IN104 : Apprentissage par renforcement dans un labyrinthe

Anthony Truchet*

11 mars 2010

Table des matières

Table des figures

Liste des tableaux

*anthony.truchet@ensta.org

Introduction

Le cours d'IN104 vise à acquérir une première expérience de *projet* de développement informatique. C'est à dire de partir d'une problématique plus ou moins définie et d'aboutir, dans les délais, à un produit "livrable" conforme aux attentes. Et ceci est tout à fait différent de simplement "produire du code qui marche"...

La problématique retenue est celle de l'*apprentissage par renforcement*. L'apprentissage par renforcement est une famille de méthodes (SEC 3 p.4) permettant à un *agent* **d'apprendre** à "bien se comporter" dans son *environnement* — nous utiliserons un petit labyrinthe (SEC 2 p.3) — *a priori* inconnu.

1 Présentation du projet

Une interface graphique simple vous sera fournie. Celle-ci permettra de visualiser au fur et à mesure le fonctionnement des algorithmes, de mieux comprendre comment ils marchent et... ce qui ne marche pas !

1.1 Déroulement du projet

Toute réalisation logicielle "non-triviale" passe par différentes étapes :

1. La prise en main de l'environnement de développement.
2. La compréhension de la problématique à traiter.
3. Le choix d'une méthode de résolution.
4. La phase de génie logiciel [1] : il s'agit de déterminer une *architecture logicielle* pour l'application développée.
5. Coder et tester l'application.
6. Créer un "livrable" : il s'agit de présenter l'application et sa documentation sous une forme utilisable par un tiers — le client, vous même ou, dans le cadre de ce cours, le jury !

Le déroulement du projet suivra donc ces étapes sur tout au long des sept séances :
– La première séance sera consacrée à la prise en main de l'environnement de développement.
– La deuxième séance permettra de découvrir l'apprentissage par renforcement et de commencer à utiliser l'interface graphique fournie.
– Lors de la troisième séance il faudra déterminer une architecture pour le projet et (commencer à) la mettre en place.
– Les 3 séances suivantes seront consacrées au développement proprement dit des fonctionnalités.
– Enfin la dernière séance sera réservées aux "finitions" du projet.

1.2 Cahier des charges

Au terme du projet chaque binôme devra fournir une application permettant :
– De lire un fichier au format "laby" (cf. 2 P.3), d'afficher le labyrinthe correspondant et de piloter manuellement le robot dans cet environnement.
– De déterminer une *politique optimale* — c'est-à-dire un meilleur comportement — pour le robot en supposant que celui-ci connaît parfaitement le monde. Il s'agit donc d'implémenter une *méthode de planification* : vous implémenterez la méthode *Value Iteration* présentée SEC 3.1.2 p.5 et plus particulièrement par EQU 6 p.6.
– De faire apprendre au robot une politique optimale, dans un environnement inconnu *a priori*, par la méthode d'apprentissage appelée *Q-learning* (cf. SEC 3.2.2 p.6).
– De mettre en œuvre l'une des techniques présentées dans la section 3.3 P.7 "Pour aller plus loin..."
– De comparer les propriétés et les performances desdifférentes méthodes implémentées.

Outre l'application trois *courts* documents seront réalisés au cours du projet :

des spécifications logicielles (entre les séances 3 et 4) Celui-ci présentera les fonctionnalités — principalement les méthodes d'apprentissage et de planification retenues — ainsi que l'architecture logicielle envisagée.

1. Le génie logiciel est un domaine a part entière, dépassant le cadre de ce cours : on n'utilisera donc pas les outils classiques du G.L. en IN104. Néanmoins *il est indispensable de réfléchir un peu à l'architecture du code avant de commencer à coder* !

une documentation de l'application Celle-ci présentera *comment* utiliser l'application et quelles sont les fonctionnalités *effectivement* disponibles.

un résumé des performances observées des différentes méthodes d'apprentissage testées.

2 L'environnement *Labyrinthe*

Afin de préserver une dimension stimulante au projet on se placera dans un *environnement* à la fois simple, visuel, et suffisamment riche pour être intéressant. Il s'agit d'un grand classique en apprentissage par renforcement : le *grid-world* ou labyrinthe [2].

2.1 Présentation du labyrinthe

Il s'agit d'une sorte de damier à géométrie torique (les bords nord-sud et est-ouest sont joints) dans lequel évolue un "robot" qui eut se déplacer d'une case à la fois dans les quatre directions cardinales.

Chaque case du damier est d'un certain type de case, défini par les caractéristiques suivantes :

la couleur : information visuelle pour l'utilisateur (voire le robot cf 2.2 P.3),

l'accessibilité : si le robot peut pénétrer sur la case,

le caractère "terminal" : lorsque le robot est sur une case terminale, l'épisode s'achève.

la récompense : la récompense (ou punition) obtenue par le robot lorsqu'il *essaye de* pénétrer sur la case,

un symbole : un caractère utilisé dans le format de fichier "laby" pour représenter le type de case.

Quatre types de cases sont proposés FIG 1 p.3 ; ils sont suffisants pour commencer mais d'autres peuvent être imaginés... Enfin, la position d'une case est repérée par un couple d'entier (ligne, colonne) à partir du coin supérieur gauche.

Symb.	Coul.	Acces.	Termin.	Récomp.	Description
_	Blanc	Oui	Non	0	Une case vide "normale"
*	Vert	Oui	Oui	10	La sortie du labyrinthe
#	Bleu	Non	—	-1	Un mur : inaccessible et rentrer dedans est légèrement pénalisé
!	Rouge	Oui	Non	-10	Une zone dangereuse

TABLE 1 – Quelques types de cases possibles

```
LABY
SIZE #lignes #colonne
#### ··· #colonnes symboles  ···####
#_*! ··· #colonnes symboles  ···___#
#lignes lignes
END
```

TABLE 2 – Le format de fichier "laby"

2.2 Lien avec le cadre des MDP

La formalisation en terme de MDP demande d'identifier un espace d'état $\mathcal{S}$ et un espace d'action $\mathcal{A}$, ainsi qu'une fonction de transition T et de récompense R. Le labyrinthe est un monde déterministe — c'est-à-dire que R et T sont de vraies fonctions, pas des variables aléatoires — R et T sont donc simplement donnée par le "sens commun" du déplacement dans un labyrinthe.

L'espace d'action est assez facilement identifiable : $\mathcal{A} = \{Nord, Sud, Est, Ouest\}$. L'état du robot dans le labyrinthe est donné par sa position (l,c) ; l'espace d'état est donc
$\mathcal{S} = \{(l,c) \in \mathbb{N}^2 / 1 \leq l \leq \#lignes, 1 \leq c \leq \#colonnes\}$.

2. NdT : traduction libre ;-)

Une variante beaucoup plus intéressante est celle où le robot ne connaît pas sa position absolue, mais perçoit simplement la couleur de sa case et des cases voisines. On se retrouve alors dans le cadre des POMDP, qui sort un peu du cadre de ce projet...

3 L'apprentissage par renforcement

L'apprentissage par renforcement est magnifiquement bien présenté dans l'ouvrage *Reinforcement Learning : An Introduction* de Richard S. Sutton et Andrew G. Barto, disponible en ligne [3]. Outre une introduction très abordable et une vision unificatrice très intéressante du problème, cet ouvrage formule très clairement les algorithmes des méthodes proposées.

Cette partie n'est qu'un survol rapide de l'apprentissage par renforcement, et un mémento de ce qui sera présenté lors de la deuxième séance du cours. Si elle peut suffire pour comprendre les algorithmes simples, il serait indispensable de se reporter au Sutton&Barto pour pouvoir implémenter les algorithmes plus avancés.

3.1 L'apprentissage par renforcement : agent situé et motivé

On distingue classiquement 2 grands types d'apprentissage : *supervisé*, où l'on connaît des échantillons de la fonction à apprendre pour se guider (e.g. réseaux de neurones *feed-forward*), et *non-supervisé* qui consiste généralement à faire de la classification automatique (e.g. cartes de Kohonen). L'apprentissage par renforcement constitue une troisième catégorie dans laquelle, bien que la fonction à apprendre ne soit pas fournie, une évaluation scalaire des performances du système est cependant fournie au système pour le guider dans son apprentissage.

Un agent (animal, robot, programme de contrôle...) est en interaction permanente avec son *environnement*. Cette interaction se traduit notamment par les changements que l'agent peut apporter à son état dans l'environnement par l'intermédiaire de ses actions. En outre, certaines actions ont un effet favorable pour l'agent (e.g. manger) alors que d'autres sont néfastes (e.g. toucher un fil électrifié) : l'agent adapte son comportement en fonction des retours cumulés sur le long terme.

3.1.1 Formalisations fondamentales

L'apprentissage par renforcement formalise ce type d'interactions autour de quelques notions clefs dont l'organisation est illustrée FIG 1 p.4. Ces schémas d'interaction sont connus sous le nom de processus de décision Markoviens (MDP).

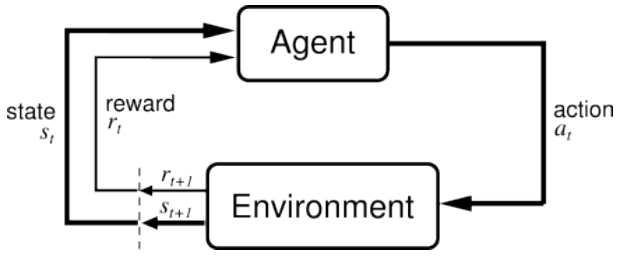

FIGURE 1 – Interactions entre agent et environnement en apprentissage par renforcement)

MDP Un MDP formalise ce que nous avons appelé l'environnement ; il est défini par un ensemble d'états $\mathcal{S}$, un ensemble d'actions $\mathcal{A}$ et :

- Une fonction de transition T telle que $T_{s,a}$ soit une variable aléatoire sur $\mathcal{S}$, représentant les probabilités de transition vers l'état suivant lorsque que l'on effectue l'action a dans l'état s : $T_{s,a}(s')$ est donc la probabilité de transition de l'état s vers l'état s' lorsque l'on fait l'action a.
- Une fonction de récompense R telle que $R_{s,a}$ soit une variable aléatoire sur $\mathbb{R}$, représentant la distribution de probabilité de récompense lorsque que l'on effectue l'action a dans l'état s : $R_{s,a}(r)$ est donc la probabilité de recevoir une récompense r dans ces conditions.

3. http ://www.cs.ualberta.ca/ sutton/book/ebook/the-book.html

On suppose que les états contiennent "toute l'information pertinente", c'est à dire qu'ils vérifient l'hypothèse de Markov comme quoi $T_{s,a}$ et $R_{s,a}$ sont indépendantes des états et actions antérieurs.

Ceci constitue la formulation générale d'un MDP dans le *cas stochastique*. Le *cas déterministe* s'obtient en remplaçant la fonction stochastique de transition et de récompense par une fonction classique ; ainsi $T_{s,a} \in \mathcal{S}$, $R_{s,a} \in \mathbb{R}$ peuvent être simplement représentées par des tables.

Il existe de nombeuses autre variantes dont on ne citera que les POMDP (*Partialy Observable Markov Decision Problems*), ou problèmes de décision Markovien partiellement observables. Il s'agit du cas (extrêmement fréquent) où l'agent ne connaît pas son état réel, mais à simplement une *perception* de son environnement qui est une fonction de son état. C'est le cas, par exemple, d'un robot qui "voit" ce qui l'entoure mais ne connaît pas sa position et son orientation absolues.

Politique et fonction de valeur associée La notion de comportement est formalisée par celle de "politique". Une politique Π est une fonction telle que Π_s soit une variable aléatoire sur $\mathcal{A}$ représentant les probabilités suivant lesquelles un agent suivant cette politique effectue chaque action lorsque qu'il se trouve dans l'état s. $\Pi_s(a)$ est donc la probabilité que l'agent choisisse l'action a dans l'état s.

Pour un MDP donné, on recherche un "meilleur comportement", il faut donc définir sur les politiques un critère d'optimalité que l'on cherche à maximiser. La *valeur* d'une politique donnée doit prendre en compte toutes les récompenses futures, tout en tenant compte d'un horizon temporel : les hypothétiques récompenses lointaines doivent avoir un poids moindre que les récompenses proches quasi-certaines.

Le choix classique est la "récompense actualisée" ou *discounted reward* ; pour un MDP et une politique donnée elle correspond de façon informelle à $r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$ où γ (*discount factor*) permet de fixer l'horizon temporel : pour $\gamma = 0$ on ne tient compte que de la récompense immédiate tandis que pour $\gamma \to 1$ on tient compte de toutes les récompenses futures de façon égale.

Pour représenter plus formellement cette notion, on définit donc, pour une politique Π, la qualité $Q^\Pi(s,a)$ d'une action a dans l'état s et la valeur $V^\Pi(s)$ d'un état par :

$$Q^\Pi(s,a) = \mathbb{E}_{r\sim R_{s,a}}[r] + \gamma.\mathbb{E}_{s'\sim T_{s,a}}\left[V^\Pi(s')\right] \quad (1)$$

$$V^\Pi(s) = \mathbb{E}_{a\sim\Pi_s}\left[Q^\Pi(s,a)\right] \quad (2)$$

3.1.2 Sans apprentissage : la programmation dynamique

L'objectif des méthodes de *programmation dynamique* est de trouver une politique optimale dans un *environnement connu*. Il ne s'agit donc pas d'une méthode d'apprentissage, mais plutôt de planification optimale. Pour ce faire, un premier problème consiste à évaluer, pour une politique donnée et dans un *MDP connu*, la *valeur* $V(s)$ de chaque état ou la *qualité* $Q(s,a)$ de chaque action dans chaque état.

Une possibilité consiste à résoudre les équations (2 et 1) de façon itérative, en l'interprétant comme une équation de mise à jour des valeurs $V(s)$ stockées dans une table. C'est l'étape appelée *Policy Evaluation* :

$$V^\Pi(s) \leftarrow \underbrace{\sum_a \Pi_s(a)}_{\text{politique}} \underbrace{\left(\sum_r R_{s,a}(r) + \gamma \sum_{s'} T_{s,a}(s')V^\Pi(s')\right)}_{\text{environnement}} \quad (3)$$

Comme on a supposé que l'on connaît T et R on peut alors reconstituer $Q(s,a)$:

$$Q(s,a) = \sum_r R_{s,a}(r) + \gamma \sum_{s'} T_{s,a}(s')V^\Pi(s') \quad (4)$$

Dès lors que l'on connaît la qualité de chaque action en chaque état, on peut améliorer itérativement la politique suivie jusqu'à présent en adoptant la politique qui consiste a effectuer, dans chaque état, l'action de qualité estimée maximale. C'est l'étape appelée *Policy Improvement* et on peut démontrer que :

$$\Pi'(s) = \text{argmax}_a\, Q^\Pi(s,a) \quad \Rightarrow \quad \forall s \in \mathcal{S},\ V^{\Pi'}(s) \geq V^\Pi(s) \quad (5)$$

Cette méthode de mise à jour de la politique, connue sous le nom de *Policy Iteration* (P.I.) converge vers la politique optimale.

Il est possible de fusionner en une seule itération les deux opération, tout en conservant les propriétés de convergence. Cet algorithme est connu sous le nom de *value itération* (V.I.) :

$$V^{\Pi}(s) \quad \leftarrow \quad \max_a \sum_r R_{s,a}(r) + \gamma \sum_{s'} T_{s,a}(s') V^{\Pi}(s') \tag{6}$$

Si ces méthodes permettent effectivement de trouver une politique optimale, elles supposent en revanche que le MDP est parfaitement connu, ce qui est plutôt l'exception que la règle en pratique. C'est pourquoi une autre grande famille de méthode d'apprentissage par renforcement a été développée, qui évite ces écueils en essayant d'estimer *directement* la fonction de valeur à partir de l'expérience de l'agent.

3.2 Les méthodes de différence temporelle

Contrairement aux méthodes V.I. et P.I. qui reposent sur la connaissance *a priori* du MDP, les méthodes dites de *différence temporelle* reposent, elles, uniquement sur la *séquence d'interactions* entre l'agent et l'environnement, et plus particulièrement sur une quantité appelée "erreur de différence temporelle" (*temporal difference error — T.D. error*), qui représente une erreur d'anticipation de la récompense.

3.2.1 Erreur de différence temporelle et *TD-learning*

On adapte les notation présentées précédemment en notant s^t, a^t, $r^t \ldots$ la valeur à la t^e interaction d'une séquence des quantités s, a, $r \ldots$ précédemment introduites. Si l'on suppose que les valeurs $V^{\Pi}(s)$ des états pour la politique Π suivie lors de la séquence sont connues exactement, alors on doit avoir :

$$V^{\Pi}(s^t) \quad = \quad r^t + \underbrace{\gamma . r^{t+1} + \gamma^2 . r^{t+1} + \cdots}_{\gamma . V^{\Pi}(s^{t+1})} \tag{7}$$

Mais comme on ne connaît pas l'environnement, on ne connaît pas $V^{\Pi}(s)$; on en construit donc progressivement une estimation $V(s)$. D'après l'équation ci-dessus, la quantité $V(s^t) - \gamma . V(s^{t+1})$ est une estimation de la récompense r^t, que l'on peut comparer à la récompense effectivement reçue.

On définit donc l'*erreur de différence temporelle* δ^t *— T.D. error —* comme la différence entre la récompense r^t effectivement reçue et cette estimation. L'algorithme TD(0) — donnée ci-dessous — utilise cette erreur δ pour apprendre $V(s)$ — conservée en mémoire sous forme de valeurs tabulées — par corrections successives :

$$\delta^t \quad = \quad r^t + \gamma . V(s^{t+1}) - V(s^t) \tag{8}$$

$$V(s) \quad \leftarrow \quad V^t(s) + \alpha . \delta^t \tag{9}$$

où α désigne le taux d'apprentissage[4] (*learning rate*). En effet, si δ^t est positive, cela signifie que l'estimation courante de $V(s^t)$ est sous estimée au vu de l'exécution de la t^e action et inversement si δ^t est négative. Cette formule itérative peut être rapprochée de celle utilisée pour la *Policy Evaluation*.

S'il est vrai que cet algorithme converge et présente les avantages mentionnés pour V.I., il ne permet que d'évaluer la fonction de valeur d'une politique connue : il ne fournit donc pas de politique et n'est pas très utile en l'état. C'est pourquoi différentes variantes ont été développées qui pallient ce problème : nous ne présenterons que SARSA et le *Q-learning*.

3.2.2 SARSA et *Q-learning*

On peut remarquer que l'algorithme TD(0) travaille sur les valeurs $V(s)$, qui ne comportant aucune information concernant l'action. Or il est possible d'appliquer exactement le même principe aux qualités $Q(s,a)$ qui, elles, contiennent une telle information. On obtient :

$$\delta^t \quad = \quad r^t + \gamma . Q(s^{t+1}, a^{t+1}) - Q(s^t, a^t) \tag{10}$$

$$Q(s,a) \quad \leftarrow \quad Q(s^t, a^t) + \alpha . \underbrace{\left(r^t + \gamma . Q^t(s^{t+1}, a^{t+1}) - Q^t(s^t, a^t) \right)}_{\delta^t} \tag{11}$$

4. On doit avoir $\alpha = O\left(\frac{1}{t}\right)_{t \to \infty}$ pour assurer la convergence de l'algorithme.

Il est nécessaire de connaître $(s^t, a^t, r^t, s^{t+1}, a^{t+1})$ pour pouvoir mettre à jour $Q^t(s^t, a^t)$, d'où le nom de cet algorithme : SARSA. La connaissance d'une estimation de $Q(s,a)$ permet de déterminer une politique ; il existe différentes méthodes pour ce faire qui sont abordée en SEC 3.2.3 p.7.

Le *Q-learning* est une variante de SARSA — un peu plus simple et rapide — dans laquelle on n'a pas besoin de connaître a^{t+1} pour mettre à jour $Q^t(s^t, a^t)$: a^{t+1} est remplacé par l'action choisie *si on suivait une politique gloutonne*. Le *Q-learning* apprend donc $Q^{\Pi_{greedy}}(s,a)$ tout en suivant une autre politique (*off-line*), alors que SARSA apprend $Q^{\Pi}(s,a)$ et Π tout en suivant et modifiant Π (*on-line*).

$$Q^{t+1}(s,a) \quad \leftarrow \quad Q^t(s^t, a^t) + \alpha . \left(r^t + \gamma . \max_{a'} Q^t(s^{t+1}, a') - Q^t(s^t, a^t) \right) \tag{12}$$

3.2.3 Détermination d'une politique et conflit exploration–exploitation

Une fois les qualités des actions connues, différentes méthodes existent pour établir une politique ; elles se différencient de par leur façon d'arbitrer le compromis exploration–exploitation : en effet afin d'assurer la convergence de tous algorithmes présenté précédemment, il est nécessaire que tous les couples états-actions soient visités régulièrement. Il est donc nécessaire que l'agent *explore* son environnement au détriment de *l'exploitation* des bonnes actions.

Les politiques les plus usuelles sont appelées :

- gloutonne (*greedy*) : choisir la meilleure action. Cette politique naïve n'est pas applicable pendant l'apprentissage, car elle ne produit aucune exploration et conduit donc systématiquement à une politique très sous optimale.
- ε-gloutonne (ε-*greedy*) : choix aléatoire uniforme avec une probabilité ε, politique gloutonne sinon. C'est une correction simple et efficace de la politique précédente, et donc très utilisée.

$$\Pi_s(a) = \begin{cases} \frac{\varepsilon}{\text{Card}(\mathcal{A})} + (1-\varepsilon) & \text{, si } a = \text{argmax}_{a'}\ Q^t(s,a') \\ \frac{\varepsilon}{\text{Card}(\mathcal{A})} & \text{, sinon} \end{cases}$$

- *soft max* : probabilité donné par une fonction de Boltzman. Le paramètre β — *température inverse* — contrôle la propension à l'exploration ou à l'exploitation : pour $\beta = 0$ la politique est purement aléatoire et pour $\beta \to \infty$ elle devient gloutonne. Contrairement à la politique précédente qui produit une exploration "aveugle", cette politique explorera peu les actions connues pour être néfastes.

$$\Pi_s(a) = \frac{e^{\beta . Q^t(a)}}{\sum_{a' \in \mathcal{A}} e^{\beta . Q^t(a')}}$$

3.3 Pour aller plus loin : mieux exploiter l'expérience

Les méthodes de différences temporelles ont de bonnes propriétés mais sont assez lentes en pratique, car elle ne mettent à jour les informations que pour *un seul état* ou *une seule* paire état-action lors de chaque interaction avec l'environnement.

Ceci fait qu'un grand nombre d'épisodes est nécessaire pour "propager la valeur" depuis un point de récompense ou de punition vers l'ensemble des états et actions. Différentes méthodes permettent de mettre à jour les informations pour plusieurs états ou une paires état-action.

On peut bien sûr essayer d'apprendre les fonctions de récompense et de transition, c'est-à-dire un *modèle de l'environnement*, ce qui est à la fois potentiellement très coûteux et très puissant — cf. 3.3.2 P.8 :architectures Dyna — mais aussi simplement se souvenir dans quelle mesure les états ou paires état-action sont "responsables" de la situation présente et les mettre à jour en conséquence — cf 3.3.2 P.8 : traces d'éligibilité.

Pour bien comprendre ces deux méthodes il sera bon de se référer au Sutton&Barto, les deux sections suivantes ne sont qu'une très (et trop !) rapide introduction.

3.3.1 Traces d'éligibilité

Les traces d'éligibilité sont une information scalaire que l'on associe à chaque état (si l'on se base sur une fonction de valeur) ou paire état-action (si l'on se base sur une fonction qualité) et qui représente dans quelle mesure l'information correspondante doit être mise à jour lors de l'interaction courante avec l'environnement.

En effet, quelque soit soit la méthode de différence temporelle utilisée, lors de chaque interaction (s,a,r,s') avec l'environnement, une certaine quantité X (Q ou V) est mise à jour suivant une formule du type :

$$X(s,a) \quad \leftarrow \quad X(s,a) + \alpha.\delta(X,s,a,r,s') \tag{13}$$

L'utilisation des traces d'éligibilités revient à "répartir" cette mise à jour sur tous les $X(s^*,a^*)$ proportionnellement à une pondération $e(s^*,a^*)$. Cette pondération — la trace d'éligibilité — s'atténue avec le temps de manière exponentielle de facteur λ et est réactivé à chaque visite réelle de l'état ou de la paire. Ceci donne la règle de mise à jour suivante :

$$e(s^*,a^*) \quad \leftarrow \quad \gamma.\lambda.e(s^*,a^*), \quad \forall (s^*,a^*) \in \mathcal{S} \times \mathcal{A} \tag{14}$$
$$e(s,a) \quad \leftarrow \quad e(s,a) + 1 \tag{15}$$
$$X(s^*,a^*) \quad \leftarrow \quad X(s^*,a^*) + \alpha.e(s^*,a^*).\delta(X,s,a,r,s'), \quad \forall (s^*,a^*) \in \mathcal{S} \times \mathcal{A} \tag{16}$$

On appelle ces algorithmes TD(λ), SARSA(λ), Q(λ)... Pour $\lambda = 0$ on retrouve les algorithmes TD, SARSA... alors que pour $\lambda = 1$ la trace d'éligibilité compte simplement le nombre de passages par l'état ou la paire au cours de l'épisode.

3.3.2 Architectures Dyna

Les architectures Dyna sont une famille générale de méthodes d'apprentissage par renforcement indirect (*model-based*), dont la structure est illustrée FIG 2 p.8.

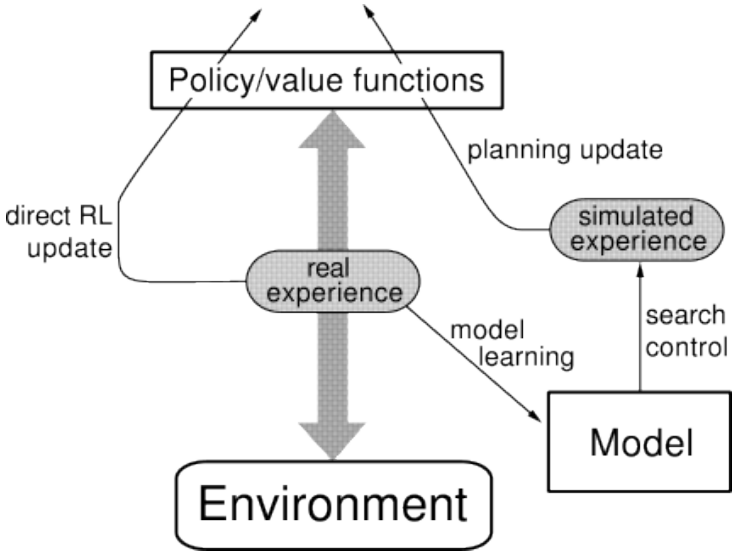

FIGURE 2 – Schéma général des architectures Dyna

L'idée au cœur de ces méthodes est celle d'un double apprentissage : d'une part on apprend un modèle du monde à partir de son expérience, et d'autre part on utilise ce modèle du monde pour apprendre la fonction de valeur et une politique en simulant des expériences dans ce modèle.

Parmi elles on peut citer Dyna-Q et Dyna-PI : la première effectue simule des étapes de *Q-learning* dans le modèle du monde, alors que la seconde résout le MDP constitué par le modèle du monde en utilisant P.I.

Ces méthodes ont l'avantage d'apprendre beaucoup plus rapidement que les méthodes directes, en effectuant plusieurs étapes de mise à jour des valeurs pour une étape d'expérience réelle. Elles permettent également de faire appel à des méthodes de programmation dynamique pour résoudre partiellement ou totalement le MDP constitué par le modèle du monde appris, et agir dans le MDP réel comme s'il était le MDP appris.