et sa dérivée $f'(x)$
Nécessite: un coefficient α et un seuil ϵ
  initialiser x_0
  $k \leftarrow 0$
  répéter
    $k \leftarrow k + 1$
    $x_k \leftarrow x_{k-1} - \alpha f'(x_{k-1})$
  jusque $|f'(x_{k+1})| < \epsilon$

Illustrons cela sur un exemple, celui de la fonction représentée à la figure 20.

— Prenons $x_0 = -2$ et $\alpha = 0{,}1$: on atteint une valeur très proche de $-0{,}3$ en une dizaine d'itérations (en prenant $\epsilon = 10^{-5}$) : on a trouvé l'optimum global.

— Pour $x_0 = 2{,}5$ et $\alpha = 0{,}1$: on atteint également une valeur très proche de $-0{,}3$ en une quinzaine d'itérations (en prenant $\epsilon = 10^{-5}$) : on a trouvé à nouveau l'optimum global.

— Toujours pour $x_0 = 2{,}5$, prenons $\alpha = 0{,}01$: on atteint alors une valeur très proche de 1,7 qui est un optimum local.

— Pour $x_0 = -4$ et $\alpha = 0{,}1$: on diverge : l'algorithme va boucler à l'infini.

D'une manière générale, les initialisations de x_0 et de α ne sont pas du tout des choix faciles à réaliser. Généralement, on essaie plusieurs valeurs, on exécute l'algorithme et on conserve le meilleur optimum trouvé lors de ces essais.

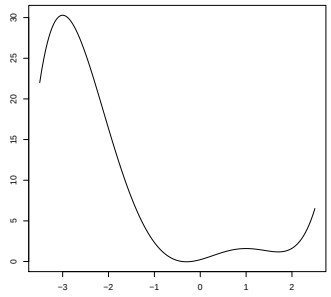


FIGURE 20 – Une fonction ($f(x) = 0{,}2x^5 + 0{,}15x^4 - 6{,}31/3x^3 + 1{,}59x^2 + 1{,}53x + 0{,}23$) pour illustrer la notion de descente de gradient. Cette fonction possède 4 optima, 2 minima et 2 maxima. Les 2 minima sont situés en $x = -0{,}3$ et $x = 1{,}7$. Selon la valeur de x_0 et de α, la descente atteint l'un des deux minima, ou diverge.

Le lecteur est plus qu'encouragé à implanter l'algorithme et à expérimenter sur des fonctions polynômiales de degré 5 par exemple.

Pour terminer, il existe des tas de variantes de la méthode de descente de gradient ; [29] décrit les plus connues.

A.1.2 Optimisation d'un modèle : le perceptron

On passe maintenant dans le monde de l'apprentissage où la fonction à minimiser est un modèle pour lequel on cherche les paramètres s'adaptant au mieux à des données. Comme modèle, on va prendre un perceptron linéaire, qui constitue l'élément de base des réseaux de neurones le plus simple. Un perceptron linéaire est composé de :

— n entrées que nous noterons e_1 ... e_n ;
— une entrée particulière dénommée biais qui vaut toujours 1 : $e_0 = 1$;
— des paramètres, ou poids synaptiques : $\theta_j, j \in \{0, ...n\}$;
— une sortie $s = \sum_{j=0}^{j=n} \theta_j e_j$.

Un perceptron linéaire réalise une tâche de régression : on dispose d'un ensemble d'exemples (x_i, y_i) avec $x_i \in \mathbb{R}^n$ et $y_i \in \mathbb{R}$. Quand on place une donnée x_i en entrée du perceptron ($e_j \leftarrow x_{i,j}$), on veut que sa sortie soit y_i. On notera $s(x_i)$ la valeur en sortie du perceptron lorsque la donnée x_i est placée sur ses entrées.

Le problème consiste à trouver les paramètres θ qui réalisent cette prédiction au mieux. Trouver ces meilleurs θ constitue un problème de régression linéaire pour lequel il existe de nombreuses méthodes. On va appliquer une descente de gradient.

Pour cela, il nous faut un critère d'optimisation : comment sait-on que tel jeu de paramètres est le meilleur ? Ou meilleur qu'un autre jeu de paramètres ? Classiquement, on utilise l'écart quadratique : quand on place une donnée x_i en entrée du perceptron, sa sortie s prend une certaine valeur : l'écart quadratique est $(s(x_i) - y_i)^2$. On va calculer l'écart quadratique moyen sur l'ensemble des données $\frac{1}{N} \sum_{i=1}^{i=N} (s(x_i) - y_i)^2$ et on va chercher à le minimiser.

Pour reprendre les notations vues plus haut, la fonction (objectif) à minimiser ici est $f \equiv \frac{1}{N}\sum_{i=1}^{i=N}(s(x_i)-y_i)^2$: les (x_i, y_i) sont ici fixés, ce sont les paramètres θ de f que l'on veut ajuster. Aussi, la dérivée de f qui nous intéresse est la dérivée par rapport aux paramètres, donc le gradient de f par rapport aux $n+1$ paramètres $\theta_0, ...\theta_n$, noté habituellement $\nabla_\theta f$. Ce gradient est un vecteur à $n+1$ dimensions et son j^{e} élément est $\frac{\partial f}{\partial \theta_j}$. Ça peut sembler compliqué mais c'est en fait très simple. Pour l'exemple (x_i, y_i), écrivons $f(x_i)$ en fonction des θ_i :

$$\begin{aligned} f(x_i) &= (s(x_i)-y_i)^2 \\ & \quad (\sum_{j=0}^{j=n}\theta_j x_{i,j} - y_i)^2 \end{aligned}$$

$f(x_i)$ est donc le carré d'une combinaison linéaire des composantes de x_i par les paramètres θ. Quand on calcule la dérivée de cette fonction par rapport à l'un des paramètres θ_j, seul $\theta_j x_{i,j}$ va contribuer un terme non nul : $\frac{\partial f}{\partial \theta_j} = \frac{\partial (\theta_j x_{i,j})^2}{\partial \theta_j} = 2(\theta_j x_{i,j} - y_i)x_{i,j}$. Donc, le gradient est :

$$\nabla_\theta f \equiv \begin{pmatrix} 2(s(x)-y) \\ 2(s(x)-y)x_1 \\ ... \\ 2(s(x)-y)x_j \\ ... \\ 2(s(x)-y)x_n \end{pmatrix}.$$

Comme plus haut mais cette fois-ci dans un espace à n dimensions, la mise à jour des paramètres va prendre la forme : $\theta_{\mathbf{k+1}} \leftarrow \theta_{\mathbf{k}} - \alpha\nabla_\theta f$. Le gradient remplace la dérivée de f unidimensionnelle.

Cette mise à jour doit être faite pour chacun des exemples. Donc, une itération de descente de gradient consiste à itérer sur chacun des exemples et à corriger chacun des paramètres. Cela donne l'algorithme 13.

Nous sommes dans un cas où la fonction possède un seul minimum ; cet algorithme est donc garanti de trouver le meilleur paramétrage. Cet algorithme est un algorithme parmi d'autres qui résolvent un problème célèbre, le problème de régression linéaire ; il détermine la droite des moindres-carrés rencontrée plus haut (*cf.* section 5.1.1).

L'algorithme 13 appelle plusieurs commentaires :

— cet algorithme peut être écrit de tas de manières différentes.

— La boucle **pour** démarrant à la ligne 6 ne doit pas être écrite comme cela est indiqué dans l'algorithme. Pour bien fonctionner, les exemples ne doivent pas être présentés dans le même ordre à chaque itération principale. Une solution simple consiste à générer une permutation p des entiers de 1 à N à chaque itération du **répéter** et d'utiliser ensuite non pas x_i mais x_{p_i}.

Algorithme 13 Principe de l'algorithme de descente de gradient appliqué à un perceptron lineaire. À l'issue de son exécution, $\theta_{\mathbf{k}}$ contient un minimum (local).

Nécessite: un ensemble de N exemples (x_i, y_i)

Nécessite: un coefficient α et un seuil ϵ

initialiser les paramètres $\theta_{\mathbf{0}}$
$k \leftarrow 0$
répéter
  $k \leftarrow k + 1$
  erreur $\leftarrow 0$
  pour $i \in \{1, ..., N\}$ **faire**
    $s_i \leftarrow \theta_0 + \sum_{j=1}^{j=n} \theta_j x_{i,j}$
    $g_j \leftarrow 0, j \in \{0, ..., n\}$
    pour $j \in \{0, ..., n\}$ **faire**
      $g_j \leftarrow 2(s_i - y_i)x_j$
    fin pour
    $\theta_{\mathbf{k}} \leftarrow \theta_{k-1} - \alpha \nabla_{\theta_{\mathbf{k-1}}}$
  fin pour
jusque $\sum_{j=0}^{j=n} |g_j| < \epsilon$

— Le critère d'arrêt peut être défini autrement. Nous l'avons écrit ici en nous inspirant de l'algorithme 12. On pourrait tout aussi bien s'arrêter quand la différence entre la valeur des paramètres entre deux itérations successives est inférieure à un seuil : $||\theta_{\mathbf{k}} - \theta_{\mathbf{k-1}}||_1 < \epsilon$. On pourrait aussi mesurer l'erreur de prédiction sur chaque exemple et arrêter les itérations lorsque cette erreur ne diminue plus ou est inférieure à un certain seuil. Mais en fait, la bonne manière de faire consiste à partitionner l'ensemble de N exemples en deux sous-ensembles, l'un étant utilisé pour calculer les paramètres comme indiqué dans l'algorithme, l'autre étant seulement utilisé pour mesurer l'erreur de prédiction à chaque itération du **répéter** ; le critère d'arrêt est alors basé sur cette erreur : on arrête quand elle ne diminue plus.

— En général, pour que l'algorithme fonctionne vraiment, α doit diminuer doucement au fil des itérations **répéter**.

— Enfin, cet algorithme a été exprimé ici comme fonctionnant sur un ensemble fixé de données. En fait, l'esprit est qu'il soit appliqué sur un flux de données, arbitrairement long, une donnée n'étant typiquement vue qu'une seule fois dans le flux. Dans ce cas, la boucle **pour** devient une boucle sur le flux, les paramètres étant mis à jour après lecture de chaque donnée. Cette version de l'algorithme est qualifiée de stochastique (« descente de gradient stochastique » donc). On obtient un algorithme adaptatif qui adapte constamment ses paramètres aux données qu'ils observent. Au fil du temps, le meilleur paramétrage peut varier, donc la valeur des paramètres ne doit jamais être figée et α doit maintenant varier de manière assez subtile. C'est typiquement le cas d'usage en apprentissage par renforcement : les données sont des échantillons mesurés le long d'une trajectoire

et non pas un jeu de données fixé une fois pour toute. Cela rend l'utilisation de perceptron et plus généralement de réseaux de neurones en apprentissage par renforcement un sujet complexe qui est longtemps resté difficile à maîtriser en pratique [23].

Avant de passer à la suite, notons que nous n'avons traité ici que du cas du perceptron linéaire, le plus simple, en particulier d'un point de vue mathématique. En général, la sortie d'un perceptron est définie par $s = \varphi(\sum_{j=0}^{j=n} \theta_j e_j)$ où φ est une fonction dénommée fonction d'activation. Le perceptron linéaire correspond au cas où φ est la fonction identité. Lorsque la fonction d'activation n'est pas l'identité, le calcul du gradient se complique, sans parler du cas, aujourd'hui courant, où la fonction d'activation n'est pas dérivable sur tout son domaine de définition.

A.2 Perceptron multi-couches

Un perceptron représente une fonction qui a exactement la forme de sa fonction d'activation. Ainsi, un perceptron linéaire produit une fonction linéaire : une droite en deux dimensions, un hyper-plan en plus grande dimension. Impossible pour un perceptron de représenter n'importe quelle fonction, encore moins si on ne sait pas à l'avance la forme de cette fonction. Pour cela, il faut utiliser une structure souple de représentation de fonctions, capable de représenter beaucoup de fonctions différentes.

En mathématiques, on sait depuis la fin du XIX^{e} siècle qu'il existe des familles de fonctions dont une combinaison linéaire des éléments peut représenter une grande variété de fonctions. Par exemple, une telle famille est constituée des monômes ; par combinaison linéaire des monômes, on peut représenter des tas de fonctions unidimensionnelles, les fonctions dérivables : on retrouve le développement en série de Taylor. Au XIX^{e} siècle, on a découvert que les sinus et cosinus constituent une autre telle famille, via les séries de Fourier. Puis on en a trouvé d'autres, comme la famille des fonctions logistiques, la famille des tangentes hyperboliques, la famille des gaussiennes, ... La fonction logistique a la forme : $\varphi(v) \equiv \frac{1}{1+e^{-av}}$, où a est un paramètre (*cf.* Fig. 21 à gauche). La fonction tangente hyperbolique a la forme $\varphi(v) \equiv \tanh(av)$ (*cf.* Fig. 21 à droite). Dans les deux cas, plus a est grand, plus la transition entre les deux valeurs extrêmes est brusque ; plus a est petit, plus la transition est progressive ; ces fonctions sont dérivables et bornées. Ces deux fonctions sont d'allure très similaire, sauf que la logistique varie entre 0 et 1, et la tangente hyperbolique entre -1 et 1 : celle-ci est symétrique par rapport à l'origine, ce qui est une propriété intéressante.

Aussi, on peut naturellement se dire qu'en utilisant un perceptron dont la fonction d'activation est issue d'une telle famille, en en combinant plusieurs, on va pouvoir représenter beaucoup de fonctions aux formes variées. Donc, au lieu d'utiliser un seul perceptron, utilisons-en plusieurs en les combinant. Cela donne l'idée du perceptron à une couche cachée. Une telle structure peut représenter n'importe qu'elle fonction continue, dérivable et bornée [24]. Un perceptron à une

23. Lecteur, oublie tout ce que j'ai expliqué dans ce cours, reviens juste à l'énoncé du problème d'apprentissage par renforcement et fais fonctionner un Q-learning neuronal sur le *cartpole* par exemple ; je te souhaite un bon amusement. Bien sûr, utiliser les bibliothèques de réseaux de neurones disponibles aujourd'hui, c'est tricher.

24. je simplifie un peu ici : l'ensemble des fonctions représentables dépend des fonctions d'activation utilisées.

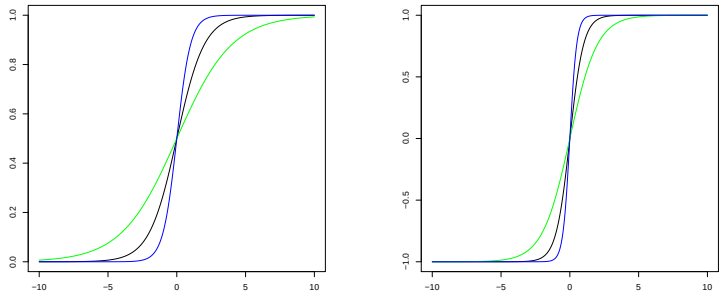


FIGURE 21 – À gauche, la fonction d'activation logistique ; à droite la tangente hyperbolique. Elles sont très ressemblantes, si ce n'est que la première varie entre 0 et 1, la seconde entre -1 et 1. On représente l'effet du paramètre a sur la forme de ces deux fonctions : $a = 0{,}5$ en vert, $a = 1$ en noir, $a = 2$ en bleu. En forme de la lettre S, on parle aussi de fonction sigmoïde.

couche cachée est constitué de :

- n entrées que nous noterons e_1 ... e_n ;
- P perceptrons à n entrées, ayant la même fonction d'activation ;
- les n entrées de chacun de ces P perceptrons sont connectées aux n entrées e_j ; ces perceptrons constituent la « couche cachée » ;
- un perceptron linéaire ayant P entrée et une sortie : l'entrée j de ce perceptron est connectée à la sortie du j^{e} perceptron de la couche cachée. La sortie de ce perceptron linéaire constitue la valeur prédite pour la donnée placée sur les entrées e_1 ... e_n ;
- chaque connexion entre une entrée e_j et le perceptron p possède un poids synaptique $\theta_{p,j}$;
- chaque connexion entre la sortie du perceptron p et le perceptron linéaire possède un poids synaptique θ_p.

Ce perceptron à une couche cachée possède donc $(n+1)P + (P+1)$ poids synaptiques, ou paramètres. L'objectif est de trouver la valeur de ces paramètres qui permet le meilleur ajustement aux exemples. C'est ce que nous avons fait plus haut dans la section 5.1.1.

Comment trouver ces poids ? Essayons de faire comme pour le perceptron linéaire. Pour cela, il nous faut calculer la fonction calculée par le réseau. C'est une combinaison linéaire de fonctions d'activation : $\sum_j \theta_j \varphi(\sum_k \theta_{j,k} x_{i,j})$ ... c'est compliqué ! Et on n'est pas au bout de nos peines puisqu'on doit maintenant calculer le gradient, donc la dérivée de cette fonction par rapport à chacun des $(n+1)P + (P+1)$ paramètres... Personne ne fait cela ! Mais entre le moment où on s'est dit qu'il fallait combiner plusieurs perceptrons, dans les années 1960, peut-être même avant, et le moment où on a été capable de calculer la dérivée de la sortie par rapport à l'un des paramètres, il s'est passé beaucoup de temps [25]. Nous verrons ce point dans la section suivante.

25. Thèse de P. Werbos en 1974 à laquelle visiblement les gens n'ont pas prêté attention puisque presque tout le monde croit que l'idée a été trouvée au même moment par plusieurs personnes, en 1986. (Je ne connais pas l'historique exact, donc j'écris les choses comme cela.)

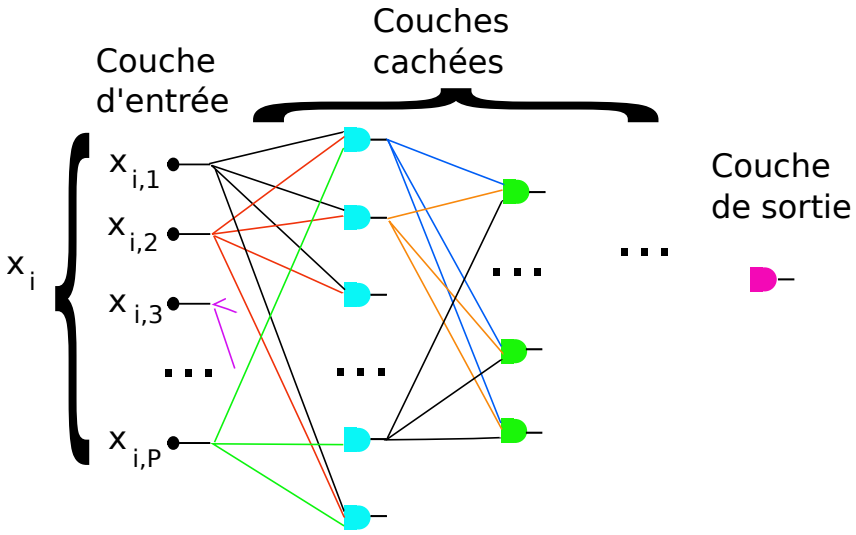


FIGURE 22 – Perceptrons à couches cachées.

La structure avec une seule couche cachée de perceptrons dont la fonction d'activation est une sigmoïde garantit de pouvoir représenter n'importe quelle fonction dérivable et bornée. Plus la forme de la fonction à représenter est compliquée, plus le nombre de perceptrons nécessaires dans la couche cachée augmente. Plutôt que d'avoir une seule couche cachée de grande taille, on peut utiliser plusieurs couches cachées en cascade, l'ensemble comprenant globalement beaucoup moins de perceptrons. On obtient une structure comme celle illustrée à la figure 22 : une donnée x_i est placée sur la couche d'entrée, propagée en entrée des perceptrons de la première couche cachée, via des connexions chacune paramétrée par un poids synaptique ; la valeur en sortie de chacun de ces perceptrons est transmise à l'entrée de chacun des perceptrons de la couche suivante, et ainsi de suite jusqu'au perceptron linéaire de sortie. Chaque connexion est paramétrée par un poids synaptique. La puissance de représentation d'un tel réseau provient du nombre de paramètres qui peut devenir très grand (actuellement, il existe des applications avec des milliards de tels poids synaptiques).

On peut aussi représenter des fonctions produisant plusieurs sorties (un vecteur). Il suffit de connecter plusieurs perceptrons linéaires en sortie, chacun étant connecté à la sortie de la dernière couche cachée.

Pour terminer, plutôt que de perceptron, il est courant de parler de « neurone », sous-entendu formel.

A.3 Rétro-propagation du gradient de l'erreur

On va très brièvement présenter la méthode utilisée pour calculer les paramètres d'un tel réseau de neurones. Cette méthode est une descente de gradient, adaptée à une telle structure de réseau de neurones. Les détails importent peu, on peut les trouver ailleurs (voir par exemple [19] ou [25, chap. 7]). L'idée consiste à corriger les poids en remontant le réseau depuis sa couche de sortie vers la couche d'entrée. Les poids sont corrigés dans cet ordre, en cascade, dans le sens

inverse donc de la propagation de la donnée de l'entrée du réseau vers sa sortie. On parle de rétro-propagation.

A.4 Autres méthodes d'optimisation

Même si elles sont moins populaires dans le domaine des réseaux de neurones pour des tâches d'apprentissage supervisé, d'autres méthodes existent pour optimiser leurs poids. Il existe des approches bayésiennes et des approches à base d'algorithmes basés sur des populations (stratégies d'évolution, algorithmes évolutionnaires).

B Régularisation

On explique brièvement la notion de régularisation car celle-ci est actuellement très utilisée dans les réseaux de neurones et en apprentissage par renforcement. On l'a mentionnée dans le cours ; nous indiquons ici quelques éléments permettant de mieux comprendre ce que c'est, son impact sur un processus d'optimisation et comment l'utiliser.

D'une manière générale, quand on optimise une fonction et qu'il existe plusieurs optima, la régularisation est une technique qui emmène l'algorithme vers certains optima plutôt que vers d'autres.

Supposons que l'on minimise une fonction objectif comme l'erreur quadratique moyenne (MSE), fonction que l'on rencontre très souvent en apprentissage automatique et que nous avons rencontrée plusieurs fois dans ce cours : $\mathcal{L} \equiv 1/N \sum_i (\hat{y}_i - y_i)^2$, où $\hat{y}_i$ est une fonction des paramètres dont on cherche la valeur minimale. Ce qui nous importe ici ce sont ses paramètres ; exprimons-le dans la notation : $\mathcal{L}(\theta) \equiv 1/N \sum_i (\hat{y}_i(\theta) - y_i)^2$. Si $\hat{y}_i(\theta)$ n'est pas juste une fonction linéaire, le problème possède plusieurs minima locaux, chacun correspondant à une certaine valeur des paramètres θ. Si on ajoute un terme à la fonction objectif qui dépend de ces paramètres, on régularise la fonction, c'est-à-dire que l'on change sa forme, en espérant que celle-ci sera plus simple pour que l'algorithme trouve le meilleur minimum. Cette technique fonctionne de manière assez impressionnante et semble un peu magique, même si les mathématiques nous en explique bien certains aspects. Par exemple, on peut exprimer le fait que parmi toutes les combinaisons de paramètres, on veut favoriser celles dont la norme est la plus petite ; au lieu de $\mathcal{L} \equiv 1/N \sum_i (\hat{y}_i - y_i)^2$ définie plus haut, on définira $\mathcal{L} \equiv 1/N \sum_i (\hat{y}_i - y_i)^2 + \lambda ||\theta||_2$. Le terme $||\theta||_2^2$ est le terme de régularisation : c'est le carré de la norme euclidienne du vecteur de paramètres, soit $\theta_1^2 + \theta_2^2 + ...\theta_n^2$; λ est un nombre dénommé le « coefficient de régularisation », qui prend une valeur réelle positive. On n'a *a priori* pas d'information sur la valeur à donner à λ ; typiquement, on essaie plusieurs valeurs et on choisit celle qui semble la meilleure. La norme euclidienne n'est qu'une fonction de régularisation que l'on peut utiliser parmi des tas d'autres. C'est tout un art de trouver une bonne fonction de régularisation qui marche bien ; dans ce cours, on en a indiqué quelques-unes pour certains algorithmes. D'une manière générale, on écrit la fonction objectif sous la forme $\mathcal{L}(\theta) \equiv$ la fonction que l'on veut minimiser $(\theta) + \lambda\Omega(\theta)$. Le terme de régularisation ne dépend pas des données, uniquement des paramètres.

Pourquoi ça marche ? D'une manière générale, l'optimisation d'une fonction consiste à trouver le minimum d'une fonction objectif tout en respectant des contraintes sur la solution : on veut trouver le minimum d'une fonction tel que ce minimum se trouve dans tel intervalle de valeur, ou sur la surface d'une certaine forme, ... C'est le domaine de l'optimisation sous contraintes : minimiser f tel qu'en ce minimum certaines contraintes soient satisfaites [26]. Lagrange a démontré que résoudre ce problème sous contraintes est équivalent à minimiser une fonction combinant f avec les contraintes, chacune multipliée par un coefficient, dit multiplicateur de Lagrange, cette combinaison portant le nom de lagrangien. Quand on combine la fonction à minimiser avec un terme de régularisation, cela correspond exactement à l'écriture d'un lagrangien ; on peut donc faire le raisonnement inverse et dire que régulariser est équivalent à ajouter une contrainte sur la solution. Nous n'irons pas plus loin dans l'explication qui nécessite des connaissances sérieuses en optimisation sous contraintes, leur exposé nous emménerait beaucoup trop loin, mais que le lecteur ait dans l'idée qu'ajouter une régularisation est équivalent à contraindre la solution peut déjà l'aider à comprendre et à apprendre à utiliser empiriquement cette technique. Pour déterminer un terme de régularisation utile, il faut se demander comment on peut caractériser l'optimum recherché.

C Estimation

« Estimation » est un terme qui a un sens mathémtique précis et nous utilisons ce terme avec ce sens dans ce cours. Nous donnons quelques précisions à ce propos.

Une estimation est une valeur que l'on donne à une variable aléatoire. Rappelons qu'une variable aléatoire est une entité que l'on peut mesurer et la valeur mesurée est susceptible de changer à chaque opération de mesure. En gros, tout ce que l'on mesure dans le monde réel est une variable aléatoire. Il n'y a que dans les mondes artificiels que l'on peut espérer mesurer la vraie valeur d'une grandeur et que l'aléa peut disparaitre pour laisser la place à la certitude.

Pour une variable aléatoire, on distingue donc sa « vraie » valeur [27] d'une valeur issue d'une ou plusieurs mesures. Dans ce cours, pour une variable aléatoire v, on note v^* sa vraie valeur (qui, comme nous sommes dans un monde artificiel dans ce cours, existe en tout cas du point de vue de la théorie mathématique sous-jacente) et on en note $\widehat{v}$ une estimation. On note bien la distinction entre « sa » vraie valeur, il y en a une seule, et « une » estimation, il y en a une infinité.

Une estimation $\widehat{v}$ peut être arbitrairement éloignée de sa vraie valeur v^* ; ce peut être une valeur arbitraire. Si la variable aléatoire à laquelle je m'intéresse est le nombre de passagers dans un bus, je peux répondre qu'une estimation en est 5, quel que soit le bus, quelle que soit l'heure, la date, ... Du moment, que je donne un nombre naturel, c'est une estimation : n'importe quel valeur dans le domaine de définition de la variable est une estimation correcte. On pourrait

26. Un programme linéaire est un cas particulier de problème d'optimisation sous contraintes.

27. l'existence même d'une telle « vraie » valeur relève de la philosophie : il y a toute une littérature à ce sujet et la question, comme toutes les questions intéressantes en philosophie, n'est pas tranchée et ne le sera probablement jamais.

même argumenter pour dire que tout réel en est une estimation approchée même si la vraie valeur est forcément un nombre naturel.

Bien entendu, dans la mise en œuvre, il nous importe que l'estimation soit la plus proche possible de la vraie valeur, ou du moins, que l'estimation apporte de l'information pertinente quant à la vraie valeur.

Comment obtient-on une estimation ? Si on veut connaître la longueur d'une barre en acier, on la mesure. Cette mesure est erronée : elle n'est correcte qu'avec une certaine précision. Dans un cours de métrologie on apprend, ou simplement par bon sens pratique on imagine, que l'on doit répéter la mesure de cette barre d'acier. À partir de cet ensemble de mesures, on va calculer une estimation de la longueur de la barre d'acier. L'idée la plus simple est de calculer la moyenne de ces valeurs. Cette valeur moyenne n'a de sens que si la longueur de la barre ne change pas à chaque mesure : si la température de son environnement varie, la longueur de la barre varie, quelle est alors la notion de longueur de la barre ? Si on suppose qu'on a pris soin de garder la barre dans un environnement strictement constant, calculer la moyenne des mesures fournit effectivement une estimation plus fiable de la valeur, quoique l'on puisse faire encore mieux : on peut retirer les valeurs extrêmes des mesures, ou calculer la médiane plutôt que la moyenne des mesures.

Mathématiquement, la valeur obtenue lors d'une mesure est décrite par une loi de probabilités, c'est-à-dire la probabilité d'observer une certaine valeur. Plusieurs cas se rencontrent :

— on connaît cette loi de probabilités. Plusieurs cas peuvent se présenter.

1. Si la loi n'est pas trop compliquée, on peut calculer la vraie valeur de la variable aléatoire avec une formule qui permet d'en calculer exactement la valeur en temps fini à partir des paramètres de la loi. Par exemple, si on sait que les observations sont distribuées selon une loi normale $\mathcal{N}(\mu, \sigma^2)$, la vraie valeur est μ.
2. Si la loi est plus compliquée, on n'a pas de moyen de calculer exactement la vraie valeur en temps fini. Il peut encore y avoir deux situations :
   (a) on dispose d'une formule exprimant une série dont on peut calculer les termes et qui tend asymptotiquement vers la vraie valeur : si tout va bien (la série converge uniformément) au bout d'un certain nombre de termes inclus dans la série, on aura une valeur précise avec un certain nombre de décimales. Si la série converge mais pas uniformément, on va avoir du mal à utiliser cette jolie formule car on ne sera pas capable de déterminer la précision du calcul réalisé.
   (b) on n'arrive pas à obtenir une telle formule. Dans ce cas comme on dispose de la loi de probabilités, on peut construire un « modèle génératif », c'est-à-dire un programme qui va engendrer des valeurs de cette variable aléatoire qui seront distribuées comme de véritables mesures de la variable. Ce cas très courant dans la pratique a été, et est encore, étudié et porte le nom de méthode de Monte Carlo. Typiquement, si on fait la moyenne des valeurs ainsi générées, celle-ci converge, lentement mais sûrement, vers la vraie valeur.

— on ne connaît pas cette loi de probabilités. Les choses se compliquent mais on se rapproche

de la vraie vie, des vraies applications.

Dans ce cas, tout ce que l'on peut faire est utiliser la ou les mesures dont on dispose et en faire la moyenne ou quelque chose comme cela [28]. On peut utiliser leur variance pour estimer la précision de la valeur obtenue.

Plusieurs cas sont possibles : i) on peut obtenir beaucoup de mesures : c'est le cas le plus simple : leur moyenne ou leur médiane va rapidement converger vers une valeur assez précise. ii) On ne peut pas obtenir beaucoup de mesures : c'est souvent le cas dans les vraies applications : dans ce cas, on essaie d'utiliser des hypothèses qui sont vraisemblables pour pallier le manque de données.

D Programmation linéaire

La programmation linéaire est un cas particulier d'un domaine qui se nomme la programmation mathématique. Malgré ce que le nom peut faire penser, la programmation linéaire (ou mathématique) n'a rien à voir avec la programmation [29]. La programmation linéaire est une manière de résoudre des problèmes d'optimisation sous contraintes ; dans ces problèmes, la fonction objectif est linéaire, c'est-à-dire qu'elle est une combinaison linéaire des variables pour lesquelles on cherche une valeur optimale. En outre, la solution recherchée doit vérifier un certain nombre de contraintes qui elles aussi, doivent s'exprimer par une combinaison linéaire des variables. Du point de vue applicatif, la programmation linéaire a une importance considérable : il existe une multitude de problèmes pratiques qui peuvent s'exprimer sous la forme d'une fonction objectif linéaire et de contraintes linéaires. La formulation d'une méthode générale permettant de résoudre efficacement ces problèmes a été réalisée dans les années 1940 et elle est depuis très utilisée en pratique.

Prenons un exemple très simple : supposons que nous ayons deux variables x_1 et x_2. On veut trouver la valeur de ces variables telles que leur somme $x_1 + x_2$ soit maximale et qu'elles vérifient les conditions (contraintes) suivantes :

$$\begin{cases} x_1 \geq 0 \\ x_2 \geq 0 \\ x_1 + 2x_2 \leq 4 \\ 4x_1 + 2x_2 \leq 12 \\ -x_1 + x_2 \leq 1 \end{cases}$$

28. Si l'on visualise l'ensemble des valeurs mesurées, il est courant que certaines mesures extrêmes se détachent des autres : dans ce cas, on calcule la moyenne sans tenir compte de ces valeurs extrêmes. Une autre manière de pratiquer est de calculer la médiane des mesures plutôt que leur moyenne : la médiane est beaucoup moins sensible aux valeurs extrêmes. Ces techniques relèvent de ce que l'on appelle l'estimation robuste, robuste devant se comprendre ici comme insensible aux valeurs extrêmes, sous-entendu, aberrantes.

29. C'est comme la « programmation dynamique » qui n'a rien à voir avec la programmation. D'ailleurs, quand il lui a fallu donner un nom à ce qu'il venait de découvrir, Bellman a eu l'idée de l'appeler programmation dynamique en référence à la programmation linéaire, le qualificatif « dynamique » étant utilisé pour donner le sentiment que ce qu'il proposait aller bien au-delà de la programmation linéaire. C'est de la comm !

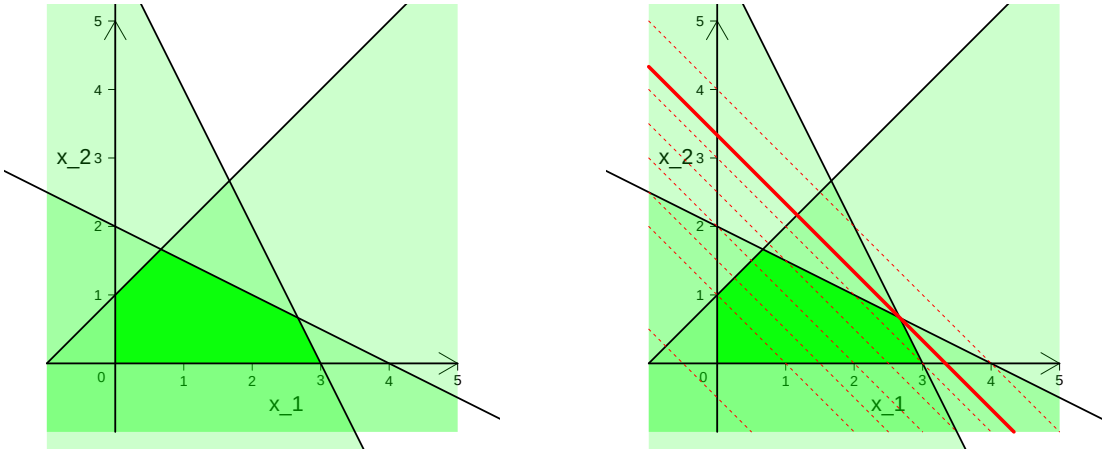


FIGURE 23 – Interprétation géométrique d'un ensemble de contraintes linéaires. À gauche, on a représenté chacune des 5 contraintes. La zone centrale où le vert est plus intense correspond à la zone dans laquelle les 5 contraintes sont vérifiées. La solution est quelque part dans cette zone. À droite : par définition de la fonction objectif, on cherche une droite d'équation $x_1 + x_2 = Cste$. On a indiqué un ensemble de telles droites en pointillés. La solution au problème est donnée par la droite « la plus haute » dont l'intersection avec la zone verte est non vide. C'est la droite plus épaisse. Cette intersection est l'un des sommets du pentagone.

Les deux premières contraintes sont dites de non négativité, les suivantes sont les contraintes principales.

Ce problème peut s'exprimer d'un point de vue géométrique : dans un problème linéaire, chaque contrainte correspond à une droite qui découpe le plan en deux zones, l'une pouvant contenir la solution, l'autre pas. Ainsi, la partie gauche de la fig. 23 illustre cette idée : on a représenté les 5 contraintes (5 contraintes = 5 droites), chacune définissant un demi-plan. L'intersection de ces 5 demi-plans est indiquée en vert plus soutenu : la solution est l'un des points de ce pentagone. Il peut arriver que les contraintes aient une intersection vide ; dans ce cas, le problème n'a pas de solution (il est dit « infaisable »). Il peut aussi arriver que l'intersection soit non bornée et aille à l'infini ; dans ce cas, on dit que le problème n'est pas borné, il n'a pas de solution finie. Dans l'exemple présent, le problème est faisable et borné. Il reste à déterminer le point de la zone verte qui maximise $x_1 + x_2$. Dit autrement, on cherche la droite $x_1 + x_2 = Cste$ pour laquelle $Cste$ est maximale ; les droites ayant cette équation sont toutes parallèles les unes aux autres, elles coupent l'axe des x_2 à la valeur $Cste$ et leur coefficient directeur est -1. On en a représenté quelques-unes en pointillés à droite de la figure 23. Celle qui nous intéresse est « la plus haute » dont l'intersection avec la zone vert sombre n'est pas vide : elle est représentée avec un trait rouge plein épais.

D'une manière générale, la forme « canonique » d'un programme linéaire de maximisation

est la suivante :

$$\begin{aligned} &\max \mathbf{c}'\mathbf{x} \\ &\text{tel que } \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ &\text{et } \mathbf{x} \geq \mathbf{0} \end{aligned}$$

où :

— $\mathbf{x}$ est un vecteur dont chaque élément est une inconnue du problème $\mathbf{x} = (x_1, x_2)'$ dans l'exemple plus haut ;

— $\mathbf{c}$ est le vecteur des coefficients des inconnues dans la fonction objectif : $\mathbf{c} = (1, 1)$ dans l'exemple plus haut ;

— la notation $\mathbf{c}'$ correspond au vecteur $\mathbf{c}$ transposé, c'est-à-dire que c'est un vecteur dont les éléments sont en ligne au lieu d'être en colonne. Aussi, $\mathbf{c}'\mathbf{x}$ est le produit scalaire des vecteurs $\mathbf{c}$ et $\mathbf{x}$;

— $\mathbf{A}$ est la matrice des coefficients des inconnues dans les contraintes ;

— $\mathbf{b}$ est le vecteur contenant les constantes apparaissant dans les contraintes.

On a donc :

$$\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 4 & 2 \\ -1 & 1 \end{pmatrix}$$

et

$$\mathbf{b} = \begin{pmatrix} 4 \\ 12 \\ 1 \end{pmatrix}$$

Une propriété remarquable est que la solution à ce type de problèmes se situe sur un sommet de la zone vérifiant les contraintes. Nous disons « la » solution mais il y a des cas dans lesquels il existe une infinité de solutions qui se situent toutes sur un segment de droite. Et, comme il a été écrit plus haut, il y a des cas dans lesquels il n'y a pas de solution.

Un algorithme très connu pour résoudre ce type de problèmes porte le nom de simplexe. Nous ne le décrivons pas ici. Une autre approche existe et se nomme une méthode de point intérieur ; c'est une méthode basée sur une descente de gradient, qui converge vers l'optimum. Dans la pratique, on utilise une implantation de ces algorithmes disponible dans une bibliothèque logicielle. Il suffit d'exprimer la fonction objectif et les contraintes, puis d'appeler la fonction qui résout le problème et renvoie la solution.

D.1 Problème dual

À un problème d'optimisation linéaire avec des contraintes linéaires est associé un autre problème d'optimisation lui aussi linéaire avec des contraintes linéaires : le problème dual. Le problème original se nomme le problème primal ; le problème dual du problème dual est le

problème primal. Ces deux problèmes sont très liés l'un à l'autre ; d'une certaine manière, les deux problèmes éclairent de deux manières différentes la même réalité : la solution de l'un donne des informations complémentaires à la solution de l'autre.

La notion de problème dual concerne tous les problèmes d'optimisation avec contraintes. Dans le cas du problème linéaire, ces problèmes primal et dual possèdent des propriétés particulières.

Avec la formulation indiquée plus haut pour le problème primal :

$$\begin{array}{l}\max \mathbf{c}'\mathbf{x} \\ \text{tel que } \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ \text{et } \mathbf{x} \geq \mathbf{0}\end{array}$$

le problème dual est :

$$\begin{array}{l}\min \mathbf{b}'\mathbf{y} \\ \text{tel que } \mathbf{A}'\mathbf{y} \geq \mathbf{c} \\ \text{et } \mathbf{y} \geq \mathbf{0}\end{array}$$

Ici, c'est le vecteur $\mathbf{y}$ qui est l'inconnue ; ses composantes sont nommées les variables duales.

Si le primal possède N variables et P contraintes, le dual contient P variables et N contraintes.

Un théorème très important montre que si un problème est borné et faisable, alors son dual est borné et faisable également : ils ont tous les deux au moins une solution. Par ailleurs, si le primal n'est pas borné, le dual est infaisable. De plus, si le primal est borné et faisable, le théorème de la dualité forte indique que la valeur maximale de la fonction objectif du primal est égale à la valeur minimale de la fonction objectif du dual : $\mathbf{c}'\mathbf{x}^* = \mathbf{b}'\mathbf{y}^*$ où $\mathbf{x}^*$ est une solution au primal et $\mathbf{y}^*$ est une solution au dual.

Le dual de l'exemple vu plus haut

$$\begin{array}{l}\max x_1 + x_2 \\ \text{tel que } \left\{\begin{array}{l} x_1 \geq 0 \\ x_2 \geq 0 \\ x_1 + 2x_2 \leq 4 \\ 4x_1 + 2x_2 \leq 12 \\ -x_1 + x_2 \leq 1 \end{array}\right.\end{array}$$

est donc :

$$\min y_1 + y_2 + y_3$$
$$\text{tel que } \begin{cases} y_1 \geq 0 \\ y_2 \geq 0 \\ y_3 \geq 0 \\ y_1 + 4y_2 - y_3 \geq 1 \\ 2y_1 + 2y_2 + y_3 \leq 1 \end{cases}$$

D.2 Au-delà de la programmation linéaire

On n'entrera pas dans les détails ici, mais il existe un autre type de problèmes relativement faciles à résoudre, les problèmes convexes, d'où la notion de « programmation convexe ».