Chapitre 1

Apprentissage par renforcement

1.1. Introduction

Par rapport aux méthodes de planification présentées au chapitre **??**, dans lesquelles l'agent connaît *a priori* la fonction de transition et la fonction de récompense du problème décisionnel de Markov auquel il est confronté, les méthodes d'*apprentissage par renforcement* permettent de traiter les situations dans lesquelles les fonctions de transition et de récompense ne sont pas connues *a priori*.

EXEMPLE.– Reprenons l'exemple de l'entretien d'une voiture abordé dans l'introduction du chapitre précédent (voir le paragraphe **??**). Si on doit maintenant faire l'entretien d'un modèle de voiture que nous n'avons jamais rencontré précédemment et qui nous est livrée sans manuel technique, nous n'avons pas assez de connaissance pour modéliser ce problème sous forme d'un MDP. En effet, il se peut que cette voiture soit, par exemple, plus robuste aux fuites d'huile et donc nous ne connaissons pas les probabilités de panne dans ce cas. De même, nous ne savons pas le prix des pièces de rechange et il est donc impossible de connaître le coût d'une action à l'avance. Une solution possible dans ce cas est d'utiliser les méthodes d'apprentissage par renforcement qui s'appuient sur une succession d'expériences : en expérimentant, nous allons petit à petit pouvoir estimer directement la valeur des actions, au sens de la fonction de valeur d'action décrite au chapitre précédent, pour chaque état de la voiture, sans forcément avoir à apprendre les différentes probabilités de panne. Et cette estimation de la valeur des actions permettra finalement de choisir l'action optimale en fonction de l'état de la voiture.

Chapitre rédigé par Olivier SIGAUD et Frédérick GARCIA.

Le présent chapitre est donc consacré au traitement de ce problème plus complexe que le précédent.

1.1.1. *Bref aperçu historique*

La présentation adoptée dans la suite de ce chapitre fait l'objet d'une reconstruction *a posteriori* qui ne rend pas compte de l'enchaînement historique des idées. Avec un souci de clarté des concepts, notre présentation s'appuiera sur l'ouvrage de Sutton et Barto [SUT 98], qui constitue une synthèse d'une qualité telle qu'il est difficile de s'en démarquer. Cependant, avant d'en venir au cœur de cette présentation, nous donnons un bref aperçu de la succession des étapes qui ont conduit à la formulation actuelle des modèles théoriques de l'apprentissage par renforcement, en nous focalisant sur le développement de modèles informatiques.

La plupart des méthodes algorithmiques de l'apprentissage par renforcement reposent sur des principes simples issus de l'étude de la cognition humaine ou animale, comme par exemple le fait de renforcer la tendance à exécuter une action si ses conséquences sont jugées positives, ou encore de faire dépendre ce renforcement de la durée qui sépare la récompense de l'action, ou de la fréquence à laquelle cette action a été testée. Cet arrière-plan psychologique a été présenté dans [SIG 04].

Les premiers travaux en informatique représentatifs du cadre de l'apprentissage par renforcement datent approximativement de 1960. En 1961, Michie [MIC 61] décrit un système capable d'apprendre à jouer au morpion par essais et erreurs. Puis Michie et Chambers [MIC 68] écrivent en 1968 un programme capable d'apprendre à maintenir un pendule inversé à l'équilibre. Parallèlement, Samuel [SAM 59] réalise un logiciel qui apprend à jouer aux dames en utilisant une notion de différence temporelle. Les deux composantes, exploration par essais et erreurs et gestion de séquences d'action par différences temporelles vont être au cœur des modèles ultérieurs. La synthèse entre les deux courants est réalisée par Klopf [KLO 72, KLO 75]. Dans la lignée de ces travaux, les principaux acteurs du développement de l'apprentissage par renforcement en informatique, Sutton et Barto, implémentent en 1981 [SUT 81] un perceptron linéaire dont l'équation de mise à jour dérive directement des théories développées en psychologie expérimentale par Rescorla et Wagner [RES 72]. Jozefowiez fait très clairement apparaître dans sa thèse [JOZ 01] que l'équation de Rescorla-Wagner, qui est d'une importance considérable pour les modèles de l'apprentissage animal, n'est rien d'autre que la version approximée par un perceptron linéaire de l'algorithme $TD(0)$ présenté au paragraphe 1.4.1. Cette équivalence explique que le recours aux algorithmes d'apprentissage par renforcement soit aussi appelé « programmation neuro-dynamique »[1].

1. Plus précisément, le terme *neuro-dynamic programming* définit l'ensemble des techniques couplant programmation dynamique ou apprentissage par renforcement et méthodes de généralisation [BER 96].

Sur ces bases, Sutton et Barto proposent en 1983 avec Anderson [BAR 83] une méthode, AHC-LEARNING[2], qui est considérée comme le véritable point de départ des travaux qui font l'objet de ce chapitre. De façon intéressante, AHC-LEARNING repose sur une architecture acteur-critique dont nous verrons au paragraphe 1.4.5 qu'elle est au cœur du dialogue qui s'instaure à présent entre la modélisation informatique et la modélisation neurophysiologique de l'apprentissage par renforcement chez l'animal.

La formalisation mathématique des algorithmes d'apprentissage par renforcement telle que nous la connaissons aujourd'hui s'est développée à partir de 1988 lorsque Sutton [SUT 88] puis Watkins [WAT 89] ont fait le lien entre leurs travaux et le cadre théorique de la commande optimale proposée par Bellman en 1957 avec la notion de MDP [BER 95].

1.2. Apprentissage par renforcement : vue d'ensemble

1.2.1. *Approximation de la fonction de valeur*

Le principe de l'apprentissage par renforcement repose en premier lieu sur une interaction itérée du système apprenant avec l'environnement, sous la forme de l'exécution à chaque instant n d'une action a_n depuis l'état courant s_n, qui conduit au nouvel état s'_n et qui fournit la récompense r_n. Sur la base de cette interaction, une politique est petit à petit améliorée. En pratique toutefois, la plupart des algorithmes d'apprentissage par renforcement ne travaillent pas directement sur la politique, mais passent par l'approximation itérative d'une *fonction de valeur*, issue de la théorie des MDP présentée au chapitre précédent.

La notion de fonction de valeur, qui associe à chaque état possible une estimation de la valeur pour l'agent de se situer en cet état en fonction de l'objectif visé[3], est fondamentale en apprentissage par renforcement. Elle permet de distinguer clairement l'apprentissage par renforcement de toutes les autres méthodes d'optimisation basées sur la simulation, comme les algorithmes génétiques [GOL 89], la programmation génétique [KOZ 92], le recuit simulé [KIR 87], etc., qui permettent aussi de construire des politiques optimales, mais qui n'exploitent pas pour cela la structure temporelle des problèmes décisionnels considérés, comme le fait l'apprentissage par renforcement à l'échelle de l'expérience (état courant, action, récompense, état suivant).

La plupart des méthodes de l'apprentissage par renforcement que nous allons voir dans ce chapitre sont étroitement liées aux algorithmes de programmation dynamique

2. *Adaptive Heuristic Critic learning.*

3. Il s'agit donc là d'une notion similaire à celle introduite en théorie des jeux sous le nom de fonction d'évaluation.

présentés au chapitre précédent. En effet, le problème de définir et de calculer des politiques optimales a été formalisé dans le cadre des MDP depuis la fin des années 50, et l'apprentissage par renforcement peut être perçu comme une simple extension des algorithmes classiques de programmation dynamique au cas où la dynamique du processus à contrôler n'est pas connue *a priori*.

1.2.2. *Méthodes directes et indirectes*

Les méthodes d'apprentissage par renforcement sont dites indirectes ou directes selon que l'on maintient ou non un modèle explicite des fonctions de transition et de récompense du MDP que l'on cherche à contrôler. A ce titre, les méthodes de programmation dynamique vues au chapitre précédent peuvent être considérées comme des cas limites de méthodes indirectes, où le modèle maintenu est le modèle exact par hypothèse.

Lorsque le modèle de la dynamique n'est pas connu initialement, les méthodes indirectes doivent donc *identifier* en ligne ce modèle. Dans les cas discrets que nous considérons, cette identification se fait simplement par maximum de vraisemblance. D'autre part, il s'agit de rechercher sur la base du modèle courant une fonction de valeur ou une politique optimale. Il est alors possible d'exploiter les algorithmes classiques de programmation dynamique.

Les méthodes directes ne passent pas par l'identification d'un modèle de la dynamique du système : les paramètres cachés $p(s'|s,a)$ et $r(s,a)$ ne sont pas estimés et seule la fonction de valeur est mise à jour itérativement au cours du temps. Le principal avantage est ici en termes de place mémoire nécessaire et, historiquement, l'apprentissage par renforcement revendiquait d'être une méthode directe. Ainsi, les algorithmes les plus classiques de l'apprentissage par renforcement que nous présentons ci-dessous partagent tous, chacun à sa manière, le même principe général qui est d'approximer à partir d'expériences une fonction de valeur optimale V sans nécessiter la connaissance *a priori* d'un modèle du processus, et sans chercher à estimer ce modèle à travers les expériences accumulées. Plutôt qu'une fonction V qui associe une valeur à chaque état, nous verrons que l'on peut aussi chercher à approximer une fonction Q qui associe une valeur à chaque action réalisée dans chaque état.

1.2.3. *Apprentissage temporel, non supervisé et par essais et erreurs*

Quelques caractéristiques permettent de distinguer l'apprentissage par renforcement des autres formes d'apprentissage.

En informatique, on distingue d'une part l'apprentissage dit « supervisé », dans lequel un « instructeur » indique à l'apprenant quelle réponse il aurait dû fournir dans

un contexte donné et, d'autre part, l'apprentissage dit « non supervisé », dans lequel l'apprenant doit identifier par lui-même la meilleure réponse possible.

Dans le cadre de l'apprentissage supervisé, la meilleure réponse possible est fournie à l'apprenant si bien qu'il n'a pas besoin de la rechercher. C'est le cas, par exemple, avec l'algorithme de rétro-propagation du gradient dans les réseaux de neurones à couches [CHA 95]. L'apprentissage par renforcement se distingue de l'apprentissage supervisé par le fait que, lorsqu'il reçoit un premier signal d'évaluation, l'apprenant ne sait toujours pas si la réponse qu'il a donnée est la meilleure possible ; il doit essayer d'autres réponses pour déterminer s'il peut recevoir une meilleure évaluation.

Parmi les méthodes d'apprentissage non supervisé, il faut distinguer les méthodes dans lesquelles l'apprentissage se fait sans évaluation, par exemple en mémorisant simplement des associations observées, et les méthodes dans lesquelles une évaluation est fournie à l'apprenant, comme c'est le cas pour l'apprentissage par renforcement. L'apprentissage par renforcement est donc une forme d'apprentissage non supervisé reposant sur une évaluation. La présence de cette évaluation implique un mode de fonctionnement par essais et erreurs.

Enfin, l'apprentissage par renforcement porte sur des séquences temporelles. Lorsque l'on cherche à classer des données décrivant des iris, même si le processus d'apprentissage est itératif (le professeur fournit des exemples l'un après l'autre), le temps ne change rien à l'affaire, au sens où l'ordre dans lequel sont présentés les exemples importe peu. Au contraire, en apprentissage par renforcement, tout choix d'une action exécutée dans un état a des conséquences à plus ou moins long terme et la donnée de la récompense immédiate n'est rien sans les autres données qui correspondent à la suite de l'interaction entre l'agent et l'environnement. On peut en effet être confronté à des *récompenses retardées* et les méthodes de l'apprentissage par renforcement offrent des outils permettant de gérer cette difficulté.

Dès lors, alors que l'apprentissage par simple mémorisation des associations observées peut se faire directement, l'apprentissage par renforcement induit par nature une activité d'exploration de la part de l'agent. Il faut que cet agent explore son environnement pour déterminer dans quelles circonstances il est puni ou récompensé et quelles sont les séquences d'action qui lui permettent d'atteindre les récompenses plutôt que les punitions. Cette nécessité d'explorer est à la source de la présence dans tous les travaux d'apprentissage par renforcement du dilemme posé par le compromis entre exploration et exploitation.

1.2.4. *Le dilemme exploration/exploitation*

Pour régler la politique de façon à maximiser sa récompense sur le long terme, la phase d'apprentissage se trouve confrontée à la nécessité de trouver un compromis

entre l'exploitation qui consiste à refaire les actions dont on connaît déjà la récompense à laquelle elles donnent lieu et l'exploration qui consiste à parcourir de nouveaux couples (état, action) à la recherche d'une récompense cumulée plus grande, mais au risque d'adopter parfois un comportement sous-optimal. En effet, tant que l'agent n'a pas exploré la totalité de son environnement, il n'est pas certain que la meilleure politique qu'il connaît est la politique optimale.

En conséquence, toutes les preuves de convergence des algorithmes d'apprentissage par renforcement exigent en théorie que toutes les transitions soient expérimentées [WAT 92]. En pratique, toutefois, on se contente d'une exploration partielle qui suffit en général à découvrir une politique satisfaisante.

L'exploration peut porter sur le choix de l'état s_n ou sur celui de l'action a_n. La plupart du temps, le choix le plus naturel concernant s_n est de poser à chaque itération $s_{n+1} = s'_n$, c'est-à-dire de laisser à la dynamique du système le soin de gérer l'exploration de l'espace d'états. D'une part, cela permet de se concentrer sur les zones importantes de l'espace d'états, accélérant ainsi la convergence (c'est aussi une des raisons de l'efficacité d'algorithmes comme RTDP, décrit au chapitre **??** du volume 2). D'autre part, c'est souvent nécessaire du fait de la structure des systèmes sur lesquels est réalisé l'apprentissage. Par exemple, dans un contexte robotique, on ne maîtrise pas le choix de l'état suivant. Il est clair toutefois que lorsque s'_n est un état absorbant, c'est-à-dire un état dans lequel le système reste une fois qu'il y est entré, il est nécessaire de réinitialiser le processus en tirant par exemple au hasard un nouvel état s_{n+1} dans S.

La seconde heuristique concerne le choix de l'action a_n. Une action choisie uniformément dans A à chaque itération satisfait bien le critère de convergence, avec une *exploration* maximale, mais un tel choix est peu efficace pour deux raisons. Tout d'abord, la valeur de la meilleure action en chaque état étant aussi souvent mise à jour que la valeur de la plus mauvaise action, l'apprentissage se fait sûrement mais très lentement. D'autre part, les récompenses accumulées au cours de l'apprentissage sont nécessairement moyennes, ce qui peut être inacceptable lorsque cet apprentissage est en prise avec le système dynamique réel et non simulé.

Inversement, le choix à chaque itération de l'action a_n correspondant à la politique optimale courante (on parle d'action gloutonne[4] et de politique gloutonne[5]), n'est pas non plus satisfaisant, car il conduit généralement soit à une politique sous-optimale, soit à la divergence de l'algorithme.

Ainsi, les algorithmes d'apprentissage par renforcement retiennent un compromis entre exploration et exploitation qui consiste à suivre la politique optimale courante la

4. *Greedy-action.*

5. *Greedy-policy.*

plupart du temps, tout en choisissant plus ou moins régulièrement une action aléatoire pour a_n. Comme le dit élégamment Jozefowiez [JOZ 01], compte tenu de la nécessité de cette exploration plus ou moins aléatoire, la résolution d'un problème d'apprentissage par renforcement reste un art plutôt qu'une science.

Plusieurs méthodes de choix de a_n ont été proposées, que l'on classe en deux catégories dites dirigées ou non dirigées [THR 92].

Les méthodes non dirigées utilisent peu d'informations issues de l'apprentissage autre que la fonction de valeur elle-même. Citons par exemple [BER 96, KAE 98] :

– sur un intervalle de N_1 itérations, suivre la meilleure politique connue, puis sur N_2 itérations, tirer uniformément a_n dans A ;

– les méthodes ϵ-*greedy*, qui consistent à utiliser à chaque itération un tirage semi-uniforme, qui consiste à suivre la meilleure politique connue avec une probabilité $1-\epsilon$, ou à tirer uniformément a_n dans A avec une probabilité ϵ, et $\epsilon \in [0, 1]$;

– les méthodes *softmax*, qui consistent à tirer a_n dans A selon une distribution de Boltzmann, la probabilité associée à l'action a étant :

$$p_T(a) = \frac{\exp(-\frac{Q_n(s_n,a)}{T})}{\sum_{a'} \exp(-\frac{Q_n(s_n,a')}{T})}$$
$$\text{avec } \lim_{n\to\infty} T = 0$$

où $Q_n(s, a)$ représente la fonction de valeur associée à la réalisation de l'action a dans l'état s.

Ces différentes fonctions d'exploration font intervenir des paramètres (N_1, N_2, τ et T) qui contrôlent le degré d'exploration dans le choix de a_n. Par exemple, si $N_1 = 0$, T très grand ou $\tau = 1$, l'algorithme d'apprentissage parcourt uniformément toutes les actions. Les cas utiles en pratique sont pour $N_1 > 0$, $T < +\infty$ et $\tau < 1$, qui assurent expérimentalement une convergence beaucoup plus rapide. La méthode de la roue de la fortune[6] est un cas particulier de méthode *softmax* dans laquelle le facteur T de température est constant au lieu de décroître.

Les méthodes dirigées utilisent pour leur part des heuristiques propres au problème de l'exploration, en se basant sur des informations acquises au cours de l'apprentissage. La plupart de ces méthodes reviennent à ajouter à la valeur $Q(s, a)$ d'une action dans un état un *bonus d'exploration* [MEU 96]. Ce bonus peut être local comme dans la méthode de l'estimation d'intervalle[7] [KAE 93], ou propagé d'état à état au cours de l'apprentissage [MEU 99]. Des définitions simples de ce bonus d'exploration conduisent à des résultats intéressants :

---

6. *Roulette wheel selection.*

7. *Interval estimation.*

– la *recency-based method* : le bonus est égal à $\varepsilon\sqrt{\delta n_{sa}}$ où δn_{sa} représente le nombre d'itérations parcourues depuis la dernière exécution de l'action a dans l'état s, et où ε est une constante inférieure à 1 ;

– la méthode de l'*uncertainty estimation* : le bonus est égal à $\frac{c}{n_{sa}}$, où c est une constante et n_{sa} représente le nombre de fois où l'action a a déjà été choisie dans l'état s.

Ces différentes méthodes d'exploration peuvent être utilisées indifféremment quel que soit l'algorithme d'apprentissage par différence temporelle auquel on fait appel. En effet, pour tous les algorithmes de différence temporelle que nous allons présenter dans la suite, si l'hypothèse de Markov est vérifiée et si l'on parcourt tous les états un nombre infini de fois, alors les valeurs $V(s_t)$ ou les qualités des actions $Q(s_t, a_t)$ convergent vers les valeurs optimales.

Notons qu'au sein d'un algorithme d'apprentissage par renforcement, il est nécessaire de distinguer la politique d'exploration simulée à chaque itération, qui est une politique aléatoire et la meilleure politique courante, qui est markovienne déterministe et qui tend vers une politique optimale π^*.

Enfin, nous verrons au paragraphe 1.5.2 qu'il existe dans le cadre de l'apprentissage par renforcement indirect des méthodes d'exploration récentes dotées d'une preuve de convergence en fonction polynomiale de la taille du problème.

1.2.5. *Des méthodes incrémentales fondées sur une estimation*

Il existe trois classes d'algorithme d'optimisation du comportement :

– les algorithmes de programmation dynamique s'appliquent dans le cas où l'agent dispose d'un modèle de son environnement, c'est-à-dire lorsque les fonctions de transition p et de récompense r sont connues *a priori*. Elles peuvent aussi s'appliquer dans le cas où l'on cherche à apprendre le modèle, donc dans le cadre des méthodes indirectes. Nous y reviendrons au paragraphe 1.5.1 ;

– les méthodes de programmation dynamique présentent l'avantage d'être incrémentales. On peut réaliser des itérations successives qui convergent peu à peu vers la fonction de valeur optimale, ce qui permet d'agir sans attendre de réaliser toutes les itérations. En revanche, elles exigent une connaissance parfaite de la fonction de transition et de la fonction de récompense du MDP associé. Les méthodes de Monte-Carlo présentent les avantages et inconvénients opposés. Elles ne présupposent aucune connaissance *a priori* du problème de décision markovien à résoudre, mais elles ne sont pas incrémentales ;

– les méthodes de différence temporelle reposent sur une estimation incrémentale du modèle de l'environnement. Comme les méthodes de Monte-Carlo, elles réalisent cette estimation sur la base de l'expérience de l'agent et se passent ainsi d'un modèle

du monde. Néanmoins, elles combinent cette estimation avec des mécanismes de propagation locale d'estimation des valeurs tirées de la programmation dynamique, ce qui leur permet de conserver un caractère incrémental.

Les méthodes de différence temporelle, qui constituent le cœur de l'apprentissage par renforcement proprement dit, se caractérisent donc par cette combinaison du recours à l'estimation avec des propriétés d'incrémentalité.

Les méthodes de programmation dynamique ont déjà été présentées au chapitre précédent. Nous consacrons la section suivante aux méthodes de Monte-Carlo, avant de nous tourner à la section 1.4 vers les méthodes de différence temporelle.

1.3. Méthodes de Monte-Carlo

1.3.1. *Préliminaires généraux sur les méthodes d'estimation*

Nous avons vu au chapitre précédent qu'il existait des méthodes de résolution de l'équation d'optimalité de Bellman qui réclamaient de calculer à chaque itération la fonction de valeur associée à la politique courante (algorithmes de *policy iteration*). Ces algorithmes de programmation dynamique nécessitant la connaissance des probabilités de transition et des fonctions de récompense, des méthodes ont été développées permettant d'estimer au mieux la fonction de valeur V^π d'une politique π fixée, sur la base des seules transitions simulées en suivant cette politique. Ces algorithmes d'apprentissage d'une fonction de valeur peuvent alors être utilisés au sein de méthodes directes en apprentissage par renforcement.

Pour des raisons de clarté, nous nous plaçons à présent dans le cas d'un MDP et d'une politique π tels que la chaîne de Markov associée $p(s_{t+1}|s_t) = p(s_{t+1}|s_t, \pi(s_t))$ conduise pour tout état initial vers un état terminal T absorbant de récompense nulle. On considère donc le critère total et on cherche à estimer $V(s) = E(\sum_0^\infty r_t \mid s_0 = s)$ à partir des seules observations (s_t, s_{t+1}, r_t).

Une façon simple de réaliser cette estimation consiste à simuler des trajectoires à partir de chacun des états s jusqu'à l'état terminal T. Si l'on note $R_k(s)$ la somme cumulée obtenue le long de la trajectoire k en suivant la politique π, alors une estimation de la fonction de valeur V en s après $k+1$ trajectoires est donnée par :

$$\forall s \in S, V_{k+1}(s) = \frac{R_1(s) + R_2(s) + ... + R_k(s) + R_{k+1}(s)}{k+1} \tag{1.1}$$

Pour ne pas avoir à stocker chacune des R_k reçues, on montre très simplement qu'un tel calcul peut se reformuler de manière incrémentale :

$$\forall s \in S, V_{k+1}(s) = V_k(s) + \frac{1}{k+1}[R_{k+1}(s) - V_k(s)] \tag{1.2}$$

Pour calculer $V_{k+1}(s)$, il suffit donc de stocker $V_k(s)$ et k. Plutôt que de stocker le nombre k d'expériences réalisées, on utilise généralement une formule d'estimation encore plus générique :

$$\forall s \in S, V_{k+1}(s) = V_k(s) + \alpha[R_{k+1}(s) - V_k(s)] \tag{1.3}$$

qui, pour α bien choisi, vérifie :

$$\lim_{k \to \infty} V_k(s) = V^\pi(s) \tag{1.4}$$

Nous retrouverons cette méthode d'estimation incrémentale dans les méthodes de différence temporelle.

1.3.2. *Les méthodes de Monte-Carlo*

Pour résoudre un problème de planification en utilisant la programmation dynamique dans le cas où l'on ne connaît pas *a priori* les fonctions de transition $p()$ et de récompense $r()$, l'approche indirecte de type *maximum de vraisemblance* qui consiste à estimer les paramètres $p()$ et $r()$ puis à calculer V en résolvant l'équation $V = L_\pi V$ (avec ici $\gamma = 1$) est généralement trop coûteuse, en temps et en espace.

On lui préfère classiquement l'approche dite de *Monte-Carlo*, qui revient à simuler un grand nombre de trajectoires issues de chaque état s de S, et à estimer $V(s)$ en moyennant les coûts observés sur chacune de ces trajectoires. À chaque expérience réalisée, l'agent mémorise les transitions qu'il a effectuées et les récompenses qu'il a reçues. Il met alors à jour une estimation de la valeur des états parcourus en associant à chacun d'eux la part de la récompense reçue qui lui revient. Au fil de ces expériences, la valeur estimée associée à chaque état converge alors vers la valeur exacte de l'état pour la politique qu'il suit.

L'apport principal des méthodes de Monte-Carlo réside donc dans la technique qui permet d'estimer la valeur d'un état sur la base de la réception de plusieurs valeurs successives de récompense cumulée associées à cet état lors de trajectoires distinctes. On s'appuie alors sur la méthode d'estimation présentée au paragraphe 1.3.1.

Soit ainsi $(s_0, s_1, \ldots, s_N)$ une trajectoire générée en suivant la probabilité de transition inconnue $p()$, et $(r_0, r_1, \ldots, r_{N-1})$ les récompenses observées au cours de cette trajectoire (s_N est l'état terminal T de récompense nulle).

Le principe de la méthode de Monte-Carlo est de mettre à jour les N valeurs $V(s_k)$, $k = 0, \ldots, N-1$, selon :

$$V(s_k) \leftarrow V(s_k) + \alpha(s_k)(r_k + r_{k+1} + \cdots + r_{N-1} - V(s_k)) \tag{1.5}$$

avec les taux d'apprentissage $\alpha(s_k)$ tendant vers 0 au cours des itérations. La convergence presque sûre de cet algorithme vers la fonction V est assurée sous des hypothèses générales [BER 96].

Cette méthode est qualifiée d'*every-visit* car la valeur d'un état peut être mise à jour plusieurs fois le long d'une même trajectoire. Les termes d'erreur associés à chacune de ces mises à jour ne sont alors pas indépendants, entraînant un biais non nul dans l'estimation de la fonction V sur la base d'un nombre fini de trajectoires [BER 96, page 190]. Une solution simple à ce problème de biais consiste alors à ne mettre à jour la valeur $V(s)$ d'un état que lors de sa première rencontre le long de la trajectoire observée. Cela définit la méthode dite de *first-visit*, qui conduit à un estimateur non biaisé de la fonction V. Expérimentalement, l'erreur quadratique moyenne de la méthode *first-visit* tend à être inférieure à celle de la méthode *every-visit* [SIN 96].

Les méthodes de Monte-Carlo permettent donc d'estimer la fonction de valeur d'une politique π en mettant à jour certaines de ses composantes à la fin de chaque trajectoire observée. Ces méthodes exigent pour fonctionner qu'un grand nombre de contraintes soient remplies. En particulier, il est indispensable que l'apprentissage soit décomposé en une succession d'épisodes de longueur finie, faute de quoi la mise à jour de l'estimation de la valeur des états ne peut pas avoir lieu. Le fait qu'il faille attendre la fin d'une expérience pour apprendre quoi que ce soit justifie l'affirmation selon laquelle ces méthodes ne sont pas incrémentales.

Il est toutefois possible d'améliorer ces algorithmes en autorisant la mise à jour de la fonction de valeur non plus à la fin de chaque trajectoire, mais à la suite de chaque transition du système. Nous développons ici cette réécriture, dont le principe est à la base des méthodes de différence temporelle.

La règle de mise à jour (1.5) de la fonction V peut être réécrite de la manière suivante (le terme γ vaut 1, il est introduit dans les équations pour faire ressortir les relations avec l'erreur de différence temporelle utilisée dans les méthodes de différence temporelle ; par ailleurs, nous utilisons la propriété $V(s_N) = V(T) = 0$) :

$$\begin{aligned} V(s_k) \leftarrow V(s_k) + \alpha(s_k) \quad & \Big((r_k + \gamma V(s_{k+1}) - V(s_k)) \\ & + (r_{k+1} + \gamma V(s_{k+2}) - V(s_{k+1})) \\ & + \cdots \\ & + (r_{N-1} + \gamma V(s_N) - V(s_{N-1})) \Big) \end{aligned}$$

soit encore :

$$V(s_k) \leftarrow V(s_k) + \alpha(s_k)(\delta_k + \delta_{k+1} + \cdots + \delta_{N-1}) \tag{1.6}$$

en définissant la différence temporelle δ_k par :

$$\delta_k = r_k + \gamma V(s_{k+1}) - V(s_k), \quad k = 0, \ldots, N-1.$$

Le terme δ_k[8] est appelé « erreur de différence temporelle[9] » [SUT 88].

Cette erreur δ_k peut être interprétée en chaque état comme une mesure de la différence entre l'estimation courante $V(s_k)$ et l'estimation corrigée à un coup $r_k + V(s_{k+1})$. Son calcul est possible dès que la transition (s_k, s_{k+1}, r_k) a été observée, et cela conduit donc à une version « *on-line* » de la règle de mise à jour (1.6) où il n'est plus nécessaire d'attendre la fin de la trajectoire pour commencer à modifier les valeurs de V :

$$V(s_l) \leftarrow V(s_l) + \alpha(s_l)\delta_k, \quad l = 0, \ldots, k \tag{1.7}$$

dès que la transition (s_k, s_{k+1}, r_k) est simulée et l'erreur δ_k calculée. Selon qu'une trajectoire peut parcourir plusieurs fois le même état ou non, cette version « *on-line* » peut légèrement différer de l'algorithme original (1.6). Toutefois sa convergence presque sûre vers V reste valide. Là encore, une approche de type *first-visit* semble préférable en pratique [SIN 96].

1.4. Les méthodes de différence temporelle

Nous allons nous tourner à présent vers les méthodes de différence temporelle, qui combinent l'incrémentalité de la programmation dynamique avec le recours à l'expérience des méthodes de Monte-Carlo.

1.4.1. *L'algorithme TD(0)*

Nous avons vu au chapitre précédent qu'il existait plusieurs critères possibles pour représenter la performance que l'agent doit maximiser sur le long terme. Historiquement, le critère qui a donné lieu aux développements les plus importants est le critère dit « γ-*pondéré* » ou actualisé. Tous les algorithmes que nous allons présenter dans cette section, qui sont les algorithmes les plus classiques de l'apprentissage par renforcement, s'appliquent dans le cadre de ce critère.

L'algorithme élémentaire d'apprentissage par renforcement, dit algorithme de « différence temporelle », s'appelle TD[10]. Nous le notons ici TD(0) pour des raisons qui apparaîtront quand nous présenterons les traces d'éligibilité. Cet algorithme repose sur une comparaison entre la récompense que l'on reçoit effectivement et la récompense que l'on s'attend à recevoir en fonction des estimations construites précédemment.

---

8. On trouve parfois $\delta_k = r_{k+1} + \gamma V(s_{k+1}) - V(s_k), \quad k = 0, \ldots, N-1$, si l'on note r_{k+1} plutôt que r_k la récompense reçue lors du passage de l'état s_k à l'état s_{k+1}.

9. *Temporal difference error.*

10. *Temporal Difference.*

Si les estimations des fonctions de valeur aux états s_t et s_{t+1}, notées $V(s_t)$ et $V(s_{t+1})$, étaient exactes, on aurait :

$$V(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + ... \quad (1.8)$$

$$V(s_{t+1}) = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... \quad (1.9)$$

Donc on aurait :

$$V(s_t) = r_t + \gamma V(s_{t+1}) \quad (1.10)$$

On voit que l'erreur de différence temporelle δ_k mesure l'erreur entre les valeurs effectives des estimations $V(s)$ et les valeurs qu'elles devraient avoir.

La méthode de différence temporelle consiste à corriger peu à peu cette erreur en modifiant la valeur de $V(s_t)$ selon une équation de type Widrow-Hoff, que l'on utilise dans le domaine des réseaux de neurones :

$$V(s_t) \leftarrow V(s_t) + \alpha[r_t + \gamma V(s_{t+1}) - V(s_t)] \quad (1.11)$$

Cette équation de mise à jour permet de comprendre immédiatement comment les algorithmes de différence temporelle combinent les propriétés de la programmation dynamique avec celles des méthodes de Monte-Carlo. En effet, elle fait apparaître les deux caractéristiques suivantes :

– comme dans les algorithmes de programmation dynamique, la valeur estimée de $V(s_t)$ est mise à jour en fonction de la valeur estimée de $V(s_{t+1})$. Il y a donc propagation de la valeur estimée à l'état courant à partir des valeurs estimées des états successeurs ;

– comme dans les méthodes de Monte-Carlo, chacune de ces valeurs résulte d'une estimation locale des récompenses immédiates qui repose sur l'expérience accumulée par l'agent au fil de ses interactions avec son environnement.

On voit donc que les méthodes de différence temporelle et, en particulier, TD(0), reposent sur deux processus de convergence couplés, le premier estimant de plus en plus précisément la récompense immédiate reçue dans chacun des états et le second approchant de mieux en mieux la fonction de valeur résultant de ces estimations en les propageant de proche en proche.

Dans le cas de TD(0), les mises à jour se font localement à chaque fois que l'agent réalise une transition dans son environnement, à partir d'une information se limitant

à son état courant s_t, l'état successeur s_{t+1} et la récompense r_t reçue suite à cette transition. Une preuve de convergence de l'algorithme a été proposée par Dayan et Sejnowski [DAY 94].

En revanche, il faut noter que, comme TD(0) estime la fonction de valeur de chacun des états d'un problème, faute d'un modèle des transitions entre les états, l'agent est incapable d'en déduire quelle politique suivre, car il ne peut réaliser un pas de regard en avant pour déterminer quelle est l'action qui lui permettra de rejoindre l'état suivant de plus grande valeur. Ce point explique que l'on préfère avoir recours aux algorithmes qui travaillent sur une fonction de valeur associée aux couples état-action plutôt qu'à l'état seul. Ce sont ces algorithmes que nous allons présenter dans ce qui suit. Nous verrons ensuite l'approche alternative proposée par les architectures acteur-critique, consistant à travailler directement sur une politique que l'on cherche à améliorer itérativement au fil de l'expérience.

1.4.2. *L'algorithme Sarsa*

Comme nous venons de l'expliquer, la forme de l'équation de Bellman $V = LV$ n'est pas satisfaisante pour en dériver directement un algorithme adaptatif de résolution. Pour cela, Watkins [WAT 89] a introduit la fonction de valeur Q, dont la donnée est équivalente à celle de V lorsque l'on connaît la fonction de transition p.

DÉFINITION 1.1.– *Fonction de valeur Q*
A une politique π fixée de fonction de valeur V^π, on associe la nouvelle fonction :

$$\forall s \in S, a \in A \quad Q^\pi(s,a) = r(s,a) + \gamma \sum_{s'} p(s'|s,a) V^\pi(s').$$

L'interprétation de la valeur $Q^\pi(s,a)$ est la suivante : c'est la valeur espérée du critère pour le processus partant de s, exécutant l'action a, puis suivant la politique π par la suite. Il est clair que $V^\pi(x) = Q^\pi(x, \pi(x))$, et l'équation de Bellman vérifiée par la fonction Q^* devient :

$$\forall s \in S, a \in A \quad Q^*(s,a) = r(s,a) + \gamma \sum_{s'} p(s'|s,a) \max_b Q^*(s',b).$$

On a alors :

$$\begin{aligned} \forall s \in S \quad V^*(s) &= \max_a Q^*(s,a), \\ \pi^*(s) &= \operatorname*{argmax}_a Q^*(s,a). \end{aligned}$$

L'algorithme SARSA est similaire à l'algorithme TD(0) à ceci près qu'il travaille sur les valeurs des couples (s, a) plutôt que sur la valeur des états. Son équation de mise à jour est identique à celle de TD(0) en remplaçant la fonction de valeur par la fonction de valeur d'action :

$$Q(s_n, a_n) \leftarrow Q(s_n, a_n) + \alpha[r_n + \gamma Q(s_{n+1}, a_{n+1}) - Q(s_n, a_n)]. \quad (1.12)$$

L'information nécessaire pour réaliser une telle mise à jour alors que l'agent réalise une transition est le quintuplet $(s_n, a_n, r_n, s_{n+1}, a_{n+1})$, d'où découle le nom de l'algorithme.

Effectuer ces mises à jour implique que l'agent détermine avec un pas de regard en avant quelle est l'action a_{n+1} qu'il réalisera lors du pas de temps suivant, lorsque l'action a_n dans l'état s_n l'aura conduit dans l'état s_{n+1}.

Il résulte de cette implication une dépendance étroite entre la question de l'apprentissage et la question de la détermination de la politique optimale. Dans un tel cadre, il n'existe qu'une seule politique qui doit prendre en compte à la fois les préoccupations d'exploration et d'exploitation et l'agent est contraint de réaliser cet apprentissage uniquement sur la base de la politique qu'il suit effectivement. On dit d'un algorithme tel que SARSA qu'il est *on-policy*. La dépendance que cela induit entre l'exploration et l'apprentissage complique considérablement la mise au point de preuves de convergences pour ces algorithmes, ce qui explique que de telles preuves de convergence soient apparues beaucoup plus tard [SIN 00] que pour les algorithmes dits *off-policy* tels que Q-learning, que nous allons voir à présent.

1.4.3. *L'algorithme Q-learning*

L'algorithme Q-learning se présente comme une simplification de l'algorithme SARSA par le fait qu'il n'est plus nécessaire pour l'appliquer de déterminer un pas de temps à l'avance quelle sera l'action réalisée au pas de temps suivant.

Son équation de mise à jour est la suivante :

$$Q(s_n, a_n) \leftarrow Q(s_n, a_n) + \alpha[r_n + \gamma \max_a Q(s_{n+1}, a) - Q(s_n, a_n)]. \quad (1.13)$$

La différence essentielle entre SARSA et Q-learning se situe au niveau de la définition du terme d'erreur. Le terme $Q(s_{n+1}, a_{n+1})$ apparaissant dans l'équation (1.12) a été remplacé par le terme $max_a Q(s_{n+1}, a)$ dans l'équation (1.13). Cela pourrait sembler équivalent si la politique suivie était gloutonne (on aurait alors $a_{n+1} = \arg\max_a Q(s_{n+1}, a)$). Toutefois, compte tenu de la nécessité de réaliser un compromis entre exploration et exploitation, ce n'est généralement pas le cas. Il apparaît donc

que l'algorithme SARSA effectue les mises à jour en fonction des actions choisies effectivement alors que l'algorithme Q-learning effectue les mises à jour en fonction des actions optimales mêmes si ce ne sont pas ces actions optimales que l'agent réalise, ce qui est plus simple.

Cette simplicité a fait la réputation de l'algorithme Q-learning. Il s'agit sans doute de l'algorithme d'apprentissage par renforcement le plus connu et le plus utilisé en raison des preuves formelles de convergence qui ont accompagné sa publication [WAT 92].

Algorithme 1.1 : Le Q-learning

```
/* α_n est un taux d'apprentissage                                  */
Initialiser(Q_0)
pour n ← 0 jusqu'à N_tot − 1 faire
    s_n ← ChoixEtat
    a_n ← ChoixAction
    (s'_n, r_n) ← Simuler(s_n, a_n)
    { mise à jour de Q_n :}
    début
        Q_{n+1} ← Q_n
        δ_n ← r_n + γ max_b Q_n(s'_n, b) − Q_n(s_n, a_n)
        Q_{n+1}(s_n, a_n) ← Q_n(s_n, a_n) + α_n(s_n, a_n)δ_n
    fin
retourner Q_{N_tot}
```

Le principe de l'algorithme Q-learning, défini formellement par l'algorithme 1.1, est de mettre à jour itérativement, à la suite de chaque transition (s_n, a_n, s_{n+1}, r_n), la fonction de valeur courante Q_n pour le couple (s_n, a_n), où s_n représente l'état courant, a_n l'action sélectionnée et réalisée, s'_n l'état résultant et r_n la récompense immédiate. Cette mise à jour se fait sur la base de l'observation des transitions instantanées et de leur récompense associée.

Dans cet algorithme, N_{tot} est un paramètre initial fixant le nombre d'itérations. Le taux d'apprentissage $\alpha_n(s, a)$ est propre à chaque couple (s, a), et décroît vers 0 à chaque passage. La fonction `Simuler` retourne un nouvel état et la récompense associée selon la dynamique du système. Le choix de l'état courant et de l'action à exécuter est effectué par les fonctions `ChoixEtat` et `ChoixAction` et sera discuté plus loin. La fonction `Initialiser` revient la plupart du temps à initialiser les composantes de Q_0 à 0, mais il existe des initialisations plus efficaces.

Il est immédiat d'observer que l'algorithme Q-learning est une formulation stochastique de l'algorithme de *value iteration* vu au chapitre précédent pour les MDP.

En effet, ce dernier peut s'exprimer directement en termes de fonction de valeur d'action :

$$
\begin{aligned}
& V_{n+1}(s) = \max_{a \in A} \overbrace{\left\{ r(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) V_n(s') \right\}}^{Q_n(s,a)} \\
\Rightarrow \quad & Q_{n+1}(s,a) = r(s,a) + \gamma \textstyle\sum_{s' \in S} p(s'|s,a) V_{n+1}(s') \\
\Rightarrow \quad & Q_{n+1}(s,a) = r(s,a) + \gamma \textstyle\sum_{s' \in S} p(s'|s,a) \max_{a' \in A} Q_n(s',a').
\end{aligned}
$$

Le Q-learning est alors obtenu en remplaçant le terme $r(s,a) + \sum_{s'} p(s'|s,a) \max_{a' \in A} Q_n(s',a')$ par son estimateur sans biais le plus simple construit à partir de la transitition courante $r_n + \max_{a'} Q_n(s'_n, a')$.

La convergence de cet algorithme est établie [WAT 89, JAA 94] (la fonction Q_n converge presque sûrement vers Q^*) sous les hypothèses suivantes :

– finitude de S et A ;

– chaque couple (s,a) est visité un nombre infini de fois ;

– $\sum_n \alpha_n(s,a) = \infty$ et $\sum_n \alpha_n^2(s,a) < \infty$;

– $\gamma < 1$ ou si $\gamma = 1$;

– pour toute politique, il existe un état absorbant de récompense nulle.

Rappelons que cette convergence presque sûre signifie que $\forall s, a$ la suite $Q_n(s,a)$ converge vers $Q^*(s,a)$ avec une probabilité égale à 1. En pratique, la suite $\alpha_n(s,a)$ est souvent définie comme $\alpha_n(s,a) = \frac{1}{n_{sa}}$.

1.4.4. *Les algorithmes TD(λ), Sarsa(λ) et Q(λ)*

Les algorithmes TD(0), SARSA et Q-learning présentent le défaut de ne mettre à jour qu'une valeur par pas de temps, à savoir la valeur de l'état que l'agent est en train de visiter. Comme il apparaît sur la figure 1.1, cette procédure de mise à jour est particulièrement lente. En effet, pour un agent ne disposant d'aucune information *a priori* sur la structure de la fonction de valeur, il faut au moins n expériences successives pour que la récompense immédiate reçue dans un état donné soit propagée jusqu'à un état distant du premier de n transitions. En attendant le résultat de cette propagation, tant que toutes les valeurs sont identiquement nulles, le comportement de l'agent est une marche aléatoire.

Figure 1.1. *Q-learning : premier et deuxième essai. On observe que, toutes les valeurs étant initialement nulles, la propagation de valeurs non nulles ne se fait qu'une fois que l'agent a trouvé une première fois la source de récompense et ne progresse que d'un pas à chaque essai de l'agent.*

Une façon d'améliorer cet état de fait consiste à doter l'algorithme d'une mémoire des transitions effectuées au cours d'une expérience afin d'effectuer toutes les propagations possibles à la fin de cette expérience. Cette mémoire des transitions effectuées précédemment est appelée une *trace d'éligibilité*. Ainsi, Sutton et Barto [SUT 98] ont proposé une classe d'algorithmes appelés « TD(λ) » qui généralisent l'algorithme TD(0) au cas où l'agent dispose d'une mémoire des transitions. Plus tard, les algorithmes SARSA et Q-learning ont été généralisés en SARSA (λ) et Q(λ), le second l'ayant été de deux façons différentes par deux auteurs différents [WAT 92, PEN 96].

Un premier procédé naïf pour accélérer l'apprentissage consiste à stocker directement une liste des couples état-action parcourus par l'agent puis, à chaque fois que l'agent reçoit une récompense, propager celle-ci en parcourant la mémoire des transitions en marche arrière depuis la récompense reçue. Avec un tel procédé, plus la mémoire des transitions est longue, plus une récompense reçue est propagée efficacement. Il apparaît donc un compromis entre la quantité de mémoire mobilisée pour apprendre et la vitesse d'apprentissage. Mais une telle méthode ne fonctionne pas sur un horizon infini.

La méthode mise en œuvre dans TD(λ), SARSA (λ) et Q(λ) est plus sophistiquée et fonctionne pour des horizons infinis. Nous discuterons à la section 1.5.1, une autre solution qui permet d'effectuer plusieurs mises à jour à chaque pas de temps, dans le cadre de l'apprentissage par renforcement indirect. Nous commençons par examiner de plus près les algorithmes TD(λ), SARSA (λ) et $Q(\lambda)$.

1.4.5. *Les architectures acteur-critique*

Historiquement, les architectures acteur-critique ont été les premières architectures informatiques imaginées par des chercheurs pour modéliser l'apprentissage par renforcement [WIT 77, BAR 83].

Nous avons dit que, dans TD(0), on met à jour la fonction de valeur en se fondant sur l'erreur de différence temporelle $\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ mais, si on ne dispose pas d'un modèle de la fonction de transition, on ne sait pas comment exploiter la fonction de valeur pour choisir une action. La solution proposée par les architectures acteur-critique consiste à tenir à jour en parallèle une structure représentant la fonction de valeur, appelée « le critique », et une structure représentant la politique, appelée « l'acteur ».

Le critique est nécessaire pour calculer la valeur de δ_t, qui dépend de la fonction de valeur. Mais le terme δ_t est aussi utilisé pour mettre à jour la politique. Si ce terme est positif, l'action réalisée mérite d'être renforcée. S'il est négatif, au contraire, il faut diminuer la tendance de l'acteur à réaliser cette action.

Cette spécification est très générale, elle impose très peu de contraintes sur les formalismes de représentation respectifs de l'acteur et du critique. La seule contrainte forte est que l'acteur soit capable de prendre en compte le signal d'erreur δ_t pour modifier sa propension à réaliser telle ou telle action dans un contexte donné. Quant au critique, tout formalisme capable de fournir une approximation de la fonction de valeur fait l'affaire.

Cependant, un grand nombre d'architectures acteur-critique s'appuient sur des réseaux de neurones formels, en raison de la relative plausibilité biologique de cette architecture. Le terme δ_t vient alors modifier le poids des connexions qui gouvernent la propension d'un réseau à réaliser telle ou telle action.

1.4.6. *Différences temporelles avec traces d'éligibilité : TD(λ)*

L'originalité de TD(λ) est de proposer un compromis entre les deux équations (1.6) et (1.7) présentées dans le cadre de l'algorithme de Monte-Carlo itératif. Soit donc $\lambda \in [0, 1]$ un paramètre de pondération. Avec les mêmes notations que précédemment, l'algorithme TD(λ) défini par Sutton [SUT 88] est le suivant :

$$V(s_k) \leftarrow V(s_k) + \alpha(s_k) \sum_{m=k}^{m=N-1} \lambda^{m-k} \delta_m, \quad k = 0, \ldots, N-1. \tag{1.14}$$

On peut essayer de mieux comprendre le rôle du coefficient λ en réécrivant l'équation (1.14) sous la forme :

$$V(s_k) \leftarrow V(s_k) + \alpha(s_k)(z_k^\lambda - V(s_k)).$$

On a alors :

$$\begin{aligned} z_k^\lambda &= V(s_k) + \sum_{m=k}^{m=N-1} \lambda^{m-k}\delta_m \\ &= V(s_k) + \delta_k + \lambda \sum_{m=k+1}^{m=N-1} \lambda^{m-k-1}\delta_m \\ &= V(s_k) + \delta_k + \lambda(z_{k+1}^\lambda - V(s_{k+1})) \\ &= V(s_k) + r_k + V(s_{k+1}) - V(s_k) + \lambda(z_{k+1}^\lambda - V(s_{k+1})) \\ &= r_k + (\lambda z_{k+1}^\lambda + (1-\lambda)V(s_{k+1})). \end{aligned}$$

Dans le cas où $\lambda = 0$, il est clair que cela revient à ne considérer qu'un horizon unitaire, comme dans le cadre de la programmation dynamique. On retrouve donc TD(0).

Si $\lambda = 1$, l'équation (1.14) se réécrit :

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

ce qui est exactement l'équation (1.5) de la méthode de Monte-Carlo.

Pour tout λ, les deux approches de type *first-visit* ou *every-visit* peuvent être considérées. De même, une version *on-line* de l'algorithme d'apprentissage TD(λ) décrit par l'équation (1.14) est possible :

$$V(s_l) \leftarrow V(s_l) + \alpha(s_l)\lambda^{k-l}\delta_k, \quad l = 0, \ldots, k \tag{1.15}$$

dès que la transition (s_k, s_{k+1}, r_k) est simulée et l'erreur δ_k calculée.

L'application du TD(λ) pour l'évaluation d'une politique π selon le critère γ-pondéré entraîne certaines modifications des algorithmes standards (1.14) ou (1.15), qu'il est nécessaire de citer ici.

Un calcul en tout point semblable au cas $\gamma = 1$ conduit à une règle du type :

$$V(s_k) \leftarrow V(s_k) + \alpha(s_k) \sum_{m=k}^{m=\infty} (\gamma\lambda)^{m-k}\delta_m. \tag{1.16}$$

Il est alors clair que l'absence potentielle d'états finaux absorbants rend inadéquate un algorithme de type *off-line* ne mettant à jour la fonction de valeur V qu'à la fin de la trajectoire, car celle-ci peut être de taille infinie. On définit donc une version *on-line* de (1.16), qui prend la forme suivante :

$$V(s) \leftarrow V(s) + \alpha(s) z_n(s) \delta_n, \quad \forall s \in S, \tag{1.17}$$

dès que la $n$ième transition (s_n, s_{n+1}, r_n) a été simulée et l'erreur δ_n calculée. Le terme $z_n(s)$, dénommé trace d'éligibilité [11] se définit ainsi dans la version la plus proche de l'algorithme TD(λ) original.

DÉFINITION 1.2.– *Trace d'éligibilité accumulative*

$$\begin{array}{lll} z_0(s) & = 0, \quad \forall s \in S, & \\ z_n(s) & = \left\{ \begin{array}{ll} \gamma\lambda z_{n-1}(s) & \textit{si } s \neq s_n, \\ \gamma\lambda z_{n-1}(s) + 1 & \textit{si } s = s_n. \end{array} \right. \end{array}$$

Ce coefficient d'éligibilité augmente donc sa valeur à chaque nouveau passage dans l'état associé, puis décroît exponentiellement au cours des itérations suivantes, jusqu'à un nouveau passage dans cet état (voir figure 1.2).

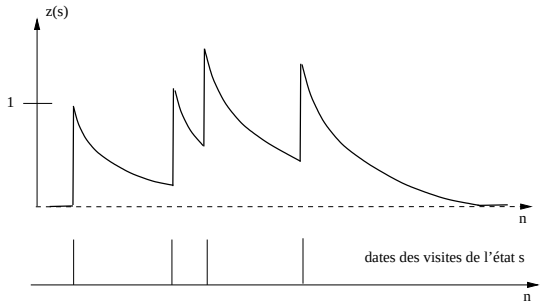

Figure 1.2. *Trace d'éligibilité cumulative : à chaque visite, on ajoute 1 à la valeur précédente, si bien que la valeur de la trace peut dépasser 1*

Dans certains cas, une définition légèrement différente de la trace $z_n(s)$ semble conduire à une convergence plus rapide de la fonction de valeur V.

11. *Eligibility trace*, ou encore *activity*.

DÉFINITION 1.3.– *Trace d'éligibilité avec réinitialisation*

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

La valeur de la trace est donc saturée à 1, comme le montre la figure 1.3.

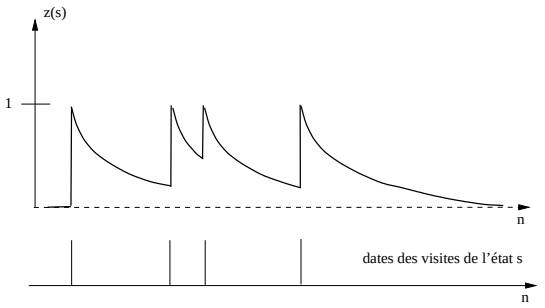

Figure 1.3. *Trace d'éligibilité avec réinitialisation : on remet la valeur à 1 à chaque visite*

La convergence presque sûre de l'algorithme TD(λ) a été montrée pour toute valeur de λ, en *on-line* ou *off-line*, sous les hypothèses classiques de visite en nombre infini de chaque état $s \in S$, et décroissance des α vers 0 à chaque itération n, telle que $\sum_n \alpha_n(s) = \infty$ et $\sum_n \alpha_n^2(s) < \infty$ [JAA 94, BER 96].

Il est à noter que l'effet du λ est encore mal compris et sa détermination optimale pour un problème donné reste très empirique.

Une implémentation directe de TD(λ) basée sur la trace d'éligibilité n'est bien sûr pas efficace dès que la taille de l'espace d'état S devient trop grande. Une première solution approchée [SUT 98] consiste à forcer à 0 la valeur de toutes les traces $z_n(s) < \varepsilon$, et donc à ne maintenir que les traces des états récemment visités (plus précisément, on cesse de maintenir un état dont la dernière visite remonte à plus de $\frac{\log(\varepsilon)}{\log(\gamma\lambda)}$ transitions).

Une autre méthode approchée [CIC 95] connue sous le nom de *truncated temporal differences*, ou TTD(λ), revient à gérer un horizon glissant de taille m mémorisant les derniers états visités et à mettre à jour sur cette base à chaque itération n la valeur de l'état visité à l'itération $(n - m)$.

1.4.7. *De TD(λ) à Sarsa(λ)*

TD(λ) peut être appliqué au problème de l'apprentissage par renforcement pour apprendre une politique optimale. Pour cela, une première approche consiste à coupler TD(λ) à un algorithme gérant l'évolution d'une suite de politiques π_n. En effet, contrairement au Q-learning qui voit la suite Q_n converger vers Q^* sans nécessiter la présence en parallèle d'une suite de politiques π_n, l'algorithme TD(λ) ne sait qu'apprendre la fonction de valeur d'une politique fixée. Dans Q-learning, une telle suite de politiques existe à travers π_{Q_n}, mais l'intérêt du Q-learning est justement que cette suite n'est qu'implicite. On retrouve donc ici un type d'opposition rencontrée au chapitre précédent en programmation dynamique entre *value iteration* et *policy iteration*.

Toutefois, il s'avère que l'algorithme Q-learning intègre directement l'idée maîtresse de TD(λ) de considérer une erreur de différence temporelle. Si l'on reprend la règle de mise à jour du Q-learning :

$$Q_{n+1}(s_n, a_n) = Q_n(s_n, a_n) + \alpha_n\{r_n + \gamma V_n(s'_n) - Q_n(s_n, a_n)\}$$

pour la transition observée (s_n, a_n, s'_n, r_n), et dans le cas où l'action a_n exécutée dans l'état s_n est l'action optimale pour Q_n, c'est-à-dire pour $a_n = \pi_{Q_n}(s_n) = \mathrm{argmax}_b\, Q_n(s_n, b)$, on constate que le terme d'erreur employé est égal à :

$$r_n + \gamma V_n(s'_n) - V_n(s_n)$$

qui est exactement celui de TD(0). Cela peut alors se généraliser à $\lambda > 0$, au travers d'un couplage entre les méthodes TD(λ) et Q-learning.

L'algorithme SARSA (λ) [RUM 94] en est une première illustration. Cet algorithme 1.2 reprend directement l'équation (1.17) en l'adaptant à une représentation par fonction de valeur d'action.

La trace d'éligibilité $z_n(s, a)$ est étendue aux couples (s, a) et l'exploration de l'espace d'états est guidée par la dynamique (sauf lors de la rencontre avec un état terminal).

1.4.7.1. $Q(\lambda)$

La prise en compte des cas où l'action optimale $\pi_{Q_n}(s'_n)$ n'a pas été sélectionnée conduit aux algorithmes $Q(\lambda)$ proposés par Watkins (voir [SUT 98]) et Peng [PEN 94]. La caractéristique du $Q(\lambda)$ de Watkins est de ne considérer un $\lambda > 0$ que le long des segments de trajectoires où la politique courante π_{Q_n} a été suivie. Les deux modifications relativement à SARSA(λ) concernent donc les règles de mise à jour de Q_n et de z_n, comme cela apparaît dans l'algorithme 1.3.

L'inconvénient de cette approche est que, pour des politiques d'apprentissage très exploratrices, les traces z_n sont très fréquemment remises à 0 et le comportement

Algorithme 1.2 : SARSA(λ)

```
/* α_n est un taux d'apprentissage                                  */
Initialiser(Q_0)
z_0 ← 0
s_0 ← ChoixEtat
a_0 ← ChoixAction
pour n ← 0 jusqu'à N_tot − 1 faire
    (s'_n, r_n) ← Simuler(s_n, a_n)
    a'_n ← ChoixAction
    { mise à jour de Q_n et z_n :}
    début
        δ_n ← r_n + γQ_n(s'_n, a'_n) − Q_n(s_n, a_n)
        z_n(s_n, a_n) ← z_n(s_n, a_n) + 1
        pour s ∈ S, a ∈ A faire
            Q_{n+1}(s, a) ← Q_n(s, a) + α_n(s, a)z_n(s, a)δ_n
            z_{n+1}(s, a) ← γλz_n(s, a)
    fin
    si s'_n non absorbant alors s_{n+1} ← s'_n et a_{n+1} ← a'_n
    sinon
        s_{n+1} ← ChoixEtat
        a_{n+1} ← ChoixAction
retourner Q_{N_tot}
```

de Q(λ) est alors assez proche du Q-learning original. Le $Q(\lambda)$ de Peng est une réponse à ce problème. Il est aussi possible d'imaginer une application directe de TD(λ) au Q-learning en ne remettant pas la trace z_n à 0 lors du choix d'une action non optimale. Il existe peu de résultats expérimentaux présentant cette approche (voir toutefois [NDI 99]) ni de comparaisons entre TD(λ), SARSA(λ) et Q(λ) autorisant de tirer des conclusions définitives sur le sujet. La seule véritable certitude issue de nombreuses applications est que la prise en compte des traces d'éligibilité avec $\lambda > 0$ accélère la convergence de l'apprentissage en termes de nombre d'itérations. L'analyse en termes de temps de calcul est plus complexe, car les algorithmes prenant en compte une trace nécessitent beaucoup plus de calculs à chaque itération.

1.4.8. *L'algorithme R-learning*

Tous les algorithmes que nous avons présentés jusqu'à présent s'appliquaient dans le cas du critère γ-pondéré. L'algorithme R-learning, proposé par Schwartz [SCH 93], est l'adaptation au critère moyen de l'algorithme Q-learning et l'on y retrouve tous les principes évoqués précédemment.

Algorithme 1.3 : Q(λ)

```
/* α_n est un taux d'apprentissage                                  */
Initialiser(Q_0)
z_0 ← 0
s_0 ← ChoixEtat
a_0 ← ChoixAction
pour n ← 0 jusqu'à N_tot − 1 faire
    (s'_n, r_n) ← Simuler(s_n, a_n)
    a'_n ← ChoixAction
    { mise à jour de Q_n et z_n :}
    début
        δ_n ← r_n + γ max_b Q_n(s'_n, b) − Q_n(s_n, a_n)
        z_n(s_n, a_n) ← z_n(s_n, a_n) + 1
        pour s ∈ S, a ∈ A faire
            Q_{n+1}(s, a) ← Q_n(s, a) + α_n(s, a) z_n(s, a) δ_n
            z_{n+1}(s, a) ← { 0                 si a'_n ≠ π_{Q_n}(s'_n)
                            { γλ z_n(s, a)      si a'_n = π_{Q_n}(s'_n)
    fin
    si s'_n non absorbant alors s_{n+1} ← s'_n et a_{n+1} ← a'_n
    sinon
        s_{n+1} ← ChoixEtat
        a_{n+1} ← ChoixAction
retourner Q_{N_tot}
```

L'objectif de cet algorithme est de construire une politique dont la récompense moyenne ρ_π est la plus proche possible de la récompense moyenne maximale ρ^* d'une politique optimale π^*. Pour cela, le R-learning maintient deux suites entrecroisées ρ_n et R_n. Notons ici que la suite ρ_n n'est mise à jour que lorsque l'action qui vient d'être exécutée était la *greedy-action* maximisant R_n dans l'état courant s_n. La suite réelle des ρ_n est une estimation du critère à optimiser. Comme Q_n dans le Q-learning, R_n représente une forme particulière de la fonction de valeur relative U d'une politique :

DÉFINITION 1.4.– *Fonction de valeur R*
À une politique π fixée de fonction de valeur U^π et de gain moyen ρ_π, on associe la nouvelle fonction :

$$\forall s \in S, a \in A \quad R^\pi(s,a) = r(s,a) - \rho_\pi + \sum_{s'} p(s'|s,a) U^\pi(s').$$

On a donc là encore $U^\pi(x) = R^\pi(x, \pi(x))$ et l'équation de Bellman vérifiée par ρ^* et R^* devient :

$$\forall s \in S, a \in A \quad R^*(s,a) = r(s,a) - \rho^* + \sum_{s'} p(s'|s,a) \max_b R^*(s',b) \qquad (1.18)$$

avec l'assurance que la politique $\pi_{R^*}(s) = \text{argmax}_a\, R^*(s,a)$ a pour gain moyen le gain optimal ρ^*.

Comme pour Q-learning, l'algorithme R-learning est une version stochastique de la méthode d'itération sur les valeurs pour l'équation (1.18).

Algorithme 1.4 : Le R-learning

```
/* αn et βn sont des taux d'apprentissage                    */
Initialiser(R0,ρ0)
pour n ← 0 jusqu'à Ntot − 1 faire
    sn ← ChoixEtat
    an ← ChoixAction
    (s'n, rn) ← Simuler(sn, an)
    { mise à jour de Rn et ρn :}
    début
        Rn+1 ← Rn
        δn ← rn − ρn + max_b Rn(s'n, b) − Rn(sn, an)
        Rn+1(sn, an) ← Rn(sn, an) + αn(sn, an)δn
        ρn+1 ← { ρn            si an ≠ π_Rn(sn)
               { ρn + βnδn     si an = π_Rn(sn)
    fin
retourner R_Ntot, ρ_Ntot
```

Bien qu'il n'existe pas de preuve formelle de convergence du R-learning vers une politique gain-optimale, de nombreuses expérimentations montrent que ρ_n approche efficacement ρ^*, avec des taux d'apprentissage $\alpha_n(s,a)$ et β_n tendant vers 0 selon les mêmes conditions que pour le Q-learning et pour le même type de compromis entre exploration et exploitation.

Bien que le R-learning soit moins connu et moins utilisé que le Q-learning, il semble qu'il présente des propriétés de vitesse de convergence plus intéressantes en pratique [MAH 96b]. Si peu de résultats théoriques existent à ce sujet, citons toutefois [GAR 98] où l'on montre qu'en horizon fini, le R-learning est très proche d'une version parallèle optimisée du Q-learning, expliquant ainsi ses meilleurs résultats expérimentaux.

D'autres algorithmes d'apprentissage par renforcement pour le critère moyen ont aussi été proposés [MAH 96b]. Parmi eux, l'algorithme B [JAL 89] est une méthode indirecte qui nécessite une estimation adaptative des fonctions $p()$ et $r()$. Citons aussi les travaux de Mahadevan [MAH 96a] qui a défini un algorithme toujours à base de modèles permettant d'apprendre des politiques biais-optimales, sur la base des équations d'optimalité de Bellman.

Nous allons voir à présent deux autres algorithmes qui visent aussi à maximiser le critère moyen, mais en construisant un modèle des transitions. Ils présentent la particularité théorique de disposer d'une preuve de convergence polynomiale en fonction du nombre d'états et non pas exponentielle comme c'est le cas pour les algorithmes précédents. Ils sont aussi connus pour avoir donné lieu à des prolongements dans le cadre des MDP factorisés, ce qui sera évoqué au chapitre **??**, dans le volume 2 de cet ouvrage.

1.5. Méthodes indirectes : apprentissage d'un modèle

Nous avons vu au paragraphe 1.4.4 qu'il existait un compromis entre la vitesse d'apprentissage et la mémoire utilisée pour apprendre. La solution consistant à mettre en œuvre des traces d'éligibilité reste limitée en ceci que l'apprentissage n'opère qu'à partir d'informations extraites du passé immédiat de l'agent. Nous allons voir à présent une approche différente dans laquelle l'agent élabore un modèle de ses interactions avec son environnement puis peut apprendre sur la foi de ce modèle, indépendamment de son expérience courante.

La question principale de ce type d'approche est la suivante : faut-il attendre de disposer d'un modèle le plus exact possible avant d'entamer la phase d'optimisation ? Peut-on entrelacer au maximum identification et optimisation, c'est-à-dire modifier à chaque transition observée la fonction de valeur et le modèle estimé ? Il semble actuellement que cette dernière approche soit préférable, comme par exemple dans ARTDP[12] [BAR 95], où acquisition du modèle, programmation dynamique et exécution sont concurrents. Parmi les algorithmes de ce type les plus connus en apprentissage par renforcement, citons aussi *Dyna* [SUT 90a], *Queue-Dyna* [PEN 93] et *Prioritized Sweeping* [MOO 93].

1.5.1. *Les architectures Dyna*

Tous les systèmes d'apprentissage par renforcement indirect[13] sont motivés par le même constat. Plutôt que d'attendre qu'un agent effectue des transitions dans son

12. *Adaptive Real-Time Dynamic Programming.*

13. *Model-based reinforcement learning.*

environnement réel, une façon d'accélérer la propagation de la qualité des situations consiste à construire un modèle des transitions réalisées par l'agent et à se servir de ce modèle pour appliquer un algorithme de propagation indépendamment du comportement de l'agent. En effet, quand un agent interagit avec son environnement, ses actions ne débouchent pas seulement sur une possible punition ou récompense, mais aussi sur une situation ultérieure. L'agent peut donc construire un modèle des transitions dont il fait l'expérience, indépendamment de tout modèle des récompenses.

L'idée de construire un modèle des transitions par apprentissage a été réalisée pour la première fois avec les architectures DYNA [SUT 90b]. En effet, Sutton a proposé cette famille d'architectures pour doter un agent de la capacité à mettre à jour plusieurs valeurs d'actions lors du même pas de temps, indépendamment de la situation courante de l'agent, de façon à accélérer sensiblement l'apprentissage de la fonction de valeur. Cette capacité est réalisée en appliquant au modèle un nombre fixé d'étapes de planification, qui consistent à appliquer un algorithme de programmation dynamique au sein du modèle de l'agent[14].

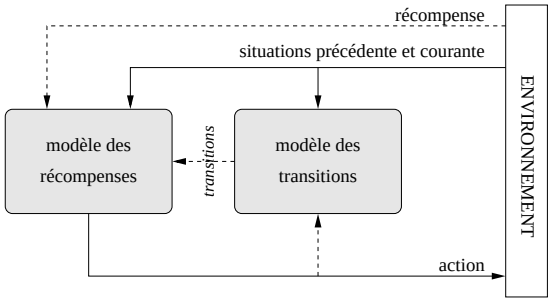

Figure 1.4. *Les architectures* DYNA *combinent un modèle des récompenses et un modèle des transitions. Le modèle des transitions est utilisé pour accélérer l'apprentissage par renforcement. La construction des deux modèles requiert une mémorisation de la situation précédente, ce qui n'est pas explicite sur la figure.*

Les architectures DYNA, présentées sur la figure 1.4, construisent un modèle des transitions effectuées par l'agent dans son environnement et un modèle des récompenses qu'il peut y obtenir.

Le modèle des transitions prend la forme d'une liste de triplets $\langle s_t, a_t, s_{t+1} \rangle$ qui indiquent que, si l'agent effectue l'action a_t dans l'état s_t, il atteindra immédiatement l'état s_{t+1}.

14. Voir le chapitre précédent.

Des transitions de la forme (s_t, a_t, s_{t+1}) constituent le modèle des interactions d'un agent avec son environnement. Au lieu d'apprendre que tel stimulus doit être suivi de telle action, l'agent apprend alors que, s'il effectue telle action après avoir reçu tel stimulus, alors il doit s'attendre à recevoir tel autre stimulus lors du pas de temps suivant.

Une fois ce modèle construit par apprentissage, on peut appliquer des algorithmes de programmation dynamique tels que l'itération sur les valeurs ou sur les politiques pour accélérer l'apprentissage de la fonction de valeur. Ainsi, le modèle des transitions peut être utilisé indépendamment de la situation courante de l'agent pour propager les valeurs de différents états à partir de diverses sources de récompense. En pratique, ce processus de propagation consiste à réaliser des actions simulées à partir de situations fictives, éventuellement plusieurs fois par pas de temps.

Lorsque l'agent est un robot réel dont chaque action effective peut être longue et difficile à mettre en œuvre, voire dangereuse pour son fonctionnement, réaliser des transitions simulées dans un modèle interne est beaucoup moins coûteux pour propager des valeurs que refaire chaque action laborieusement. En outre, si les buts attribués à l'agent changent, ce qui a pour effet de modifier les récompenses qu'il recevra, alors l'agent est capable de reconfigurer rapidement sa stratégie de comportement plutôt que de « désapprendre » tout ce qu'il a péniblement appris et d'apprendre à nouveau une autre stratégie comportementale.

Par ailleurs, la construction d'un tel modèle peut doter un agent de capacités d'anticipation. En effet, s'il enregistre les associations entre les états dans lesquels il se trouve d'une part et les récompenses qu'il reçoit d'autre part, l'agent peut choisir son action en fonction du caractère désirable pour lui de l'état auquel cette action doit le conduire. Au lieu d'avancer aveuglément vers une récompense attendue, l'agent peut alors mettre en œuvre des capacités de planification à plus long terme. En parcourant par un regard en avant le graphe des transitions d'état en état dont il dispose, l'agent devient capable de prévoir l'évolution de ses interactions avec son environnement en fonction de ses actions à un horizon indéfini. Si bien que, s'il veut atteindre un but dans un certain état, il peut rechercher au sein du graphe la séquence d'action qui le conduit à ce but, avant d'effectuer la séquence d'actions correspondante.

Les architectures *Dyna* constituent une famille de systèmes qui vérifient les principes que nous venons de décrire. Au sein de cette famille, on distingue cependant des variations sur l'algorithme d'apprentissage ou la méthode d'exploration utilisés. Le système original, *Dyna-PI*[15] repose sur un algorithme d'itération sur les politiques. Sutton [SUT 90b] montre que ce système est moins flexible que le système *Dyna-Q*,

15. PI pour *Policy Iteration*. A noter que dans [SUT 98], Sutton appelle ce même système *Dyna-AC*, avec AC pour *Actor Critic*.

qui repose sur le Q-learning. Il propose en outre une version dotée de capacités d'exploration active, nommée *Dyna-Q+* et présente la différence de performance entre ces trois systèmes sur des environnements changeants dans lesquels soit des chemins optimaux sont bloqués, soit, au contraire, des raccourcis apparaissent.

Enfin, tous ces systèmes reposant sur un algorithme de programmation dynamique pour propager efficacement la fonction de valeur ou la fonction de qualité, il est possible de rendre cette programmation dynamique plus efficace avec un algorithme de « balayage prioritaire » tel que *Prioritized Sweeping* [MOO 93], qui met à jour en priorité les valeurs des états dans lesquels l'agent est susceptible de se trouver, ou d'autres variantes telles que *Focused Dyna* [PEN 92], *Experience Replay* [LIN 93] et *Trajectory Model Updates* [KUV 96].

Examinons à présent les algorithmes dotés d'une preuve de convergence en fonction polynomiale de la taille du problème.

1.5.2. *L'algorithme E^3*

Le premier de ces algorithmes s'appelle E^3, pour *Explicit Explore and Exploit* [KEA 98]. De même que les architectures DYNA, l'algorithme E^3 repose sur la construction d'un modèle des transitions et des récompenses. Néanmoins, au lieu de chercher à construire un modèle de l'environnement tout entier, il ne mémorise qu'un sous-ensemble des transitions rencontrées, à savoir celles qui jouent un rôle dans la définition de la politique optimale.

Pour construire ce modèle, l'algorithme visite les états de façon homogène, ce qui signifie que, quand il arrive dans un nouvel état, il choisit l'action qu'il a effectuée le moins souvent dans cet état. Il tient alors à jour une probabilité observée de transition vers des états suivants, ce qui revient à se donner un modèle approché du MDP sous-jacent.

Au cœur de l'algorithme E^3 se trouve un mécanisme de gestion du compromis entre exploration et exploitation qui repose sur le principe dit « de la case courrier[16] ». L'idée est que, si l'on visite un nombre fini d'états de façon homogène, il finit toujours par y avoir un état qui a été visité suffisamment souvent pour que l'estimation des probabilités de transition construite sur la base des observations à partir de cet état soit proche de la distribution de probabilité effective engendrée par l'environnement. Les auteurs définissent alors une notion d'état « connu » de telle façon que le nombre de visites suffisant pour connaître un état reste polynomial en fonction de la taille du problème.

16. *The pigeon hole principle.*

Le modèle construit par E^3 est un MDP dans lequel figurent tous les états « connus » avec les probabilités de transition observées et un état absorbant qui représente à lui seul tous les états non encore « connus ». L'algorithme est alors face à une alternative :

– soit il existe une politique proche de l'optimum qui repose uniquement sur des états connus et l'algorithme peut trouver cette politique en appliquant un algorithme de programmation dynamique sur le modèle qu'il a construit ;

– soit cette condition n'est pas vérifiée et il faut explorer davantage, donc choisir des actions qui mènent à l'état absorbant afin de mieux connaître les états de l'environnement qu'il regroupe.

Cet algorithme est simple et présente l'intérêt de disposer de preuves qui garantissent le caractère polynomial de la convergence. Pour déterminer dans quelle branche de l'alternative ci-dessus on se trouve, l'algorithme est capable de calculer si explorer davantage au-delà d'un horizon fixé pour mieux connaître le modèle permet d'améliorer de façon significative la meilleure politique connue. Si ce n'est pas le cas, il vaut mieux exploiter qu'explorer.

1.5.3. *L'algorithme $R_{\max}$*

L'algorithme R_{max} [BRA 01] apparaît comme une alternative intéressante à l'algorithme E^3. Il repose sur le même principe général que E^3 consistant à construire un modèle des transitions et à rechercher une politique qui maximise la récompense moyenne sur un horizon fini. Cependant, d'une part, il simplifie la gestion du dilemme entre exploration et exploitation en initialisant toutes les espérances de récompense à une valeur optimiste égale au maximum des récompenses immédiates atteignables dans l'environnement[17] et, d'autre part, il étend le cadre des MDP dans lequel travaille E^3 au cadre des jeux stochastiques à somme nulle, ce qui permet de prendre en compte la présence d'un adversaire. La principale différence provient de ce que l'initialisation des récompenses à une valeur optimiste permet à l'algorithme R_{max} d'éviter d'avoir à gérer explicitement le compromis entre exploration et exploitation. En effet, l'agent se contente d'aller vers les états dans lesquels il s'attend à recevoir une récompense élevée. La récompense attendue peut être élevée pour deux raisons : soit parce que l'état est mal connu et dans ce cas le comportement de l'agent relève de l'exploration, soit parce que l'agent sait qu'il a déjà reçu des récompenses dans l'état en question et son comportement relève de l'exploitation.

Les auteurs argumentent la supériorité de leur approche sur celle de l'algorithme E^3 en soulignant que, en présence d'un adversaire, on ne maîtrise pas complètement

17. D'où le nom de l'algorithme, R_{max}.

les transitions du système, donc on ne maîtrise pas totalement le choix entre exploration et exploitation. Avec R_{max}, l'algorithme explore ou exploite « au mieux » compte tenu des choix réalisés par l'adversaire. L'algorithme est globalement plus simple et constitue donc un excellent candidat pour gérer efficacement le compromis entre exploration et exploitation.

Comme l'algorithme E^3, R_{max} repose sur deux hypothèses irréalistes qui compromettent son application en pratique. D'une part, on suppose connu l'horizon T au-delà duquel chercher à améliorer le modèle en explorant ne permet plus d'améliorer la politique de façon significative. D'autre part, on suppose qu'à chaque pas de temps, on est capable de déterminer efficacement la politique optimale sur cet horizon T, compte tenu de la connaissance dont on dispose dans le modèle. La difficulté provient de ce que T peut être grand, si bien que trouver une politique optimale sur cet horizon peut être très long et que donner à T une valeur arbitraire suppose de prendre une valeur encore plus grande que la valeur idéale, ce qui ne fait qu'aggraver le problème.

En vue d'applications pratiques de ces algorithmes conçus essentiellement en fonction de la possibilité de disposer de preuves théoriques de convergence, des variantes ont été proposées pour accélérer la recherche d'une politique optimale en faisant un échantillonnage heuristique[18] des transitions plutôt qu'une exploration systématique homogène [PER 04, KEA 02]. Dans [BRA 03], les auteurs utilisent aussi des valeurs bien inférieures à la valeur de T théorique et montrent que l'algorithme se comporte de mieux en mieux au fur et à mesure que T augmente, au prix d'un temps de calcul croissant.

1.6. Conclusion

L'apprentissage par renforcement est aujourd'hui une discipline très active. A l'interface entre apprentissage automatique et sciences cognitives, contrôle optimal et optimisation par simulation, l'apprentissage par renforcement utilise des techniques variées pour aborder le problème de l'acquisition d'un comportement optimal dans un environnement incertain et dynamique.

Nous avons principalement présenté dans ce chapitre les méthodes classiques de l'apprentissage par renforcement, qui ont historiquement fondé la discipline. Toutefois, l'apprentissage par renforcement étant en plein développement, de nombreuses méthodes plus récentes n'ont pu être présentées ici. On peut citer par exemple les travaux actuels sur les modèles d'apprentissage par renforcement probablement approximativement corrects (PAC), comme MBIE [STR 05] ou *Delayed Q-learning* [STR 06].

18. *Heuristic sampling.*

On trouvera enfin dans la suite de cet ouvrage d'autres approches classiques qui partagent également l'emploi de la simulation, telles les méthodes de résolution en ligne (chapitre **??** du volume 2), les méthodes de programmation dynamique avec approximation de la fonction de valeur (chapitre **??** du volume 2) ou encore les méthodes de gradient pour l'optimisation de politiques paramétrées (chapitre **??** du volume 2).

1.7. Bibliographie

[BAR 83] BARTO A., SUTTON R., ANDERSON C. W., « Neuron-like Adaptive Elements That Can Solve Difficult Learning Control Problems », *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-13, n°5, p. 834–846, 1983.

[BAR 95] BARTO A., BRADTKE S., SINGH S., « Learning to Act Using Real-time Dynamic Programming », *Artificial Intelligence*, vol. 72, p. 81–138, 1995.

[BER 95] BERTSEKAS D., *Dynamic Programming and Optimal Control*, Athena Scientific, Belmont, MA, 1995.

[BER 96] BERTSEKAS D., TSITSIKLIS J., *Neuro-Dynamic Programming*, Athena Scientific, Belmont, MA, 1996.

[BRA 01] BRAFMAN R. I., TENNENHOLTZ M., « R-max : a General Polynomial Time Algorithm for Near-optimal Reinforcement Learning », *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01)*, p. 953–958, 2001.

[BRA 03] BRAFMAN R. I., TENNENHOLTZ M., « Learning to Coordinate Efficiently : A Model Based Approach », *Journal of Artificial Intelligence Research*, vol. 19, p. 11–23, 2003.

[CHA 95] CHAUVIN Y., RUMELHART D. E. (dir.), *Backpropagation : theory, architectures, and applications*, Lawrence Erlbaum Associates, Inc., Mahwah, N.J., 1995.

[CIC 95] CICHOSZ P., « Truncating Temporal Differences : On the Efficient Implementation of TD(λ) for Reinforcement Learning », *Journal of Artificial Intelligence Research*, vol. 2, p. 287–318, 1995.

[DAY 94] DAYAN P., SEJNOWSKI T. J., « TD(λ) Converges with Probability 1 », *Machine Learning*, vol. 14, n°3, p. 295–301, 1994.

[GAR 98] GARCIA F., NDIAYE S., « A Learning Rate Analysis of Reinforcement-Learning Algorithms in Finite-Horizon », *Proceedings of the 15th International Conference on Machine Learning (ICML'98)*, Morgan Kaufmann, San Mateo, CA, p. 215–223, 1998.

[GOL 89] GOLDBERG D. E., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison Wesley, Reading, MA, 1989.

[JAA 94] JAAKKOLA T., JORDAN M. I., SINGH S. P., « On the Convergence of Stochastic Iterative Dynamic Programming Algorithms », *Neural Computation*, vol. 6, p. 1185–1201, 1994.

[JAL 89] JALALI A., FERGUSON M., « Computationally Efficient Adaptative Control Algorithms for Markov Chains », *Proceedings of the IEEE Conference on Decision and Control (CDC'89)*, vol. 28, p. 1283–1288, 1989.

[JOZ 01] JOZEFOWIEZ J., *Conditionnement opérant et Problèmes décisionnels de Markov*, Thèse de doctorat de l'Université Lille III, Lille, 2001.

[KAE 93] KAELBLING L. P., *Learning in Embedded Systems*, MIT Press, Cambridge, MA, 1993.

[KAE 98] KAELBLING L., LITTMAN M., CASSANDRA A., « Planning and Acting in Partially Observable Stochastic Domains », *Artificial Intelligence*, vol. 101, p. 99–134, 1998.

[KEA 98] KEARNS M., SINGH S., « Near-Optimal Reinforcement Learning in Polynomial Time », *Machine Learning*, vol. 49, 1998.

[KEA 02] KEARNS M. J., MANSOUR Y., NG A. Y., « A Sparse Sampling Algorithm for Near-Optimal Planning in Large Markov Decision Processes », *Machine Learning*, vol. 49, n°2-3, p. 193–208, 2002.

[KIR 87] KIRKPATRICK S., C. D. GELATT J., VECCHI M. P., « Optimization by simulated annealing », *Readings in computer vision : issues, problems, principles, and paradigms*, p. 606–615, Morgan Kaufmann Publishers Inc., San Francisco, CA, Etats-Unis, 1987.

[KLO 72] KLOPF H. A., Brain Function and Adaptive Systems, A Heterostatic Theory, Rapport n°Technical Report AFCRL-72-0164, Air Force Cambridge Research Laboratories, 1972.

[KLO 75] KLOPF H. A., « A Comparison of Natural and Artificial Intelligence », *SIGART newsletter*, vol. 53, p. 11–13, 1975.

[KOZ 92] KOZA J. R., *Genetic Programming : On the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, MA, 1992.

[KUV 96] KUVAYEV L., SUTTON R. S., « Model-Based Reinforcement Learning with an Approximate, Learned Model », *Proceedings of the 9th Yale Workshop on Adaptive and Learning Systems*, Yale University Press, New Haven, CT, p. 101–105, 1996.

[LIN 93] LIN L.-J., « Scaling Up Reinforcement Learning for Robot Control », *Proceedings of the 10th International Conference on Machine Learning (ICML'93)*, p. 182–189, 1993.

[MAH 96a] MAHADEVAN S., « An Average-Reward Reinforcement Learning Algorithm for Computing Bias-Optimal Policies », *Proceedings of the National Conference on Artificial Intelligence (AAAI'96)*, vol. 13, 1996.

[MAH 96b] MAHADEVAN S., « Average Reward Reinforcement Learning : Foundations, Algorithms and Empirical Results », *Machine Learning*, vol. 22, p. 159–196, 1996.

[MEU 96] MEULEAU N., Le dilemme entre exploration et exploitation dans l'apprentissage par renforcement, thèse de doctorat, Cemagref, université de Caen, 1996.

[MEU 99] MEULEAU N., BOURGINE P., « Exploration of Multi-State Environments : Local Measures and Back-Propagation of Uncertainty », *Machine Learning*, vol. 35, n°2, p. 117–154, 1999.

[MIC 61] MICHIE D., « Trial and Error », *Science Survey*, vol. 2, p. 129–145, 1961.

[MIC 68] MICHIE D., CHAMBERS R., « BOXES: An Experiment in Adaptive Control », *Machine Intelligence*, vol. 2, p. 137–152, 1968.

[MOO 93] MOORE A., ATKESON C., « Prioritized Sweeping : Reinforcement Learning with Less Data and Less Real Time », *Machine Learning*, vol. 13, p. 103–130, 1993.

[NDI 99] NDIAYE S., Apprentissage par renforcement en horizon fini : application à la génération de règles pour la conduite de culture, thèse de doctorat, université Paul Sabatier, Toulouse, 1999.

[PEN 92] PENG J., WILLIAMS R., « Efficient Learning and Planning within the DYNA framework », *Proceedings of the 2nd International Conference on Simulation of Adaptive Behavior (SAB'92)*, p. 281–290, 1992.

[PEN 93] PENG J., WILLIAMS R. J., « Efficient Learning and Planning within the Dyna Framework », *Adaptive Behavior*, vol. 1, n°4, p. 437–454, 1993.

[PEN 94] PENG J., WILLIAMS R. J., « Incremental Multi-Step Q-learning », *Proceedings of the 11th International Conference on Machine Learning (ICML'94)*, p. 226–232, 1994.

[PEN 96] PENG J., WILLIAMS R. J., « Incremental Multi-Step Q-learning », *Machine Learning*, vol. 22, p. 283–290, 1996.

[PER 04] PERET L., GARCIA F., « On-line Search for Solving MDPs via Heuristic Sampling », *Proceedings of the European Conference on Artificial Intelligence (ECAI'04)*, 2004.

[RES 72] RESCORLA R. A., WAGNER A. R., « A Theory of Pavlovian Conditioning : Variations in the Effectiveness of Reinforcement and Nonreinforcement », BLACK A. H., PROKAZY W. F. (dir.), *Classical Conditioning II*, p. 64–99, Appleton Century Croft, New York, NY, 1972.

[RUM 94] RUMMERY G. A., NIRANJAN M., On-Line Q-learning using Connectionist Systems, Rapport, Cambridge University Engineering Department, Cambridge, Royaume-Uni, 1994.

[SAM 59] SAMUEL A., « Some Studies in Machine Learning using the Game of Checkers », *IBM Journal of Research Development*, vol. 3, n°3, p. 210–229, 1959.

[SCH 93] SCHWARTZ A., « A Reinforcement Learning Method for Maximizing Undiscounted Rewards », *Proceedings of the 10th International Conference on Machine Learning (ICML'93)*, 1993.

[SIG 04] SIGAUD O., Comportements adaptatifs pour les agents dans des environnements informatiques complexes, mémoire d'habilitation à diriger des recherches, université Paris VI, 2004.

[SIN 96] SINGH S. P., SUTTON R. S., « Reinforcement Learning with Replacing Eligibility Traces », *Machine Learning*, vol. 22, n°1, p. 123–158, 1996.

[SIN 00] SINGH S. P., JAAKKOLA T., LITTMAN M. L., SZEPESVARI C., « Convergence Results for Single-Step On-Policy Reinforcement Learning Algorithms », *Machine Learning*, vol. 38, n°3, p. 287–308, 2000.

[STR 05] STREHL A. L., LITTMAN M. L., « A theoretical analysis of Model-Based Interval Estimation », *Proceedings of the 22nd International Conference on Machine Learning (ICML'05)*, ACM, New York, NY, Etats-Unis, p. 856–863, 2005.

[STR 06] STREHL A. L., LI L., WIEWIORA E., LANGFORD J., LITTMAN M. L., « PAC model-free reinforcement learning », *Proceedings of the 23rd International Conference on Machine Learning (ICML'06)*, ACM, New York, NY, Etats-Unis, p. 881–888, 2006.

[SUT 81] SUTTON R., BARTO A., « Toward a Modern Theory of Adaptive Network : Expectation and Prediction », *Psychological Review*, vol. 88, n°2, p. 135–170, 1981.

[SUT 88] SUTTON R., « Learning to Predict by the Method of Temporal Differences », *Machine Learning*, vol. 3, n°1, p. 9–44, 1988.

[SUT 90a] SUTTON R. S., « Integrated Architectures for Learning, Planning and Reacting Based on Approximating Dynamic Programming », *Proceedings of the 7th International Conference on Machine Learning (ICML'90)*, p. 216–224, 1990.

[SUT 90b] SUTTON R. S., « Planning by Incremental Dynamic Programming », *Proceedings of the 8th International Conference on Machine Learning (ICML'91)*, Morgan Kaufmann, San Mateo, CA, p. 353–357, 1990.

[SUT 98] SUTTON R. S., BARTO A. G., *Reinforcement Learning : An Introduction*, Bradford Book, MIT Press, Cambridge, MA, 1998.

[THR 92] THRUN S., « The Role of Exploration in Learning Control », WHITE D., SOFGE D. (dir.), *Handbook for Intelligent Control : Neural, Fuzzy and Adaptive Approaches*, Van Nostrand Reinhold, Florence, Kentucky, 1992.

[WAT 89] WATKINS C., Learning from Delayed Rewards, thèse de doctorat, Cambridge University, Cambridge, Royaume-Uni, 1989.

[WAT 92] WATKINS C., DAYAN P., « Q-learning », *Machine Learning*, vol. 8, n°3, p. 279–292, 1992.

[WIT 77] WITTEN I. H., « An Adaptive Optimal Controller for Discrete-Time Markov Environments », *Information and Control*, vol. 34, p. 286–295, 1977.