Apprentissage par renforcement développemental pour la robotique autonome

Luc Sarzyniec

Université Henri Poincaré

Master 2 Recherche, Reconnaissance Apprentissage Raisonnement

Rapport de stage recherche

Nancy-Université
Université Henri Poincaré

Maîtres de stage : Olivier Buffet et Alain Dutech

Groupe de recherche : MAIA

Apprentissage par renforcement développemental pour la robotique autonome.

Luc Sarzyniec

Laboratoire Lorrain de Recherche Informatique
615 r. du Jardin Botanique
54600 Villers-lès-Nancy

Nancy, le 30 août 2010

REMERCIMENTS

Je tiens à remercier l'équipe MAIA de m'avoir accueilli durant mon stage et formé à l'utilisation du robot Khepera3. Je tiens plus particulièrement à remercier mes deux encadrants de stage Alain Dutech et Olivier Buffet qui ont toujours été disposés à m'aider dans ma démarche de recherche.

Résumé

J'ai effectué mon stage dans un laboratoire de recherche, le LORIA. J'ai travaillé au sein l'équipe de recherche MAIA qui a pour principal thème de recherche l'étude des processus de décisions intelligents.

La problématique du stage porte sur l'apprentissage par renforcement appliqué à la robotique et plus particulièrement, l'apprentissage d'une tâche complexe divisée en plusieurs tâches plus simples.

Table des matières

Chapitre 1

Introduction

1.1 Cadre du stage

1.1.1 LORIA

Mon stage s'est déroulé au Laboratoire Lorrain de Recherche en Informatique et ses Applications (LORIA) qui est une Unité Mixte de Recherche (UMR7503) commune aux établissements suivants :

- CNRS, Centre National de Recherche Scientifique
- INPL, Institut National Polytechnique de Lorraine
- INRIA, Institut National de Recherche en Informatique et en Automatique
- UHP, Université Henri Poincaré, Nancy 1
- Nancy 2, Université Nancy 2

Le LORIA est situé à Villers-lès-Nancy, à proximité de la faculté de sciences.

La création de cette unité mixte de recherche a été officialisée le 19 décembre 1997 par la signature du contrat quadriennal avec le Ministère de l'Education Nationale, de la Recherche et de la Technologie et par une convention entre les cinq partenaires. Cette unité, renouvelée en 2001, succède ainsi au CRIN (Centre de Recherche en Informatique de Nancy), et associe les équipes communes entre celui-ci et le centre de recherche INRIA Nancy - Grand Est.

Le LORIA est un Laboratoire de plus de 450 personnes parmi lesquelles

- 150 chercheurs et enseignants-chercheurs
- un tiers de doctorants et post doctorants
- des ingénieurs, techniciens et personnels administratifs

organisé en équipes de recherche et services de soutien à la recherche.

Les chercheurs et enseignants-chercheurs sont distribués dans 30 équipes de recherches. Les équipes travaillent sur des thèmes de recherches très variés allant de

FIGURE 1.1 – Entrée principale du LORIA

systèmes de développement d'intelligence économique aux systèmes d'automatisation de la preuve, de reconaissance de forme ou encore d'intelligence artificielle.

1.1.2 MAIA

Mon stage s'est déroulé au sein de l'équipe MAchine Intelligente Autonome (MAIA), mes encadrants étaient Alain Dutech et Olivier Buffet.

L'équipe MAIA a pour principal thème de recherche l'étude des processus de décisions intelligents, notamment dans les systèmes multi-agents et les systèmes de décisions soumis à l'incertitude. Notre groupe s'intéresse à la modélisation, la construction et la simulation de systèmes composés d'un ou plusieurs agents. Un agent peut prendre différentes formes : physique (comme un robot mobile, ou un robot d'anesthésie), et ou logiciel (softbot, par exemple un assistant intelligent) ou biologique (humains, insectes sociaux). La modélisisation et l'interprétion des comportements collectifs ou individuels a des applications dans le domaine médical (Télé-diagnostic, Personnes Agées), la biologie du comportement (Arthropodes sociaux) ou encore l'assistance aux utilisateurs et aux usagers (Tourisme assisté par ordinateur avec le Cycab).

L'équipe est composée de onze membres permanents, et d'environs le même nombre de doctorants.

1.2 Objectifs du stage

Lors d'un apprentissage par renforcement on considèrere un agent au sein d'un environnement et qui doit prendre des décisions en fonction de sa situation actuelle (état) dans l'environnement. L'agent peut interragir avec l'environnement par le biais d'actions et l'environnement lui rend une récompense (positive, négative ou nulle) en fonction de celles-ci. Cette classe de problème constitue aujourd'hui un domaine très vaste et très ouvert dans la recherche, elle impacte un grand nombre de projets et pourra compter un grand nombres d'applications dans la vie de tous les jours.

Les principales motivations de ce stage sont l'apprentissage par renforcement appliqué à la robotique et plus particulièrement à la robotique développementale. L'équipe MAIA a par le passé étudié l'apprentissage d'une tâche complexe en la divisant en plusieurs tâches simples devenant de plus en plus complexes. Cette méthode d'apprentissage semble plus probante que l'apprentissage direct d'une tâche complexe [Buffet et al., 2001].

L'objectif de ce stage est de faire évoluer les capacités de perception et de mouvement d'un robot en parallèle pendant une tâche d'apprentissage.

L'apprentissage par renforcement appliqué à la robotique reste un problème encore ouvert aujourd'hui pour plusieurs raisons : la difficulté d'effectuer des simulations longues (batteries, accumulation d'erreurs), l'inexactitude des données retournés par les capteurs, les biais dans les consignes envoyés au robot et dans le type d'environnement dans lequel nous aurons à travailler (serveur/client), la latence qui fausse les consignes/résultats envoyées et reçu par les robots. Ces problèmes liés au cadre robotique influent grandement sur l'apprentissage par renforcement de tâches plus ou moins complexes.

En plus de ces difficultés purement techniques liées au cadre robotique, d'autres problèmes (plus théoriques de l'apprentissage par renforcement) sont à traiter, notamment :

- pouvoir traiter des signaux continus en entrée (capteurs) et en sortie (moteurs) ;
- pouvoir gérer un environnement d'une grande richesse ; la question posée est de savoir comment le robot va pouvoir "choisir" quels sont les éléments de son environnement qui sont pertinent pour qu'il puisse réaliser sa tâche ;
- pouvoir apprendre malgré des signaux de récompense peu fréquents ; si la tâche à apprendre est complexe, l'apprentissage par renforcement, qui s'appuie sur une exploration aléatoire de l'environnement, ne converge que très lentement ;

Dans le but d'essayer de pallier un certain nombre de problèmes impactés par la mise en oeuvre de l'apprentissage par renforcement de tâches complexes en robotique, nous avons décidé d'emprunter un certain nombre de concepts propres au domaine de la robotique développementale. Cependant, même si cette approche nous permet de nous

abstraire d'un certain nombre de difficultés, de nouvelles problématiques apparaissent :

- le transfert des connaissances et/ou capacités acquises dans un cadre simple vers un cadre plus complexe ;
- le changement de but à atteindre au cours de l'apprentissage ;

Un exemple d'évolution des capacités du robot pourrait être, par exemple, d'augmenter ou de raffiner les actions qu'un robot peut effectuer au cours d'un apprentissage. Un autre exemple d'évolution pourrait être le changement de but d'un robot pendant sont apprentissage (se rapprocher d'une couleur, puis suivre des flèches de cette couleur (s'en éloigner)). Le problème ainsi posé pourrait donc être résumé de la façon suivante : lorsque qu'un robot a acquis des connaissances dans une configuration donnée (tâches, capacités perceptives, capacités d'actions), comment transférer et utiliser ces connaissances dans une nouvelle configuration ?

Pendant le stage nous avons travaillé avec différentes technologies auxquelles nous n'avions pas été familiarisées : un robot Khepera3 muni d'une caméra KorwlCam et l'environnement de développement URBI (voir partie 4.2.3). Une partie du stage a donc consisté à apprendre à les manipuler et à en évaluer les limites.

Nous nous pencherons dans une première partie sur les outils théoriques existants utilisés dans le cadre du projet, puis sur l'approche théorique du problème envisagé enfin sur la partie expérimentations et résultats des tests.

Nous tenons aussi à rappeler que ce rapport a été réalisé suite à un stage effectué dans un cadre scientifique. Il répond donc aux exigences de présentation de celui-ci, cependant les aspects traités relatifs à l'ingénierie logicielle seront exposés dans des sous-parties.

Chapitre 2

Contexte théorique

2.1 Introduction

Nous commencerons par exposer les principes de l'apprentissage par renforcement, puis nous définirons l'approche développementale de l'apprentissage que nous souhaitons appliquer à la robotique. Enfin nous introduirons brièvement les outils théoriques que nous allons utiliser pour répondre à la problématique posée.

2.1.1 Principes de l'apprentissage par renforcement

Une solution utilisée couramment pour mettre en oeuvre un apprentissage par renforcement [Bellman, 1957] est de considérer un agent au sein d'un environnement et qui doit prendre des décisions en fonction de sa situation actuelle (état) dans l'environnement. L'agent peut interragir avec l'environnement par le biais d'actions et l'environnement lui rend une récompense (positive, négative ou nulle) en fonction de celles-ci.

Plus formellement, l'apprentissage par renforcement fait référence à une classe de problèmes d'apprentissage automatique, dont le but est d'apprendre, à partir d'expériences, ce qu'il convient de faire en différentes situations, de façon à optimiser une récompense numérique au cours du temps.

L'agent va donc chercher à maximiser les récompenses obtenues en effectuant différentes expériences et types d'expériences (exploration, suivi d'une politique, ...) au sein de l'environnement. La nature et le choix des expériences à effectuer à un moment précis de l'apprentissage varie en fonction de l'algorithme d'apprentissage choisi.

Comme nous l'avons déjà présenté (voir 1.2), l'apprentissage par renforcement d'une tâche complexe n'est, dans la pratique, pas toujours possible. Nous avons donc choisi de nous inspirer des concepts d'apprentissage progressif inspiré par des concepts de la psychologie du développement.

2.1.2 Approche développementale de l'apprentissage

Les principes de l'approche développementale de l'apprentissage sont inspirés de l'observation du comportement humain et plus particulièrement celui des très jeunes enfants (voir figure 2.1). Cette méthode s'inspire des mécanismes de développement présents chez les enfants, et identifiés par la psychologie du développement, les neurosciences cognitives, et la linguistique cognitive. La partie qui intéresse les chercheurs dans ce domaine est le mécanisme d'apprentissage où l'enfant *prend progressivement conscience* du monde qui l'entoure, de ses propres capacités et appréhende l'environnement qui l'entoure tout en continuant de faire évoluer ses capacités.

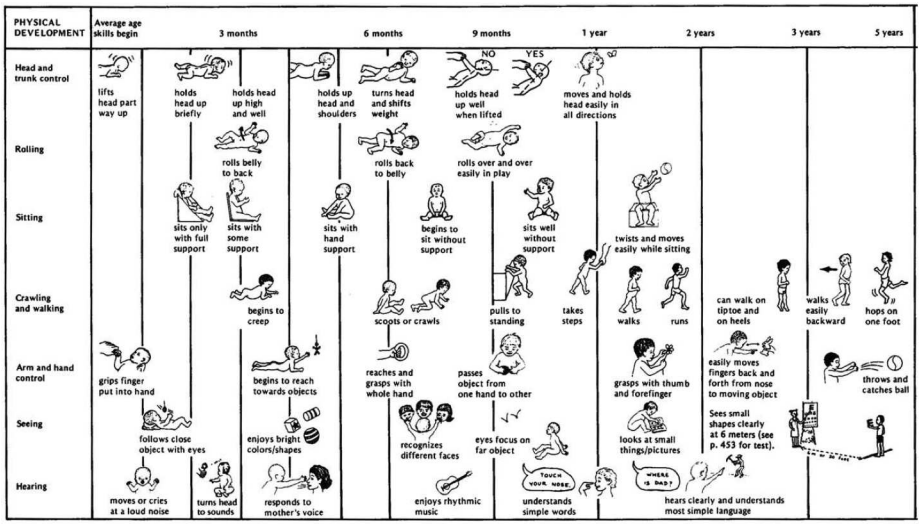

FIGURE 2.1 – Evolutions constatés chez l'enfant dans ses premières années

Par exemple, quelques grands défis de l'apprentissage développemental et social sont :

- comment découvrir et apprendre à contrôler un corps inconnu et possiblement changeant ? Comment apprendre à utiliser ce corps pour manipuler des objets ?
- comment apprendre de nouveaux savoir-faire sensorimoteurs au cours d'interactions sociales ?
- comment apprendre les rudiments du langage ?
- comment permettre des interactions sociales naturelles et intuitives avec d'autres personnes personnes, et comment ces interactions peuvent-elles changer la nature du problème ?

2.1.3 Outils utilisés

Pendant le stage, nous avons travaillé sur l'apprentissage par renforcement. Naturellement, nous avons employé l'un de ses formalisme mathématique les plus communs : les processus de décisions Markoviens (MDP). Il existe un grand nombre d'algorithmes permettant de mettre en place l'apprentissage par renforcement en se basant sur le cadre Markovien. Nous avons choisi d'utiliser l'algorithme Q-Learning. Les processus de décisions Markoviens, qui sont particulièrement adaptés au travail dans un environnement à espace d'état/actions discret, le sont un peu moins lorsque l'on travaille dans un environnement continu comme celui de la robotique. Nous avons donc procédés à différentes adaptations des algorithmes utilisés comme (1) l'approximation de fonctions pour tenter d'approximer la fonction d'action/valeur $Q(s, a)$ (voir section 2.3) où s est continu et (2) l'ajout de traces d'éligibilités lors de l'apprentissage pour accélérer celui-ci.

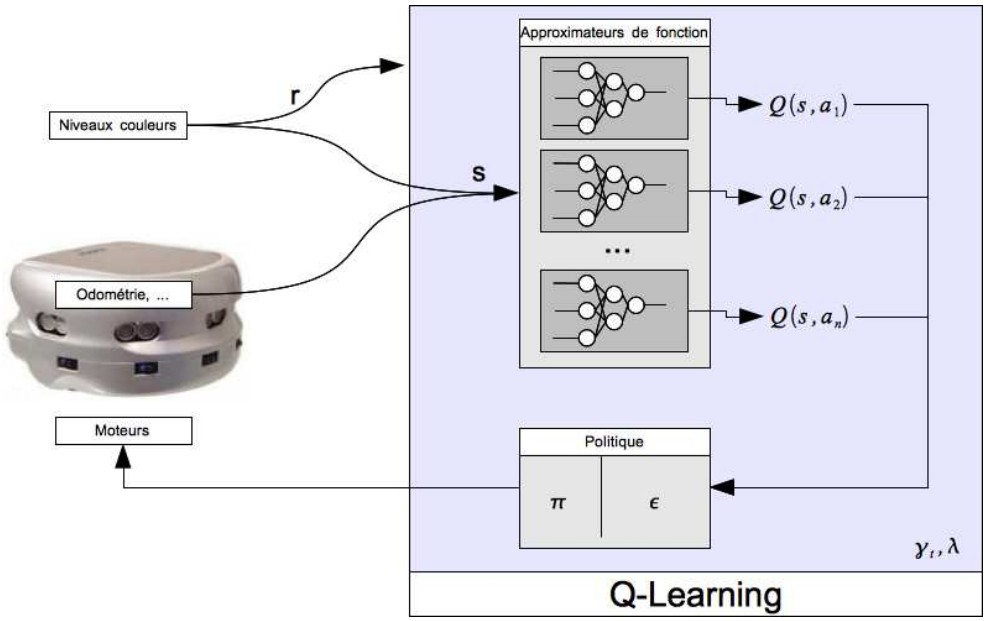

FIGURE 2.2 – Configuration de travail

Nous commencerons par présenter les processus de décision de Markov (section 2.2), puis nous introduirons les concepts d'un algorithme d'apprentissage par renforcement, le Q-Learning dans la section 2.3, nous expliquerons le fonctionnement de l'approximateur de fonction que nous avons utilisés (section 2.4). Enfin nous présenterons un moyen d'accélérer l'apprentissage par renforcement, les traces d'éligibilités, dans la section 2.5.

2.2 Processus de décision de Markov

2.2.1 Formalisme

Un processus de décision de Markov (MDP) est un modèle stochastique qui peut être vu comme une chaîne de Markov contrôlée, c'est à dire à laquelle on ajoute une composante décisionnelle. C'est un outil mathématique qui permet de formaliser l'apprentissage par renforcement. Un MDP vérifie la propriété de Markov qui peut être résumée de la manière suivante : prédire le futur à partir du présent est tout aussi efficace que le prédire en possédant des informations concernant le passé.

Plus formellement la propriété de Markov peut être définie de la manière suivante :

$$\text{Soient } X_0 \dots X_n \text{ des variables aléatoires réelles, nous avons donc}$$
$$\forall n \geq 0, \forall (i_0, \dots, i_{n-1}, i, j) \in E^{n+2}$$
$$P(X_{n+1} = j | X_0 = i_0, \dots, X_{n-1} = i_{n-1}, X_n = i) = P(X_{n+1} = j | X_n = i)$$

Un processus de décision de Markov est un quadruplet $\langle S, A, T, R \rangle$ qui peut être défini de la façon suivante :

- $S = s_0, \dots, s_{|S|}$ est l'ensemble fini discret des états possibles du système à contrôler ;
- $A = a_0, \dots, s_{|A|}$ est l'ensemble fini discret des actions que l'on peut effectuer pour contrôler le système ;
- $T : S \times A \times S \to [0; 1]$ est la fonction de transition du système T et donne la probabilité que le système passe d'un état à un autre en ayant choisi une action donnée ;
- $R : S \times A \times S \to \mathbb{R}$ est la fonction de récompense, elle indique la valeur réelle obtenue lorsque l'on effectue l'action a dans l'état s pour arriver dans l'état s' ;

Dans le cadre des MDP on appelle politique $\pi : S \to A$ une fonction qui indique pour chaque état quelle est l'action à effectuer. Il s'agit là d'une politique déterministe où, contrairement à une politique stochastique, il n'y a pas d'ambiguïté sur l'action à effectuer.

Pour plus d'informations sur les Processus de décision de Markov, le lecteur peut se référer à [Buffet, 2003], [Puterman, 1994].

2.2.2 Exemple : La modélisation dans le projet ratmaze

Dans ce projet, nous avons choisi d'implanter un solver dans un environnement ou un rat doit retrouver un morceau de fromage dans un labyrinthe. Le programme en lui-même a été réalisé en langage C. Ce projet nous a permis de mieux comprendre les mécanismes et les enjeux de l'apprentissage par renforcement.

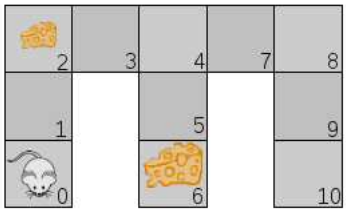

FIGURE 2.3 – Environnement de travail du projet Ratmaze

L'environnement de travail sur lequel nous avons effectué les tests est un labyrinthe de 10 cases comprenant deux récompenses (voir figure 2.3). Le processus de décision Markovien associé au labyrinthe est établi de la façon à ce qu'à chaque case corresponde un état.

$$S = \Big\{ ⓪, ①, ②, \dots \Big\}$$

Il y a quatres actions de déplacements possibles au sein du labyrinthe.

$$A = \begin{Bmatrix} UP \\ DOWN \\ LEFT \\ RIGHT \end{Bmatrix}$$

Lors d'un mouvement, le rat a 90% de chance de se retrouver dans l'état souhaité et 10% de chances de se retrouver dans une des cases (atteignables) adjacentes. Si le mouvement n'est pas possible, le rat reste sur la même case.

$$T = \begin{Bmatrix} \dots, \\ (②, UP, ②) \rightarrow 1, \\ (②, DOWN, ①) \rightarrow 0.9, \\ (②, DOWN, ③) \rightarrow 0.1, \\ (②, LEFT, ②) \rightarrow 1, \\ (②, RIGHT, ③) \rightarrow 0.9, \\ (②, RIGHT, ①) \rightarrow 0.1, \\ \dots \end{Bmatrix}$$

La récompense associée à la case ⑥ est cinq fois supérieure à celle en case ②. Les récompenses associés aux autres cases sont nulles.

$$R = \begin{Bmatrix} \dots, \\ (*, *, ① \rightarrow 0), \\ (*, *, ② \rightarrow 1), \\ (*, *, ⑥ \rightarrow 5), \\ \dots \end{Bmatrix}$$

Le rat débute son exploration sur la case ⓪.

$$s_0 = ⓪$$

2.3 Un algorithme d'apprentissage par renforcement : Q-Learning

2.3.1 Formalisme

Le Q-Learning [Barto et al., 1989] est un algorithme d'apprentissage par renforcement. L'agent évolue dans un environnement qu'il ne connait que partiellement (il ne connait pas la matrice de transition T et la fonction de récompense R), ce qui nous rapproche du cadre expérimental du stage : l'apprentissage par renforcement sur un robot. Dans cet algorithme on apprend en faisant des simulations suivant une stratégie exploratoire

Dans cet algorithme, on travaille avec une fonction de valeur à apprendre $Q(s, a)$ qui associe à un couple état/action une valeur numérique estimant l'intérêt à faire cette action dans cet état. L'intérêt de cette fonction est de donner un critère de choix d'une action $a \in A$ dans un état $s \in S$ donné. On notera π la politique qui à chaque état $s \in S$ associe une action $a \in A$ à effectuer. Le principe est que, pour toute nouvelle expérience $(s, a) \rightarrow (s', r)$, on met à jour Q en effectuant un compromis entre l'ancienne estimation et la nouvelle expérience. Un coefficient α_t est introduit pour permettre de régler la portée de ce compromis (de quelle façon on tiens compte des nouvelles expériences dans l'apprentissage. Dans l'algorithme, on cherche à maximiser Q selon le critère $E[\sum_{t=0}^{\infty} \gamma^t r_t]$. La nouvelle expérience laisse espérer un gain de $r + \gamma * \max_{a' \in A} Q(s', a')$. La fonction de mise à jour de Q s'écrit donc :

$$Q(s,a) \leftarrow \underbrace{(1-\alpha_t) * Q(s,a)}_{\text{estimation précédente}} + \underbrace{\alpha_t * \left[r + \gamma * \max_{a' \in A Q(s',a')} \right]}_{\text{nouvelle expérience}}$$

.

L'algorithme 1 décrit le fonctionnement de base du Q-Learning.

Pour tout état $s \in S$, on notera l'espérance de gain en cet état : $V(s) = \max_{a \in A} Q(s, a)$. La solution optimale au système alors considéré est notée V^* (à laquelle est associée une unique fonction Q^*).

Le facteur $\alpha_t \in [0; 1[$ est le taux d'apprentissage qui détermine dans quelle mesure les nouvelles informations acquises vont remplacer les anciennes. Un taux d'apprentissage de 0 ne fera rien apprendre à l'agent tandis qu'un taux d'apprentissage à 1 ne lui fera considérer que l'information la plus récente.

Algorithm 1 Algorithme Q-Learning

$Q(s, a) \in \mathbb{R}$ arbitrairement initialisé pour tout $(s, a) \in S \times A$
Initialiser s
for all pas de temps **do**
  Choisir a de manière aléatoire
  Effectuer l'action a ; observer r, s'
  $Q(s, a) \leftarrow (1 - \alpha_t) * Q(s, a) + \alpha_t * [r + \gamma * \max_{a' \in A} Q(s', a')]$
  $s \leftarrow s'$
end for

Le facteur $\gamma \in [0; 1]$ permet de déterminer l'importance des récompenses futures dans l'apprentissage. Dans la pratique α_t tends vers 0 au cours de l'apprentissage. Un facteur de 0 fera considérer que les dernières récompenses à l'agent tandis qu'un facteur proche de 1 lui fera considérer les récompenses à long terme autant qu'à court terme.

Si on connait la valeur de Q optimale (notée Q^*), une politique (π gloutonne) optimale en un état s est $\pi^*(s) \in argmax_{a \in A} Q^*(s, a)$.

Remarque : cette version de l'algorithme est adaptée à un environnement à espace d'états discrets.

2.3.2 Exemple : La résolution dans le projet Ratmaze

Dans le projet Ratmaze, nous avons implanté plusieurs algorithmes d'apprentissage par renforcement : l'algorithme Value Iteration [Bellman, 1957] (calcul de V^* par programmation dynamique), l'algorithme Policy Iteration [Howard, 1960] (calcul de π^* par programmation dynamique) et le Q-Learning. Nous ne nous intéresserons dans cette section qu'aux tests effectués sur ce dernier.

Nous avons dans le cadre du mini-projet essayé de faire varier différents paramètres dans les algorithmes utilisés et constaté en quoi ces valeurs influaient sur le résultat.

Nous avons constaté qu'à partir d'un certain nombre d'itérations, la politique optimale est atteinte pour chaque état. Ìl faut bien sûr que chaque état ait été visité un nombre de fois suffisant (l'action effectuée à partir d'un état source étant aléatoire, cela peut prendre du temps). Théoriquement pour obtenir une fonction de valeur optimale, il faudrait réaliser une infinité d'itérations.

Pour pallier en partie à ce problème (phase d'exploration des fois trop courte pour avoir rencontré tous les états un nombre suffisant de fois), nous avons introduit un facteur ϵ qui détermine le type d'action à effectuer lors d'un passage dans la boucle (choisir la meilleure action ou choisir une action aléatoire). Faire varier ce facteur nous

permet de privilégier soit l'exploration dans le labyrinthe soit la recherche de la politique optimale.

Nous avons très vite constaté que choisir la meilleure action au cours des premières itérations de l'algorithme n'avait pas beaucoup de sens, étant donné que les Q-valeurs n'étaient pas encore connues/significatives pour la plupart des états. Nous avons donc choisi d'explorer aléatoirement le labyrinthe pendant les N_{seuil} premières itérations puis de laisser le choix entre l'exploration ou la meilleure politique (utilisation du facteur ϵ).

Un problème subsistait lorsque le rat avait atteint l'état but. Même en utilisant le facteur ϵ, son exploration se limitai aux cases adjacentes au but. Nous avons donc décidé, lors de l'atteinte du but, de "téléporter" le rat aléatoirement dans le labyrinthe.

Pour faire nos mesures, nous avons choisi de lancer une estimation de cumul de gains ($R_g = \sum_{0}^{T-1} \frac{r_t(S_t)}{T}$) en estimant une politique à un moment précis de l'apprentissage. Le gain cumulé est calculé sur un nombre N_{cumul} d'itérations en suivant la politique courante. La fonction d'évaluation est lancée tous les N_{pas} itérations de l'algorithme principal.

L'algorithme 2 constitue l'algorithme final du Q-Learning et l'algorithme 3 la procédure d'évaluation.

Algorithm 2 Algorithme Q-Learning

$Q(s, a) \in \mathbb{R}$ arbitrairement initialisé pour tout $(s, a) \in S \times A$
Initialiser s
for $i = 0$ to N_{iter} **do**
  $r \leftarrow random([0, 1])$
  if $(i < N_{seuil})||(r \geq \epsilon)$ **then**
    $a \leftarrow random(UP, DOWN, LEFT, RIGHT)$
  else
    $a \leftarrow \max_{a' \in A} Q(s, a')$
  end if
  if $i \mod N_{pas}$ **then**
    $\forall st \in S, \pi(st) \leftarrow argmax_{act \in A} Q(st, act)$ {Définition de la politique}
    $evaluer(\pi)$ {Voir algorithme 3}
  end if
  Effectuer l'action a ; observer r, s'
  $Q(s, a) \leftarrow (1 - \alpha_t) * Q(s, a) + \alpha_t * [r + \gamma * \max_{a' \in A} Q(s', a')]$
  $s \leftarrow s'$
end for

Algorithm 3 Algorithme d'évaluation de Q-Learning

1: $sum \leftarrow 0$
2: **for all** $s \in S$ **do**
3: $s_{cur} \leftarrow s$
4: **for** $i = 0$ to N_{cumul} **do**
5: $sum \leftarrow sum + r_t(s_{cur}, Act(s, \pi(s)))$
{ $Act(s, a) = argmax_{a \in A}(max_{s' \in S} T(s, a, s'))$}
6: $s_{cur} \leftarrow Act(s, \pi(s))$
7: **end for**
8: **end for**
9: $sum \leftarrow sum/(|S| * N_{cumul})$

La figure 2.4 illustre le comportement de l'algorithme sur le labyrinthe avec $N_{iter} = 1600$, $Nseuil = 200$, $N_{cumul} = 100$ et N_{pas}=20. Nous pourrons constater que jusqu'à ce que le seuil N_{seuil} soit atteint (lorsque le choix de l'action à effectuer est totalement aléatoire), la croissance de l'espérance de gain est linéaire et qu'elle semble suivre une croissance de type $(1 - e^{-x})$ en utilisant le facteur ϵ.

2.4 Cas des espaces d'état continus : utilisation d'un approximateur de fonctions

Dans le cadre du stage, nous avons été amené à travailler avec des approximateurs de fonctions pour permettre d'effectuer un apprentissage par renforcement se basant sur l'algorithme de Q-Learning dans un espace d'état continu (contraintes de la robotique).

2.4.1 Introduction aux perceptrons multi-couches

En effet, si l'on se place dans un environnement à espace d'états continu, il est très difficile de tenter d'approximer $Q(s, a), s \in S, a \in A$ optimal car S n'est pas fini. Pour pallier à ce problème, il est possible d'avoir recours à un approximateur de fonction pour tenter d'approximer la fonction de valeur Q. Il existe plusieurs manières de procéder et plusieurs types d'approximateurs, l'une consiste à essayer d'approximer $V(s) = \max_{a \in A} Q(s, a)$, une autre (celle que l'on a choisi) est d'affecter à chaque action $a \in A$ un approximateur $L(a)$ qui lui est propre. Nous avons choisi d'utiliser un approximateur de fonction d'un type bien particulier, les réseaux de neurones artificiels.

Un réseau de neurones artificiels [Lettvin et al., 1959] est un modèle de calcul dont la conception est très schématiquement inspirée du fonctionnement des neurones biologiques. Le neurone formel est conçu comme un automate doté d'une fonction de

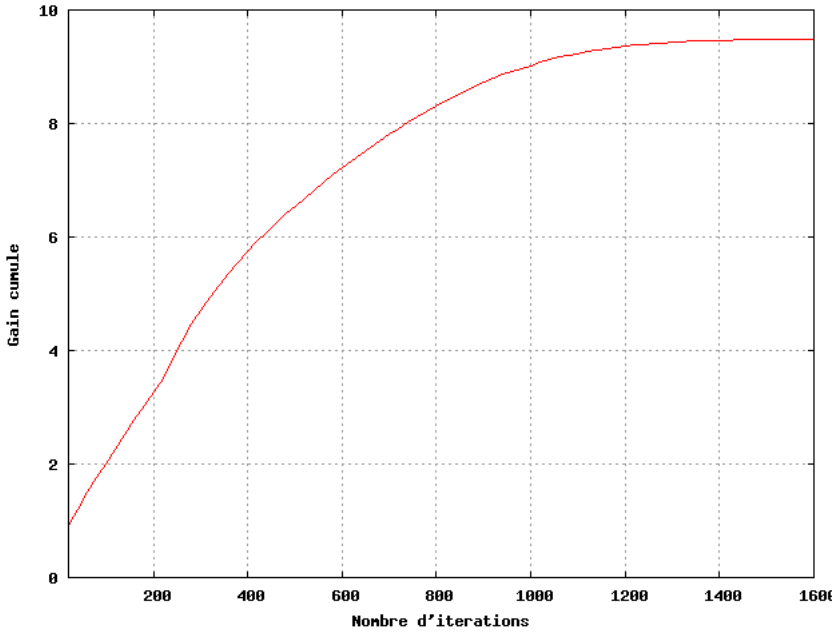

FIGURE 2.4 – Évolution du gain cumulé périodique

transfert qui transforme ses entrées en sortie selon des règles précises.

Très synthétiquement, par exemple, dans un neurone (voir figure 2.5), on calcule la sortie du réseau en effectuant le calcul $f(\vec{x}) = \phi(\vec{x}.\vec{w})$, où $\vec{x}$ est le vecteur d'entrée, ϕ la fonction de transfert et $\vec{w}$ les poids du neurone. Lors de l'apprentissage, le vecteur $\vec{w}$ sera mis à jour de façon à corriger les erreurs commises lors de l'approximation.

Ces neurones sont par ailleurs associés en réseaux dont la topologie et les connexions est variable. En général, les neurones sont organisés en couches, on distinguera plus particulièrement la couche d'entrée, les couches cachées et la couche de sortie (voir figure 2.6). On conviendra de numéroter les couches en partant de la couche d'entrée (numérotée 1) jusqu'à la couche de sortie.

Dans le cadre du stage, nous avons avons travaillés avec une classe de réseaux de neurones particuliers : les perceptrons multicouches. De plus nous avons travaillés avec une topologie de réseaux bien particulière, en effet, nous ne comptons qu'un seul neurone sur la couche de sortie et tous les neurones d'une couche n sont "connectés" aux neurones de la couche $n + 1$. Pour finir, nous avons travaillé avec différents types de fonctions d'activations : la fonction identité ($I(x) = x$) pour le neurone de sortie, la fonction sigmoïde ($\sigma(x) = \frac{1}{1+e^{-x}}$) pour les autres neurones.

Nous allons dans la suite généraliser les méthodes de calcul des sorties d'un perceptron multicouche, puis expliquer de quelle façon on le fait apprendre à partir de ses

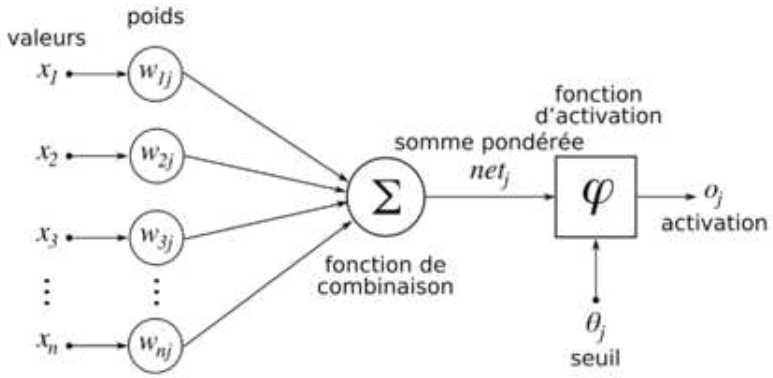

FIGURE 2.5 – Structure d'un neurone artificiel

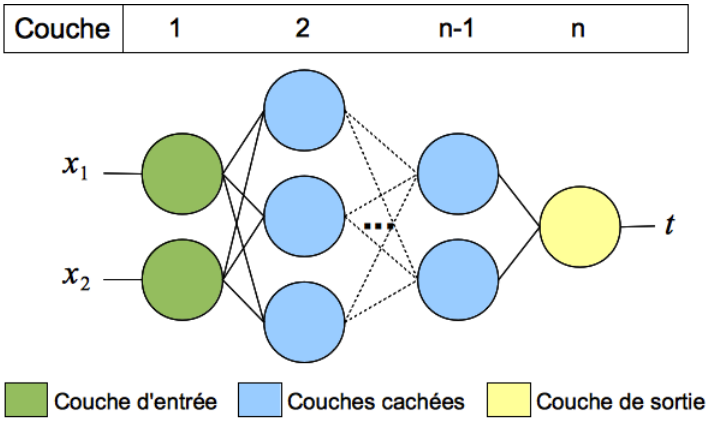

FIGURE 2.6 – Structuration en couches d'un réseau de neurones

erreurs.

Soit un réseau de neurones composé de N couches, nous décrirons par $w_{i,j}$ le poids connectant le neurone d'indice i, situé sur la couche n, au neurone d'indice j, situé sur la couche $n+1$. L'ensemble $In(k)$, le neurone d'indice k étant sur la couche n, correspond à l'ensemble des indices des neurones de la couche précédente $(n-1)$ connectés au neurone d'indice k. L'ensemble $Out(k)$, où k est sur la couche n, correspondant à l'ensemble des indices des neurones de la couche suivante $(n+1)$ qui sont connectés à k. La valeur de sortie de chaque neurone i est décrite par Φ_i. Enfin, on notera h_i le potentiel du neurone i, soit $h_i(\vec{x}) = \sum_{j \in In(k)} w_{j,i} * \Phi_j(\vec{x})$. Dans notre explication les fonctions d'activations des neurones sont des fonction sigmoïdes.

Nous avons donc :

$$\text{sur la couche d'entrée : } \Phi_1(\vec{x}) = \sigma\left[\sum_{j \in [0, dim(\vec{x})]} (w_{j,i} * x_j)\right],$$

$$\text{sur les autres couches : } \Phi_k(\vec{x}) = \sigma(h_k(\vec{x})),$$

$$\text{avec } \sigma(z) = \frac{1}{1+e^{-z}}.$$

Considérant un vecteur d'entrée $\vec{x}$ et une valeur de sortie attendue $y_{\vec{x}}$, on notera $t_{\vec{x}}$ la sortie produite par le réseau de neurones avec l'entrée x. L'erreur alors commise par le réseau de neurones est donc définie comme $E_{\vec{x}} = \frac{1}{2}(t_{\vec{x}} - y_{\vec{x}})^2$.

Pour permettre au réseau d'apprendre, il nous faut minimiser l'erreur commise lors de son approximation de $f(\vec{x}) = y_{\vec{x}}$. Il nous faudra donc calculer l'influence de chaque poids $w_{i,j}$ sur l'erreur $E_{\vec{x}}$, soit : $\nabla w_{i,j}(\vec{x}) = \frac{\delta E_{\vec{x}}}{\delta w_{i,j}}$. Nous calculerons donc la valeur de chaque $\nabla w_{i,j}(\vec{x})$ ce qui nous permettra d'effectuer une descente du gradient. Nous ne détaillerons pas ici toute la méthode de descente du gradient, mais seulement la méthode simplifiée de calcul que nous avons mis en place.

On calculera tout d'abord, pour chaque neurone k, un "gradient intermédiaire" $\vec{\nabla}_k(\vec{x})$, qui nous permettra de calculer chaque $\vec{\nabla} w_{i,k}(\vec{x}), i \in In(k)$:

$$\text{sur la couche de sortie : } \vec{\nabla}_z(\vec{x}) = \sigma'(h_z(\vec{x})) * (y_{\vec{x}} - t_{\vec{x}}),$$

$$\text{sur les couches intermédiaires : } \vec{\nabla}_i(\vec{x}) = \sigma'(h_i(\vec{x})) * \sum_{n \in Out(i)} \left(\vec{\nabla}_n(\vec{x}) * w_{i,n}\right).$$

On obtient ensuite la valeur de chaque $\vec{\nabla} w_{i,j}$ qui est calculée tel que suit :

$$\vec{\nabla} w_{i,j}(\vec{x}) = \vec{\nabla}_j(\vec{x}) * w_{i,j}$$

Il ne reste plus qu'à mettre à jour le poids qui y est associé à chaque $\vec{\nabla} w_{i,j}(\vec{x})$ dans le réseau de la façon suivante :

$$w_{i,j} \leftarrow w_{i,j} - \alpha_t * \vec{\nabla} w_{i,j}(\vec{x})$$

où α_t est le coefficient d'apprentissage.

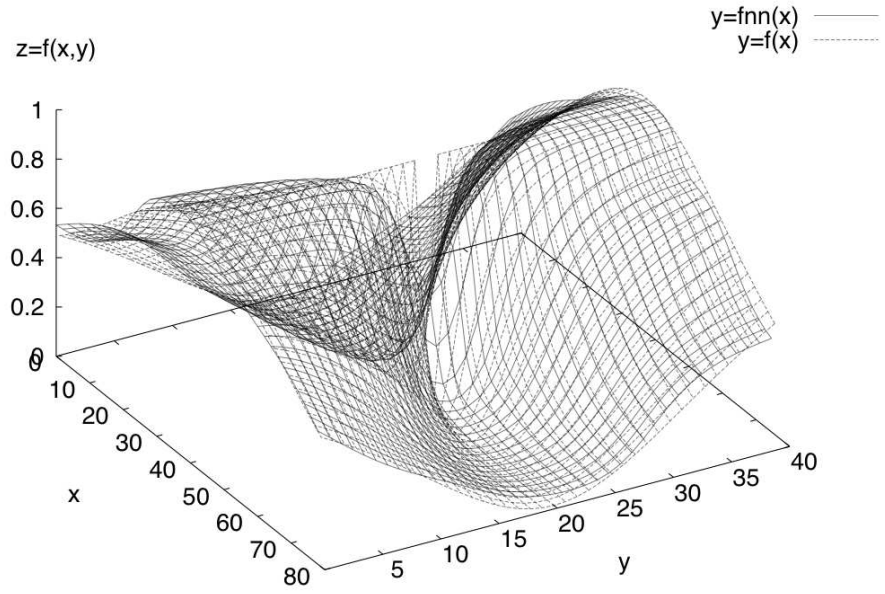

FIGURE 2.7 – Approximation du conoïde de Plücker avec notre réseau de neurones

2.4.2 Exemple : Approximateur de fonctions

Nous avons développé dans le cadre du stage, une bibliothèque permettant de créer des réseaux de neurones en C++ (pour des besoins de portabilité avec l'environnement utilisé par la suite). Nous n'avons pas utilisé de librairies externes.

Pour tester notre approximateur, nous avons tout d'abord essayé d'approximer le comportement d'un opérateur logique OU Exclusif, puis nous avons tenté d'approximer le conoïde de Plücker ($pl(x, y) = \frac{x^2-y^2}{x^2+y^2}$). La figure 2.7 montre nos résultats dans l'approximation du conoïde de Plücker (on travaille avec $\alpha_t = 0.1$).

Dans la suite, nous allons illustrer le fonctionnement de notre perceptron multicouche pour approximer le conoïde de Plücker. Nous travaillerons avec un réseau de neurones composé de trois neurones en entrée et un neurone en sortie. On travaille avec un vecteur d'entrée x de dimension 2 (voir figure 2.8). La fonction d'activation que nous utiliserons pour chaque neurone est la fonction sigmoïde (σ), sauf pour le dernier neurone ou l'on utilise la fonction identité (I). On notera y_i la sauvegarde de la dernière valeur de Φ_i qui a été calculée (dernier calcul de la sortie du réseau).

On détaillera, dans notre exemple, la mise à jour du poids $w_{1,2}$. On utilisera ici l'exemple $pl(\vec{x}) = y_{\vec{x}}$ pour l'apprentissage, considérant que la valeur de sortie du

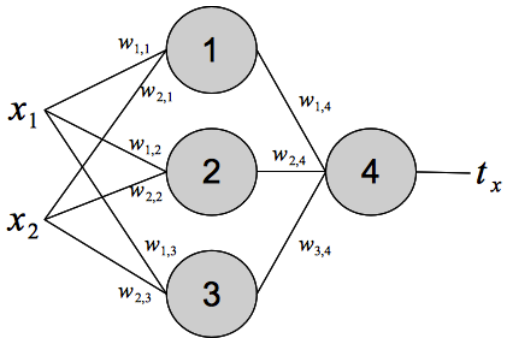

FIGURE 2.8 – Réseau de neurones utilisé dans l'exemple

réseau de neurones est $t_{\vec{x}}$. Pour simplifier les écritures, on nottera $\vec{\nabla} w_{i,j}(\vec{x})$ tel que $\vec{\nabla} w_{i,j}$ (resp. pour les $\vec{\nabla}_k(\vec{x})$).

On commence par calculer la valeur des "gradients intermédiaires" entrant en compte dans les calculs :

$$\vec{\nabla}_4 = I'(h_4) * (y_{\vec{x}} - t_{\vec{x}})$$
$$\text{puis } \vec{\nabla}_2 = \sigma'(h_2) * [(I'(h_4) * (y_{\vec{x}} - t_{\vec{x}})) * w_{2,4}]$$

On calcule ensuite la valeur de $\vec{\nabla} w_{1,2}$ de la façon suivante :

$$\vec{\nabla} w_{i,j} = [\sigma'(h_2) * (I'(h_4) * (y_{\vec{x}} - t_{\vec{x}})) * w_{2,4}] * w_{1,2}$$

D'où la mise à jour du poids $w_{1,2}$:

$$w_{1,2} \leftarrow w_{1,2} - [\alpha_t * \sigma'(h_2) * (I'(h_4) * (y_{\vec{x}} - t_{\vec{x}})) * w_{2,4} * w_{1,2}]$$
$$\text{où } \sigma'(z) = z * (1 - z),$$
$$I'(z) = 1,$$
$$h_2 = w_{1,2} * x_1 + w_{2,2} * x_2,$$
$$\text{et } h_4 = w_{1,4} * y_1 + w_{2,4} * y_2 + w_{3,4} * y_3.$$

Pour des raisons pratiques, certaines fonctionnalités ont été ajoutés à notre librairie comme la sauvegarde et le chargement d'une configuration (topologie et valeurs des poids) par exemple.

2.4.3 Utilisation d'approximateur de fonctions dans notre algorithme de Q-Learning

L'approximateur de fonction (réseau de neurones) présenté ci-dessus nous permet maintenant de tenter d'approximer $Q(s, a)$ avec s continu. L'algorithme 4 présente les

modifications apportés à l'algorithme initial de Q-Learning pour le faire fonctionner de cette manière.

Algorithm 4 Algorithme Q-Learning et approximateur de fonctions

$L(a), a \in A$ liste de réseaux de neurones arbitrairement initialisés, pour des raisons de convergence, nous éviterons cependant de tout initialiser à zéro
Initialiser s
for $i = 0$ to N_{iter} **do**
  $r \leftarrow random([0; 1])$
  if $(i < N_{seuil})||(r \geq \epsilon)$ **then**
    Choisir a de manière aléatoire
  else
    $a \leftarrow \max_{a' \in A} Q(s, a')$
  end if
  Effectuer l'action a ; observer r, s'
  $L(a).append(s, (r + [\gamma * \arg\max_{L(a), a \in A} L.sortie(s')]))$
  $s \leftarrow s'$
end for

2.5 Une solution au problème d'affectation des crédits : le Q(λ)-Learning

2.5.1 Formalisme

Afin d'optimiser notre apprentissage, nous nous sommes intéressés au problème d'affectation des crédits [Singh et al., 1996] car la récompense obtenue à un instant t dépend en partie des actions prises aux instants précédents. Ainsi, une récompense r reçu à l'instant t peut servir à ré-estimer les actions précédentes. Dans notre cas, cela consiste à rajouter une trace d'éligibilité à l'algorithme de Q-Learning de manière à ce que chaque action effectuée dans l'environnement (qui amène à un résultat bon ou mauvais) soit mieux prise en compte dans l'apprentissage. Pour résumer, il s'agit de renforcer non seulement la valeur $Q(s, a_t)$ de l'action a_t dans un état s qui a conduite à une récompense à l'instant t, mais aussi les valeurs $Q(s, a_{t-i}), \ldots, Q(s, a_{t-1})$ des actions $a_{t-i}, \ldots, a_{t-1}$ qui ont conduite à ce résultat. Il nous faudra pour cela jouer avec un paramètre, λ, qui permet de pondérer l'importance des expériences précédentes.

Théoriquement, il existe plusieurs types de traces, on s'intéressera aux traces d'accumulation et aux traces de remplacement. La trace d'accumulation conserve sa valeur peu importe le nombre de fois que l'état s_t a été atteint. On peut la définir de la façon

FIGURE 2.9 – Traces d'accumulation et de remplacement

suivante :

$$e_{t+1}(s) = \begin{cases} \gamma\lambda e_t(s) & \text{si } s \neq s_t \\ \gamma\lambda e_t(s) + 1 & \text{si } s = s_t \end{cases}$$

La trace de remplacement est réinitialisée à chaque fois que l'état s_t est re-rencontré, on peut la définir de la façon suivante :

$$e_{t+1}(s) = \begin{cases} \gamma\lambda e_t(s) & \text{si } s \neq s_t \\ 1 & \text{si } s = s_t \end{cases}$$

La figure 2.9 montre la différence dans le temps entre ces deux types de traces.

Nous avons choisi d'utiliser une trace de remplacement. Pour mettre en place cette outil dans notre algorithme de Q-Learning, nous avons du faire quelques adaptations [Främling, 1997] car, dans notre cas, on a un approximateur de fonction par action possible. Dans le cas général (Q estimable de façon "tabulaire"), la valeur de la trace dépend, non seulement de l'état s_t, mais aussi de l'action a_t qui est appliquée à l'instant t :

$$e_t(s,a) = \begin{cases} 1 & \text{si } s_t = s \text{ et } a_t = a \\ 0 & \text{si } s_t = s \text{ et } a_t \neq a \\ \gamma\lambda e_{t-1}(s,a) & \text{si } s_t \neq s \end{cases}$$

En pratique, nous avons modifié les fonctions de mise à jour des poids dans les réseaux de neurones, de telle façon à ce que le calcul de la trace ne dépende plus que de l'action a choisie (comme on travaille avec un espace d'états continu, il y a très peu de chances de retomber plusieurs fois sur le même état). Pour stocker les traces, on a introduit un vecteur $\vec{M}$. La mise à jour du poids $w_{i,j}$ est maintenant faite de la façon suivante :

$$\vec{M}(i,j) \leftarrow \begin{cases} \vec{M}(i,j) * \lambda * w_{i,j} + \vec{\nabla}_j(\vec{x}) & \text{si } a = a_t, \\ \vec{M}(i,j) * \lambda & \text{sinon.} \end{cases}$$

Puis on calcule :

$$\vec{\nabla} w_{i,j}(\vec{x}) = (t_x - y_x) * \vec{M}(i,j)$$
$$\text{et } w_{i,j} \leftarrow w_{i,j} - \alpha_t * \vec{\nabla} w_{i,j}(\vec{x})$$

.

On remarquera qu'apprendre sans trace revient à apprendre en utilisant l'algorithme avec trace mais avec un facteur $\lambda = 0$.

2.5.2 Exemple : Le projet mountain car

Dans le problème du mountain car [Sutton and Barto, 1998], une voiture placée entre deux montagnes doit tenter d'atteindre le sommet de celle qui lui fait face (voir figure 2.10). Ce problème nous a intéressé car il est nécessaire de travailler sur un espace d'états continus (la position de la voiture) et car l'inter-dépendance entre les couples état/action est très forte (la voiture doit osciller entre les deux flans de montagne pour pouvoir atteindre le sommet).

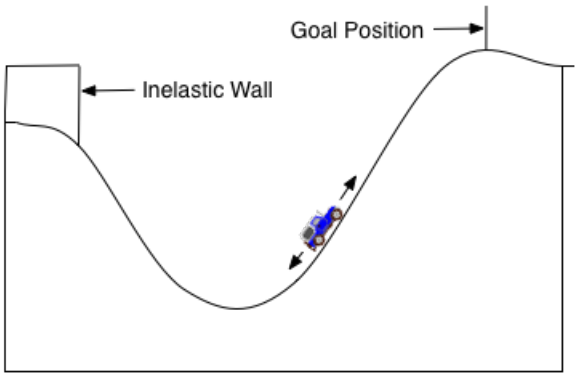

FIGURE 2.10 – Illustration du problème du mountain car

Le programme a été implémenté en C++. Vis-à-vis de la problématique du stage, ce problème est particulièrement intéressant car assez simple pour s'assurer du bon fonctionnement des différentes parties implantées qui sont inter-dépendantes (Q-Learning, Réseaux de neurones, apprentissage avec trace). L'environnement de travail possédant un espace d'états continu, on se rapproche de plus en plus du domaine dans lequel on aura à travailler qui est la robotique.

Le processus de décision Markovien associé à ce problème est le suivant :

- S : Velocity [-0.07 ;0.07], Position [-1.2 ;0.6] ;
- A : RECULER, RESTER, AVANCER ;
- R : $\begin{cases} 0 & \text{quand but atteint,} \\ -0.01 & \text{sinon.} \end{cases}$

Nous avons rencontrés lors de ce mini projet différents problèmes de paramétrage des algorithmes (les paramètres jouent un grand rôle dans la qualité de l'apprentissage) et avons été sensibilisé au rôle de chacun des paramètres ($\lambda, \gamma, \epsilon, \alpha_t$). Cette phase d'observation nous a été très utile pour mieux comprendre et envisager les problèmes dans l'apprentissage par le robot dans la suite du projet.

2.6 Bilan

Nous avons étudié un certain nombre d'outils nous permettant de mettre en place un apprentissage par renforcement développemental et en travaillant dans un domaine continu. L'algorithme de Q-Learning nous permet, une fois modifié et équipé d'approximateurs de fonctions, de mettre en place un tel apprentissage. L'utilisation de traces d'éligibilités nous permet de rendre cet apprentissage un peu plus efficace.

Cependant, suite au travail sur différents projets qui ont permis de tester nos outils, nous pouvons maintenant envisager un certain nombre de nouveaux problèmes. L'apprentissage en milieu continu nécessite beaucoup de pas d'apprentissage, il va nous falloir trouver une façon de le mettre en place sur un robot sur lequel il n'est possible d'effectuer des simulations que pendant un laps de temps limité (et qui prennent plus de temps). Le choix de l'environnement de travail est lui aussi primordial, un environnement trop simpliste ou trop compliqué peut dégrader les performances du système d'apprentissage. Nous avons aussi pu appréhender les enjeux du choix des paramètres dans nos différents algorithmes, il nous faudra être certain de s'assurer de la fiabilité de ceux-ci pour obtenir un apprentissage efficace (nous avons pu constater qu'un mauvais réglage des paramètres pouvait très fortement dégrader l'apprentissage). Enfin, nous avons pu nous rendre compte en testant les robots que la latence et les erreurs sont un problème majeur qu'il faudra considérer.

Chapitre 3

Apprentissage par renforcement développemental

3.1 Problématique

Au delà de la problématique liée à la robotique déjà abordée dans l'introduction (asservissement, erreurs des capteurs, latence, voir partie 1.2), une partie de la problématique du stage est liée à la robotique développementale. En effet il sera nécessaire, dans notre démarche, de trouver des solutions pour permettre au robot de continuer à apprendre malgré le changement dans sa perception de l'environnement qui l'entoure et le développement de ses capacités motrices.

Cette problématique est encore un domaine de réflexion ouvert dans le monde de la recherche. Il n'existe pas encore de solution "clé en main" pour résoudre ce type de problèmes. Certains chercheurs ont cependant abordé ce problème de manière conceptuelle [Lungarella et al., 2003], mais aucune solution concluante n'a été formulée pour le moment.

Une partie de la subtilité du problème réside dans le fait que pendant un certain temps, on fait apprendre une tâche bien spécifique au robot (avec un but à atteindre précis) puis, à un instant t, on décide de lui faire apprendre une nouvelle tâche (parfois contradictoire avec la première). Il nous faudra donc trouver des solutions pour, non seulement, initier le robot à la nouvelle tâche à apprendre, mais aussi lui permettre de conserver et d'exploiter les connaissances déjà acquises en trouvant un juste milieu pour appliquer les anciennes et nouvelles connaissances.

Une autre subtilité est liée à l'évolution des capacités du robot : de quelle façon peut-on inciter le robot à envisager de nouvelles actions/perceptions ? Il s'agit là du même type de problématique que pour l'apprentissage d'une nouvelle tâche : le robot

apprend avec un certain nombre d'actions/perceptions puis, à un instant t, on en introduit de nouvelles. Sachant que le robot est déjà familier avec un ensemble d'actions précis (qui lui permet certainement d'obtenir des récompenses assez efficacement), comment l'inciter à utiliser de nouvelles actions (qui ne lui permettent pas dans l'immédiat d'atteindre le but d'une façon aussi rapide) ? Sachant que le robot a appris à travailler dans un domaine d'états précis, comment conserver les connaissances acquises lorsque l'on fait évoluer ce même domaine ?

Une dernière question est à envisager. Nous disposons d'un système d'apprentissage permettant d'apprendre avec trace d'éligibilité (ce qui a tendance à avantager/désavantager plusieurs couple d'états/actions) et d'un système permettant d'apprendre sans trace d'éligibilité (ce qui a tendance à avantager/désavantager ponctuellement un couple d'états/actions). De quelle façon faut-il utiliser ces deux outils de manière à optimiser l'apprentissage du robot ?

Finalement rappelons qu'il est très important de considérer que pour toutes les problématiques d'évolutions qui vont être détaillées, il faut bien tenir compte du fait que lorsque l'on change l'environnement de travail, la politique à apprendre change elle aussi. Celà a pour effet de modifier la fonction de valeur Q que l'on cherche à approximer.

Dans la suite, nous commencerons par proposer une solution permettant de faire évoluer les capacités de perceptions du robot. Puis nous traiterons de l'évolution des capacités motrices de celui-ci. Nous proposerons ensuite une approche pour le changement de la tâche à apprendre au cours de l'apprentissage. Finalement, nous expliquerons la démarche adoptée concernant l'utilisation des différents types d'apprentissage par renforcement à notre disposition (avec ou sans trace d'éligibilité).

3.2 Evolution du domaine des perceptions

Tout d'abord rappelons que nous travaillons avec un algorithme de Q-Learning adapté pour un environnement à espace d'états continu, pour ce faire nous avons associé à chaque action a un réseau de neurones qui lui est propre et qui tente d'approximer $Q(s, a)$ pour les états s visités lors de l'apprentissage. Rappellons aussi que dans le cadre d'apprentissage mis en place, les perceptions du robot (càd les états du MDP) sont utilisés par l'approximateur de fonctions (un des réseaux de neurones) comme entrée pour décrire la situation courante du robot. Le gain espéré dans une situation (perception) s et pour une action a donnée ($Q(s, a)$) est alors approximé par celui-ci. Modifier les perceptions du robot impacte donc directement sur la structure interne de l'approximateur de fonction (le nombre/le type des entrées de celui-ci n'étant plus les mêmes).

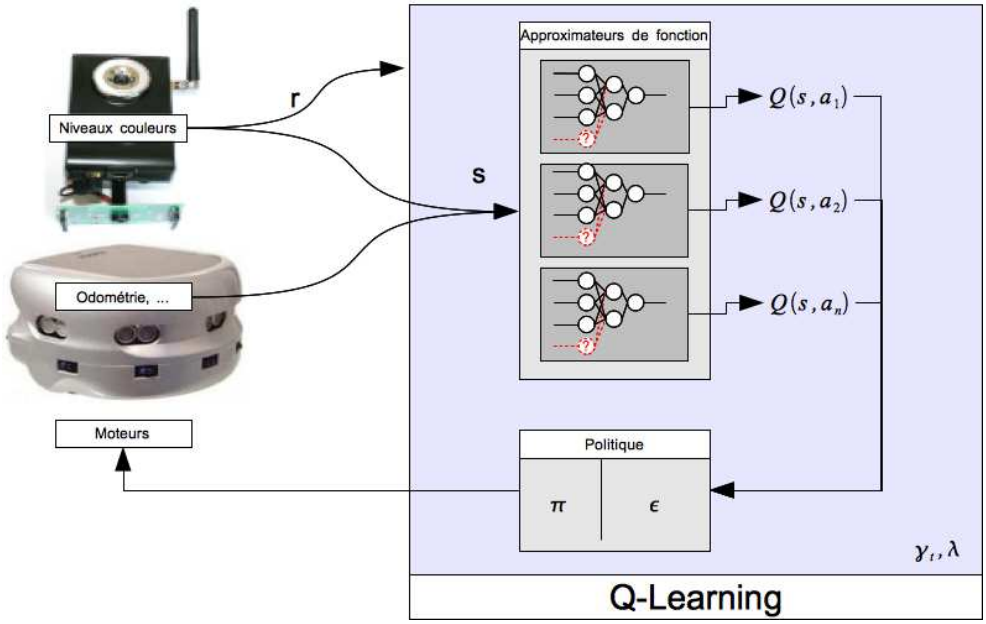

FIGURE 3.1 – Modifications (en rouge) liées à l'évolution du domaine des perceptions

Par exemple, dans le cadre expérimental du projet, nous avons travaillé avec des perceptions qui correspondent à la vision de bandes de couleurs par le robot. Nous avons ensuite augmenté le nombre de bandes (passant de 3 bandes à 5). La structure des réseaux de neurones associés à chaque action a donc dû changer (passage de réseaux à couche d'entrée de taille 3 à des réseaux à couche d'entrée de taille 5). Dans la pratique cela revient à rajouter un certain nombre de poids sur la couche d'entrée ($(5-3) * N_e$, où N_e est le nombre de neurones sur la première couche cachée).

La question est donc, de quelle façon doivent être initialisés les poids qui sont rajoutés entre la couche d'entrée et la première couche cachée ? Un grand nombre de possibilités existe, telle que :

- l'initialisation aléatoire des nouveaux poids ;
- la copie des poids suivant la sémantique du changement : si on part d'un domaine d'états correspondant à des bandes disposées comme il suit [gauche, centre, droite], et que l'on ajoute des bandes [gauche, gauche-centre, centre, droite-centre, droite], celà reviendrait à copier les poids associés, dans le réseau de neurones, à la bande "gauche" pour la nouvelle bande "gauche-centre" (idem pour droite) ;
- la moyenne entre les poids suivant la sémantique du changement : si l'on se place dans notre exemple, pour "milieu-gauche", faire la moyenne entre les poids de

"gauche" et "centre" (idem pour droite) ;

Nous avons aussi envisagé la possibilité de normaliser les poids après avoir effectué l'un de ces changements pour éviter aux réseaux de neurones de saturer.

La solution envisagée dans le stage est la copie des poids suivant la sémantique du changement. N'ayant pas eu l'occasion de tester d'autres méthodes, nous ne sommes cependant pas certains qu'il s'agisse de la solution la plus appropriée.

Nous nous sommes aussi posé des questions quant aux solutions abordables pour changer le domaine de perception sans pour autant avoir à changer la structure interne des approximateurs de fonctions (dans notre exemple, cela reviendrait à déplacer les bandes sans en rajouter). Par manque de temps, cette question est restée ouverte.

Rappelons aussi qu'une modification de l'espace d'état (des perceptions) dans lequel travaille le robot risque de perturber fortement l'apprentissage des fonctions de valeurs $Q(s, a)$ étant donné que des nouveaux domaines de perceptions sont ainsi "explorables" par le robot. Forcer le robot à re-prendre une stratégie d'exploration suite à ces changements semble donc nécessaire. Comme nous utilisons un algorithme de Q-Learning avec une exploration de type ϵ-gourmande, il suffirait de fixer un ϵ très faible (voir partie 2.3.2).

3.3 Evolution du domaine des actions

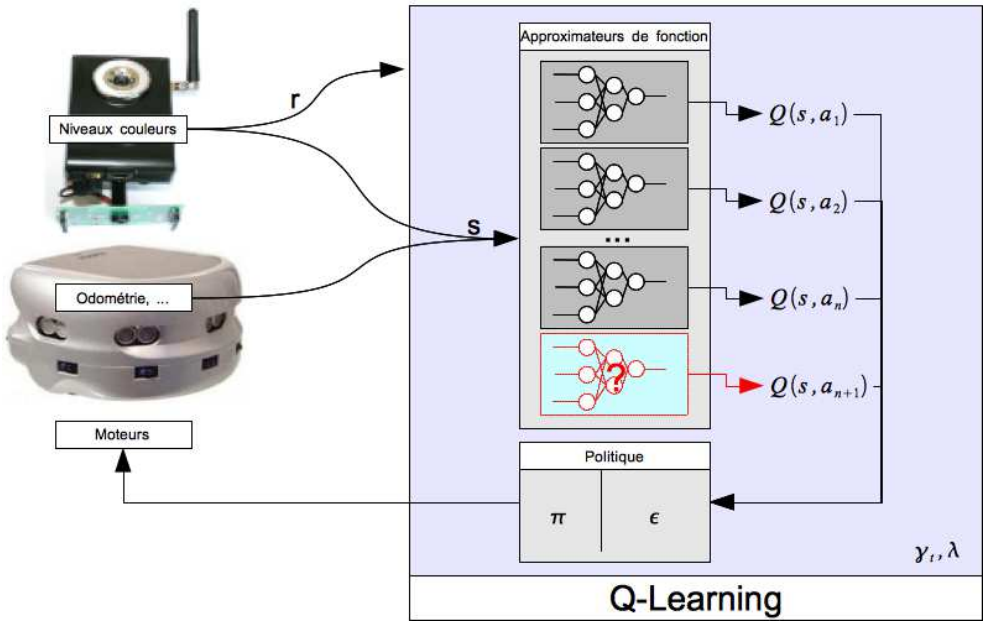

FIGURE 3.2 – Modifications (en rouge) liées à l'évolution du domaine des actions

Rappellons une nouvelle fois que nous travaillons avec un algorithme de Q-Learning adapté pour un environnement à espace d'états continu. Pour ce faire nous avons associé à chaque action a un réseau de neurones qui lui est propre et qui tente d'approximer $Q(s, a)$ pour les états s parcourus lors de l'apprentissage. Dans cette situation, ajouter une action revient à ajouter un nouvel approximateur de fonction. La question est : de quelle façon initialiser cet approximateur au moment de l'ajout ? Encore une fois, il existe plusieurs possibilités :

- l'initialisation aléatoire des nouveaux poids ;
- copier le réseau (et donc ses poids) de l'action la plus proche sémantiquement : par exemple, si on ajoute l'action "tourner à gauche lentement", celà reviendrais à dupliquer le réseau associé à l'action "tourner à gauche" et l'associer à cette nouvelle action ;
- la moyenne entre les poids suivant la sémantique du changement ;

Nous avons envisagé la possibilité de normaliser les poids après avoir effectué l'un de ces changements.

Nous avons choisi la copie des réseaux des actions les plus proches sémantiquement dans la phase d'expérimentation. La principale motivation de ce choix viens du fait que chaque action n'étant parcourue qu'un nombre limité de fois, l'apprentissage de la façon d'employer les nouvelles actions risquait d'être grandement ralenti en partant d'une nouvelle base.

Encore une fois, il faudra bien considérer que suite à l'ajout de nouvelles actions, il est nécessaire de forcer le robot à reprendre une stratégie exploratoire pour lui permettre d'apprendre à les utiliser. On accompagne ici le robot, en lui suggérant de réapprendre à se servir de ses fonction motrices, ça se rapproche de la notion de "scaffolding" expliquée dans [Lungarella et al., 2003].

Dans la phase expérimentale, nous avons aussi modifié l'environnement de travail (et donc l'espace d'états accessibles par le robot) de façon à ce que le robot soit confronté à des situations dans lesquelles il vaut mieux utiliser la/les nouvelle(s) actions. Par exemple, si le but à atteindre par le robot est l'alignement avec des formes colorées et les actions disponibles sont "tourner à gauche", "ne pas bouger", "tourner à droite" et que l'on souhaite ajouter les actions "tourner à gauche lentement" et "tourner à droite lentement", il peut être intéressant de réduire la largeur de certaines formes afin que les actions de rotation classiques ne permettent plus d'atteindre une récompense mais que les nouvelles actions, elles, le permettent.

3.4 Changements de buts

L'apprentissage par renforcement comme celui que nous avons mis en place est principalement guidé par un système de récompenses. La fonction de valeur $Q(s, a)$ est renforcée (ou non) pour chaque action a en fonction de la situation (l'état s) dans laquelle se trouve le robot dans l'environnement à un moment donné. Suivant la configuration que nous avons choisi, le robot est récompensé ponctuellement quand un certain seuil est atteint (par exemple, quand on dépasse une teneur en bleu sur l'image acquise par la caméra). Si l'apprentissage fonctionne de la manière escomptée, le robot apprend donc à se rapprocher le plus efficacement, partant d'une configuration précise, d'un état lui permettant d'obtenir une récompense. Notre problème est donc, une fois que le robot a appris à atteindre un but précis et que l'on modifie le but à atteindre, de trouver une manière d'apprendre à atteindre le nouveau but, tout en exploitant les connaissances déjà acquises. Par exemple, si le robot sait tourner sur lui même de façon à s'aligner avec un objet de couleur bleu, lorsque l'on souhaite lui apprendre à se rapprocher du bleu, il ne faut pas qu'il "oublie" de quelle façon il procédait pour s'aligner.

Nous avons réfléchi à plusieurs solutions, tout d'abord, une des questions à se poser est la suivante : soit un but B_1 que le robot à appris à atteindre et un nouveau but B_2, faut-il continuer à récompenser le robot quand il atteint B_1 alors que le nouveau but à atteindre est B_2 (si oui, la valeur de la récompense doit-elle changer) ? Plus précisément, nous nous sommes intéresses au cas particulier ou les tâches qui sont liés aux buts B_1 et B_2 sont "complémentaires" (par exemple, apprendre à tourner puis apprendre à avancer et à tourner).

Une des solutions que nous avons envisagé est de "placer" le nouveau but à atteindre dans un domaine d'état qui n'a pas encore été exploré par le robot.

Par exemple, si on considère un système ou l'espace d'états dans lequel évolue le robot est décrit par l'angle $\theta \in [\frac{-\pi}{2}, \frac{\pi}{2}]$ duquel il s'est déplacé depuis sa position initiale. Le robot ne peut que tourner sur lui même. Au début du scénario, le robot utilise une caméra pour détecter des figures colorées, pour cela il se fie à la quantité de couleur β_1 présente dans l'image. On souhaite changer le but et éloigner les figures colorées. Par rapport au précédent but, le robot sera récompensé quand il détecte une quantité de couleurs β_2 moins importante que β_1. L'idée serait ici de permettre au robot d'explorer une partie du domaine d'état jusqu'ici inaccessible et on placerait le nouveau but dans une des parties encore inexplorées. Dans cet exemple, on pourrait lui permettre de tourner sur $\theta' \in [-\pi, \pi]$, et on placerait les objets colorés plus éloignés dans $\theta'_1 = [-\pi, \frac{-\pi}{2}]$ ou $\theta'_2 = [\frac{\pi}{2}, \pi]$.

3.5 Choix du type d'apprentissage

Lors d'un apprentissage avec trace d'éligibilité, on modifie non seulement la fonction de valeur $Q(s, a)$ du couple état/action courant mais aussi les fonctions des couples explorés plus ou moins récemment (en fonction de λ). Lorsque dans l'état courant, on rencontre une situation donnant lieu à une récompense, un certain nombre d'états précédents sont aussi renforcés. Cependant, lorsque l'on rencontre une situation ne donnant pas lieu à une récompense, les états précédents sont eux renforcés négativement. Rappelons aussi que l'on travaille dans un domaine ou les récompenses sont peu fréquentes, l'apprentissage avec trace d'éligibilité peut donc constituer un frein à l'apprentissage en lui-même.

L'apprentissage sans trace d'éligibilité affecte uniquement l'état courant. Cependant, comme on se situe dans un environnement à domaine d'état continu, chaque état n'est exploré qu'un très petit nombre de fois. Les récompenses dans le domaine étudié étant peu fréquentes, il se peut donc, en utilisant cette stratégie, que l'apprentissage soit très lent et/ou parte dans une mauvaise direction (les couples états/actions permettant d'atteindre un état but étant souvent renforcés négativement).

Aussi, nous pourrons remarquer que, les actions étant effectuées suivant une politique aléatoire, le nombre de fois que chaque action est utilisée (et donc que la Q-valeur du couple action/valeur soit renforcée) ne peut théoriquement être nul (ou presque nul).

Suite à nos expérimentations, nous avons constaté que d'employer une stratégie utilisant une trace uniquement avait tendance à faire stagner le système à partir d'un certain nombre de pas d'apprentissage, la phase d'apprentissage en partant du début étant pourtant satisfaisante. Une raison pouvant expliquer ce phénomène est qu'à partir d'un certain seuil, le grand nombre de récompenses négatives obtenues par le robot et affectant un grand nombre de couples état/action fait saturer les réseaux de neurones associés aux actions et bloquent ainsi l'apprentissage.

La solution que nous avons adoptés est donc de commencer par une phase d'apprentissage avec trace (pour renforcer un grand nombre de couples état/action) puis de passer à une politique d'apprentissage sans trace (pour renforcer les couples dans les domaines donnant lieu à une récompense) avec une répartition de la moitié de la durée d'apprentissage pour chacune des phases.

Chapitre 4

Expérimentations

4.1 Introduction

4.1.1 Objectifs

Rappelons tout d'abord que le but de nos expérimentations est d'illustrer le fait que la mise en place d'un apprentissage de type développemental dans un cadre robotique peut être efficace lorsque la tâche à apprendre est complexe. Il nous faut pour cela réfléchir à un scénario permettant de mettre un tel apprentissage en place considérant les problématiques déjà évoquées dans les parties 1.2 et 3.1. Ce scénario devra nous permettre de mettre en place les mécanismes évoqués dans la partie 3 et d'en évaluer les performances.

Pour mettre un tel scénario en place, nous disposons dans le laboratoire d'un certain nombre d'outils :

- des robots Khepera3 munis de différents types de capteurs (infra-rouge, ultrason, odometrie) et étant piloté par le biais d'un réseau sans-fil ;
- de caméra KorwlCam qu'il est possible d'installer sur les robots ;
- de bibliothèques de programmations développées par l'équipe MAIA pour interagir avec le robot ;
- de différents environnements de développement pour créer des scripts utilisables sur le robot ;

Dans tous les cas, nous serons donc amenés à travailler avec le robot Khepera 3.

Au delà des problèmes liés à la robotique et à la robotique développementale déjà évoquée, nous aurons aussi, éventuellement, à traiter des problèmes lié au traitement de l'image en temps réel. Nous remarquerons aussi que peu importe le type de scénario choisi, comme nous travaillerons avec un robot piloté par le biais d'un réseau sans-fil, nous serons confrontés à des problèmes de latence. Nous devrons donc choisir un

scénario limitant, un maximum, les confrontations à ces différents problèmes.

4.1.2 Scénario retenu

Comme nous venons de l'évoquer le robot Khepera 3 dispose de plusieurs types de capteurs (infra-rouge, ultrason, odométrie) et nous disposions d'une caméra Korwl-Cam. Dans notre scénario, nous avons choisi de ne pas utiliser les capteurs du robot car, considérant le type d'apprentissage que l'on veut mettre en place, ils sont soit trop sensibles au cadre dans lequel s'effectue les tests (infra-rouge), soit pas assez précis (ultrason, odométrie). Nous avons donc choisi de travailler avec la caméra en utilisant la librairie RetineURBI développée par notre tuteur Alain Dutech (voir partie 4.2.4).

Le scénario d'apprentissage dévelopemental retenu est un scénario dans lequel le robot travaille avec des formes colorées (une seule teinte). Il peut appréhender ces formes grâce à la caméra et à la librairie RetineURBI (concentration d'une teinte sur une ou plusieurs bandes verticales) et dispose d'un nombre d'actions limitées (avancer, reculer, rotation gauche, rotation droite).

Le but final est de permettre au robot de se diriger dans un labyrinthe en suivant des flèches (triangles isocèles) colorées. Pour cela, le robot va d'abord apprendre à s'aligner avec un objet coloré, puis apprendre à s'en rapprocher, ensuite il apprendra à appréhender des formes triangulaires colorées (flèches) et enfin apprendre à suivre les flèches. Certaines contraintes liés à l'environnement de travail nous ont fait opter pour la détection de formes de teintes bleutées (voir partie 4.2.4).

4.2 Environnement de travail

Cette partie présente les détails techniques des outils que nous avons utilisés qui ont influencés notre choix de scénario. Elle n'est cependant pas nécessaire à la compréhension des mécanismes d'expérimentation et des résultats présentés dans la partie 4.3.

4.2.1 Khepera3

Le robot Khepera 3 est développé par l'entreprise K-Team, il embarque d'un système d'exploitation de type Unix nommé KoreBot aussi développé par K-Team. Le système KoreBot assure une compatibilité avec la plupart des librairies développées pour GNU/Linux. Une boite à outils est aussi fournie avec le robot pour permettre le développement d'application utilisant ses différentes fonctionnalités motrices et sensorielles.

FIGURE 4.1 – Le robot khepera3

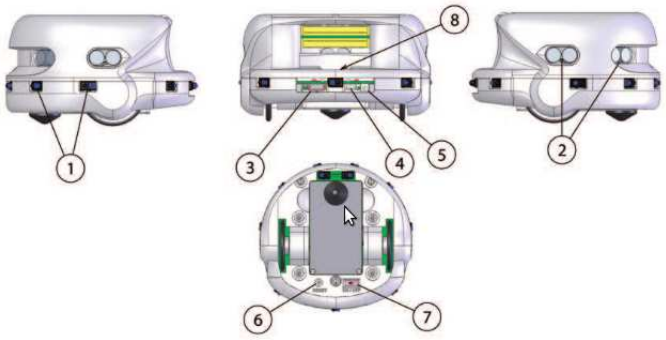

FIGURE 4.2 – Plan du robot

Le robot peut se déplacer à l'aide de ses deux roues munies d'une semelle en caoutchouc qui permet de garantir l'adhérence sur différent types de surfaces. Il est possible de commander chaque roues indépendamment en précisant une vitesse angulaire à atteindre ou bien un nombre de tour de roues à effectuer (pas très précis).

Le robot perçoit l'environnement qui l'entoure à l'aide de plusieurs types de capteurs. Il dispose de 9 capteurs à infrarouge pour la détection des obstacles, 5 capteurs à ultrasons pour la détection à longue portée d'objets et 2 capteurs à infrarouge placés sous la coque pour la détection de surfaces au sol. L'équipe MAIA a aussi développé un module d'odométrie qui permet de connaitre, plus ou moins précisément, la position atteinte par le robot (x,y, angle) suite à des déplacements.

Il est possible de communiquer avec le robot à travers le réseau sans-fil (Wi-Fi ou Bluetooth), celui-ci dispose aussi d'une batterie lui permettant de se déplacer de façon indépendante (durant nos tests nous avons constaté autonomie d'une heure environs). Enfin, il dispose d'un Bus permettant d'accueillir les cartes Compact Flash, d'un port USB et d'un Bus KB-250 permettant de connecter différents types de modules d'ex-

tension.

Les caractéristiques techniques (résumées) du robot sont les suivantes :

- Processeur : DsPIC 30F5011 à 60MHz ;
- RAM : 4 Ko sur DsPIC, 64 Mo de KoreBot Extension ;
- Flash : 66 Ko sur DsPIC, 32MB KoreBot Extension ;
- Déplacement : 2 servomoteurs avec codeurs incrémentaux (environ 22 impulsions par mm de mouvement), vitesse maximum 0.5 m/s ;
- 9 capteurs infra-rouges pour capter la proximité jusqu'à 25cm ;
- 2 capteurs infra-rouges détecteurs de proximité du sol ;
- 5 capteurs à ultrasons avec une portée de 20 cm à 4 mètres ;
- Entrees/sorties : 2 LED programmables ;
- Diamètre : 130 mm ; Hauteur : 70 mm ; Poids : 690 g ;

4.2.2 KorwlCam

La caméra KorwlCam est développée par l'entreprise K-Team. Cette caméra permet de capturer des images et des videos dans différentes résolutions et différents formats d'image. Nous utilisons des images dans la résolution 320x200 pixels au format Bitmap.

Il existe deux types de connectiques pour communiquer avec la caméra : une liaison filaire et une liaison sans-fil (Wi-Fi). Pour des raisons pratiques, nous utilisons la liaison sans-fil (ce qui évite de gêner le robot dans ses déplacement), cependant ce type de liaison implique un plus grand niveau de latence et est soumise aux problèmes liés au Wi-Fi (déconnexions, qualité du signal, ...).

Dans sa version de base, la caméra ne propose qu'une alimentation de type filaire, l'équipe MAIA a fait développer une carte permettant de fixer la caméra sur le robot Khepera 3 (voir figure 4.3) et d'utiliser la batterie de celui-ci pour l'alimenter. Le désavantage de ce type de configuration est la consommation plus rapide des ressources électriques du robot.

4.2.3 URBI

URBI est un environnement de développement développé par la société Gostai. Cet environnement comprend un langage de script (l'URBI script), c'est-à-dire un langage qui n'est pas compilé, mais interprété en temps réel. C'est un langage évènementiel avec une syntaxe proche de la syntaxe C++. Son fonctionnement est basé sur une architecture client/serveur. Le "noyau" URBI fonctionne sur le serveur, le robot, et le code à exécuter est envoyé au serveur par un client, l'ordinateur (voir figure 4.4). Ce langage est principalement destiné à la modélisation de comportements et est utilisé

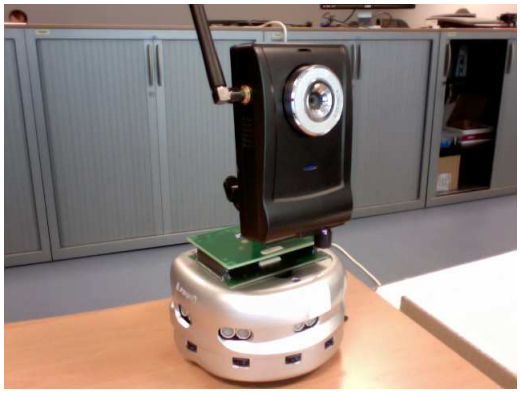

FIGURE 4.3 – Robot Khepera3 équipé de la caméra KorwlCam

dans des domaines tels que la robotique ou les jeux vidéos.

Rappelons que les langages de script ont pour principal désavantage la lenteur de l'exécution du code, la limitation dans les possibilités de programmation et le faible nombre de librairies externes à disposition. Nous avons choisi d'utiliser l'environnement de développement URBI pour plusieurs raisons :

- les facilités de programmations en URBI script ;
- la possibilité de créer et d'utiliser des modules C++ au sein de l'environnement URBI (ce qui permet de pallier aux limitations du langage de script) ;
- l'équipe MAIA a déjà développé des drivers de haut niveaux permettant de piloter le robot Khepera3 et d'utiliser la caméra KorwlCam en passant par URBI ;

Dans le cadre de l'application que nous avons donc développé, le module d'apprentissage (Q(λ)-Learning, modélisation de l'environnement de travail) est écrit en URBI script, le reste (librairie de réseaux de neurones) est écrit en C++ et employé au sein d'URBI par le biais de modules (UObjects). Nous avons choisi de procéder ainsi pour la raison que les calculs liés à la mise à jour des réseaux de neurones sont assez complexes (et donc plus long a exécuter et à mettre en place dans un langage de script).

Suite à ce choix, un seul problème persiste encore, comme exposé dans la figure 4.4, la plus grosse partie de l'environnement (le serveur) est installée sur le robot. Le robot Khepera3 ne disposant pas d'une mémoire suffisante et d'une Unité Arithmétique et Logique (donc pas possibilité d'effectuer les calculs en nombre flottants nécessaires

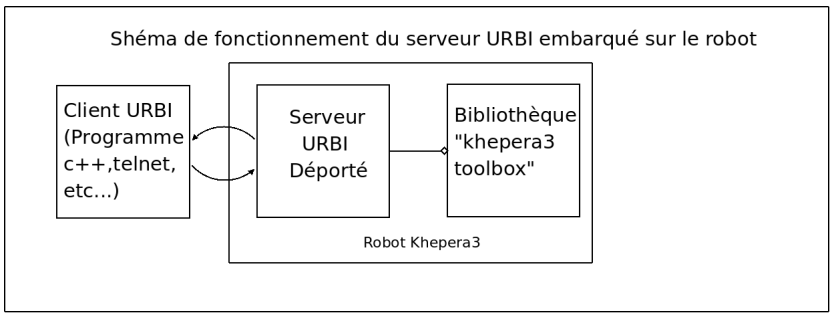

FIGURE 4.4 – Fonctionnement classique d'URBI

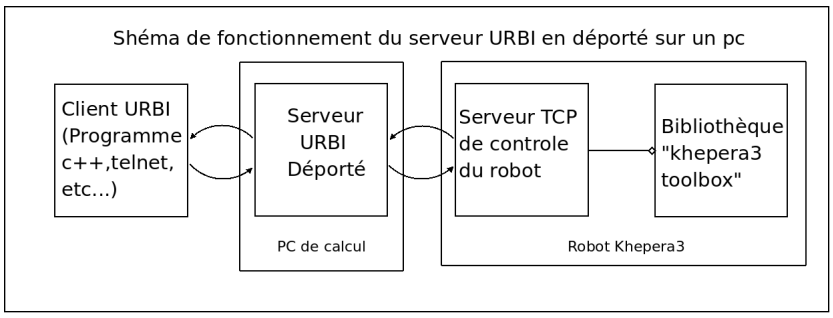

FIGURE 4.5 – Fonctionnement d'URBI déporté

dans les algorithmes utilisés), nous avons dû trouver une solution pour déporter le serveur sur un ordinateur. L'équipe MAIA avait déjà travaillé sur cette question et a développé une architecture permettant de ce faire. Dans cette architecture (figure 4.5), tous les calculs sont faits sur ordinateur et une connexion est mise en place avec le robot pour permettre de contrôler ses mouvements et récupérer les données fournies par ses capteurs.

4.2.4 RetineURBI

La librairie RetineURBI a été développée par notre tuteur Alain Dutech, elle utilise les technologies OpenCV, libopenstream, GTK+ et des bibliothèques développées par des équipes du LORIA. Nous ne rentrerons pas dans les détails d'implémentation cependant nous allons rappeler certains des principes de fonctionnement. Ce module à pour but de permettre d'obtenir la concentration en une couleur (sous la forme d'un pourcentage) sur un certain nombre de bandes verticales de l'image (colonnes de 1 pixel en un point de la largeur précis) tout en éliminant les informations trop bruitées (bruit liés à la luminosité et aux reflets dans l'environnement).

Comme précisé plus tôt, nous travaillons sur des images renvoyées par la caméra KorwlCam au format Bitmap, malheureusement l'espace colorimétrique utilisé dans ce format pour représenter les couleurs (l'espace RGB) n'est pas très adapté aux traitements que nous voulons effectuer sur l'image, une partie du module se charge donc de transposer les couleurs de l'image dans un espace colorimétrique plus adapté, l'espace TSV (Teinte Saturation Valeur, HSV en anglais), voir [Job].

Contrairement à la représentation classique des couleurs dans une image (en niveaux de Rouge, Vert et Bleu (RGB)), l'espace colorimétrique TSV représente les couleurs par trois composantes :

- la teinte (Hue) suivant l'angle correspondant sur le cercle des couleurs ;
- la saturation, ou intensité, de la couleur qui varie entre 0 et 100% ;
- la valeur (Value), "brillance", ou encore clareté de la couleur qui varie entre 0 et 100% ;

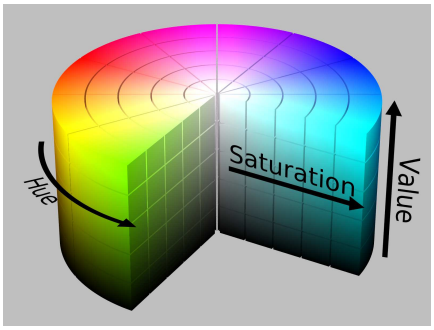

FIGURE 4.6 – L'espace colorimétrique TSV

Travailler dans cet espace colorimétrique est très intéressant vis à vis du but du module, car on peut très facilement détecter quels sont les pixels de l'image ou la couleur est trop claire, trop sombre ou pas assez saturée et ne pas les considérer dans le traitement effectué sur l'image (calcul de pourcentages sur les bandes verticales), ce qui est relativement complexe à effectuer dans un espace (RGB).

Considérant cet espace de travail, il suffit donc ensuite de parcourir tous les pixels d'une colonne, de ne sélectionner que ceux dont la teinte est satisfaisante (bleu) et dont la saturation et la valeur sont conformes à nos contraintes, puis d'en calculer le pourcentage par rapport au nombre de pixels dans la colonne afin d'obtenir le résultat escompté.

Dans le cadre du projet, nous avons fait un certain nombre de tests pour bien choisir la couleur à détecter dans notre environnement de travail (bureau éclairé par des néons), nous avons finalement choisi de détecter des couleurs bleutées dont la teinte varie entre 200° et 280°, la saturation entre 0% et 100% et la valeur entre 20% et 80%.

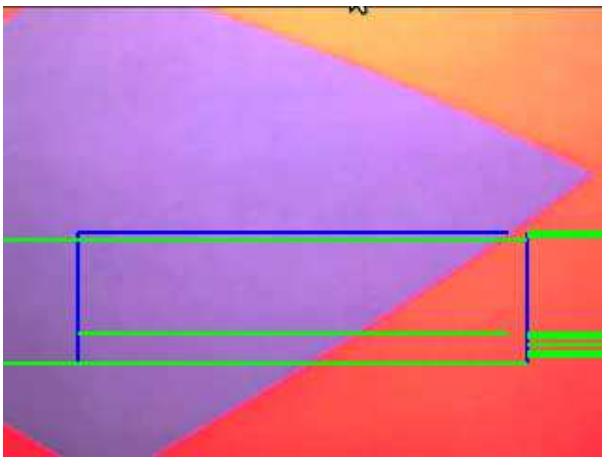

FIGURE 4.7 – Capture d'une image avec 2 bandes

Dans la figure 4.7, nous avons souhaité obtenir la concentration en bleu sur deux bandes verticales placées à 40 et 280 pixels de largeur, les valeurs retournées par le module sont [100%,16%].

4.2.5 Configuration de l'environnement de travail

Nous avons travaillé dans une configuration bien particulière, ce qui a augmenté la latence déjà induite par la communication sans-fil avec le robot. Pour utiliser URBI en mode déporté, nous avons du travailler sur une machine, cependant, elle ne possedait pas de carte Wi-Fi. Nous avons donc utilisé notre machine personnelle pour assurer le relais entre la machine du LORIA et le robot (via un tunnel SSH). Le robot lui se connectait à une borne Wi-Fi disposé dans la salle ou l'on a travaillé. La figure 4.8 résume les liens entre les différents composants de l'installation.

Pour réduire le problème de latence lors de la communication avec le robot nous avons décidé de temporiser les actions qu'il effectue. Ce choix implique un ralentissement des simulations, sachant que le robot risque d'avoir besoin de procéder à un grand nombre d'expériences pour apprendre, nous avons choisi de travailler avec des jeu d'exemples. Ainsi, plutôt que d'apprendre seulement à travers de simulations réelles, le robot pourra apprendre par le biais de simulations virtuelles dans lesquelles il répètera un certain de fois des expériences déjà effectuées et enregistrées dans le jeu d'exemples.

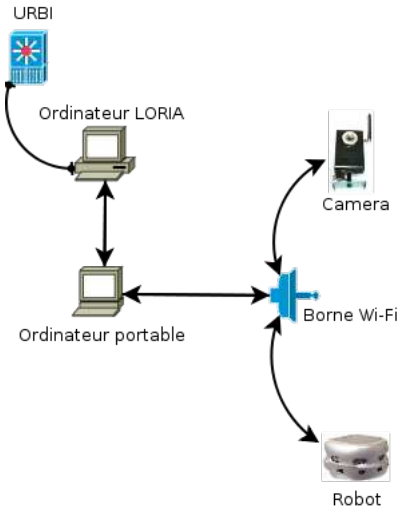

FIGURE 4.8 – Configuration de travail (réseau)

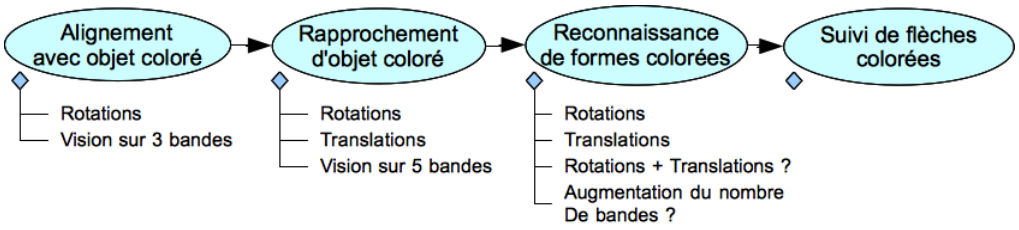

FIGURE 4.9 – Différentes phases du scénario

4.3 Découpage de l'apprentissage en phases

Rappelons que, le but dans le scénario qui a été choisi est d'apprendre au robot de se diriger dans un labyrinthe en suivant des flèches (triangles isocèles) colorées. Conformément à notre l'approche théorique que nous essayons d'illustrer, nous avons choisi de découper l'apprentissage de cette tâche complexe en plusieurs sous-tâches (plus simples). Le robot va donc commencer par apprendre à s'aligner avec un objet de couleur [phase 1], puis apprendre à s'en rapprocher [phase 2], apprendre à détecter des formes triangulaires (les flèches) [phase 3] et enfin apprendre à se diriger à partir des formes triangulaires [phase 4].

Par manque de temps, nous n'avons eu le temps de travailler et d'effectuer des tests que sur la phase 1, nous avons cependant détaillerons la démarche à suivre pour l'expérimentation dans les phases 2 et 3. Comme nous n'avons pas réfléchi aux détails d'expérimentation de la phase 4, celle-ci ne sera pas présentée (voir figure 4.9).

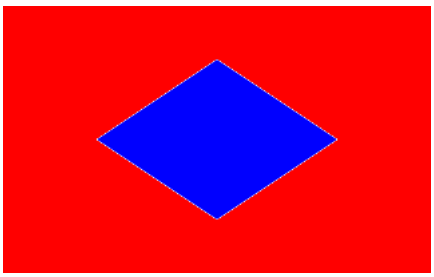

FIGURE 4.10 – Petit Losange

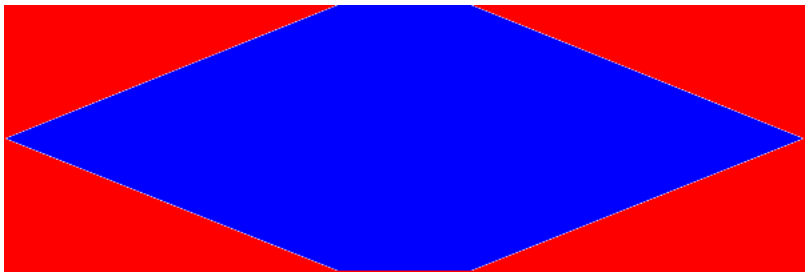

FIGURE 4.11 – Grand Losange

4.3.1 Phase 1 : Alignement avec une couleur

Démarche

Après un grand nombre d'expérimentations pour trouver le bon type de forme à utiliser pour l'apprentissage, nous avons choisi dans cette phase d'utiliser des losanges plus ou moins larges. La forme du losange nous permet d'explorer une partie du domaine d'états bien spécifique : les bords du losange contiennent peu de bleu tandis que son centre en contient beaucoup. La tâche à apprendre par le robot dans cette phase est l'alignement avec le bleu. Pour s'aligner, si le robot détecte une certaine quantité de la couleur sur le côté, la meilleure façon d'obtenir une récompense peut être d'effectuer une rotation dans ce sens. La forme de losange est donc très pratique car, une fois une partie du losange détectée sur le côté, plus le robot se rapproche du centre, plus la concentration en bleu est importante. Nous avons pensé que ce type de parcours du domaine d'états était très adapté pour apprendre au robot à s'aligner avec la couleur choisie (le robot tourne sur lui même, la concentration en bleu augmente jusqu'à ce qu'il soit récompensé).

Pour permettre un apprentissage rapide et efficace, nous avons enfermés le robot dans une boite dans laquelle nous avons placés les losanges sur les faces intérieures (voir figure 4.3.1). Rappelons tout d'abord que le robot perçoit seulement les concentra-

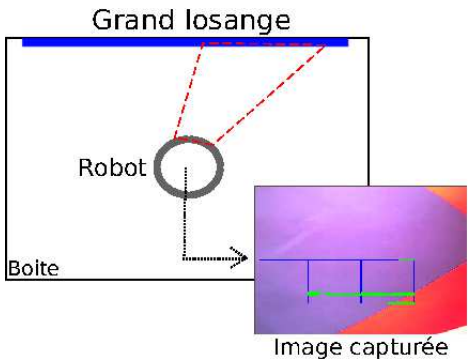

FIGURE 4.12 – Capture d'une image du grand losange

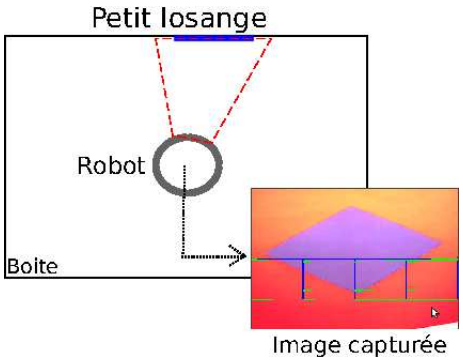

FIGURE 4.13 – Capture d'une image du petit losange

tions de couleurs sur des bandes verticales. A chaque bande verticale est donc associé un chiffre compris entre 0 et 1 représentant le pourcentage de bleu contenu dans celle-ci. Pour commencer, nous avons travaillé avec trois bandes verticales (respectivement placées à 80 (B_0), 160 (B_1) et 240 pixels (B_2)).

Nous avons fait face à des difficultés techniques pour que le robot puisse détecter convenablement la couleur bleu. En effet, l'environnement de travail est fort assujetti au changement de luminosité, la couleur avait tendance à se refléter et le robot détectait du bleu là ou il n'y en avait pas. Nous avons donc décidé de remplir les bords du losange avec une autre couleur que le blanc (le rouge) pour limiter les effets de la réflexion.

Le robot commence par travailler avec trois actions :

- a_0 : tourner à gauche (vitesse de rotation des roues : 2000 tics/s) ;
- a_1 : tourner à droite (vitesse de rotation des roues : 2000 tics/s) ;
- a_2 : ne pas bouger ;

On en ajoute ensuite deux :

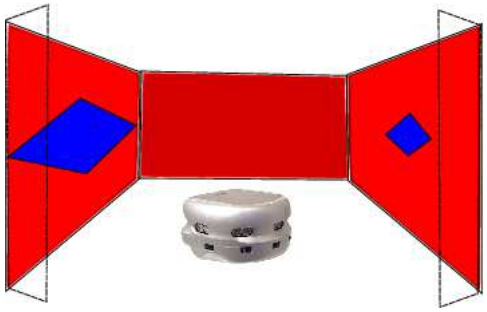

FIGURE 4.14 – Environnement dans lequel évolue le robot

– a_3 : tourner à gauche lentement (vitesse de rotation des roues : 1000 tics/s) ;
– a_4 : tourner à droite lentement (vitesse de rotation des roues : 1000 tics/s) ;

Le robot évolue dans un environnement où sont disposés deux losanges, en faisant varier la longueur et la largeur des losanges, nous avons pu permettre au robot d'apprendre les avantages des différents types d'actions. Un losange large et long pour lui apprendre à utiliser les actions de base (a_0, a_1, a_2). Un losange peu large et long pour lui permettre d'apprendre à utiliser les actions lentes (a_3, a_4), le losange devait être suffisamment petit pour qu'une action de rotation classique ne lui permette d'atteindre un but que très difficilement.

La fonction de transition est elle aussi sujet au changement car l'environnement de travail (l'espace d'états) et l'ensemble des actions sont modifiés.

Le robot est récompensé suivant la fonction de récompense suivante :

$$r = \begin{cases} 1 & \text{si } [(B_0 * 0.7 + B_1 + B_3 * 0.7) \geq 1.6) \wedge (B_0 * 0.95 \leq B_1) \wedge (B_2 * 0.95 \leq B_1)], \\ 0 & \text{sinon} \end{cases}$$

Concernant les outils utilisés, nous avons travaillé avec les paramètres suivants :
– Q-Learning : $\gamma = 0.9, \epsilon_{init} = 0.2$;
– Réseaux de neurones : 6 neurones en couche caché ;

Résultats des tests

Rappelons que l'idée est ici de comparer différentes méthodes d'apprentissage lors d'une évolution dans les perceptions ou les capacités motrices du robot. Comme précisé dans la partie précédente, dans cette expérience, nous avons ajouté un certain nombre d'actions possible à l'ensemble d'actions du robot au cours de l'apprentissage. Il nous a fallu trouver un moyen de comparer les différentes méthodes d'apprentissage, nous avons, pour cela, développé différents outils de visualisation et avons réfléchi à des indicateurs de performances possibles.

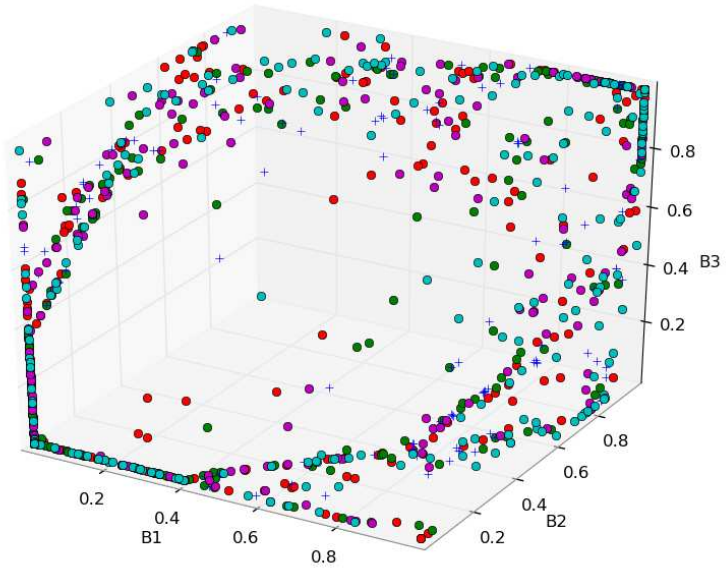

FIGURE 4.15 – Trajectoire suivie par le robot dans un des jeux d'exemples

Dans les figures décrivant des trajectoires, chaque point correspond à un état précis (B_1,B_2,B_3), la couleur du point correspond à l'action :

- *Rouge : tourner à gauche ;*
- *Vert : tourner à droite ;*
- *Bleu : ne pas bouger ;*
- *Magenta : tourner lentement à gauche ;*
- *Cyan : tourner lentement à droite ;*

Tout d'abord, il nous a fallu nous assurer que le robot menait une exploration conforme à nos attentes (pas d'incohérences dues à des problème lors de l'acquisition des images, ...). Comme nous pouvons le voir sur la figure 4.15 (qui est issue d'un des jeux d'exemples), le domaine d'états semble parcouru de manière convenable : il n'y a pas d'incohérences du type $B_1 < B_0 \wedge B_1 < B_2$ et toutes les actions semblent utilisées de manière équitable.

La deuxième chose à faire a été de choisir les bon paramètres pour les simulations. Nous avons pour cela estimé la valeur des politiques obtenues en faisant varier différents paramètres en comparant la somme pondérée des valeurs d'apprentissage à chaque pas de celui-ci :

$$g = \sum_{t=0}^{N_pas} \left(r_t + \gamma_t * \max_{a \in A} Q(s_t, a) \right)$$

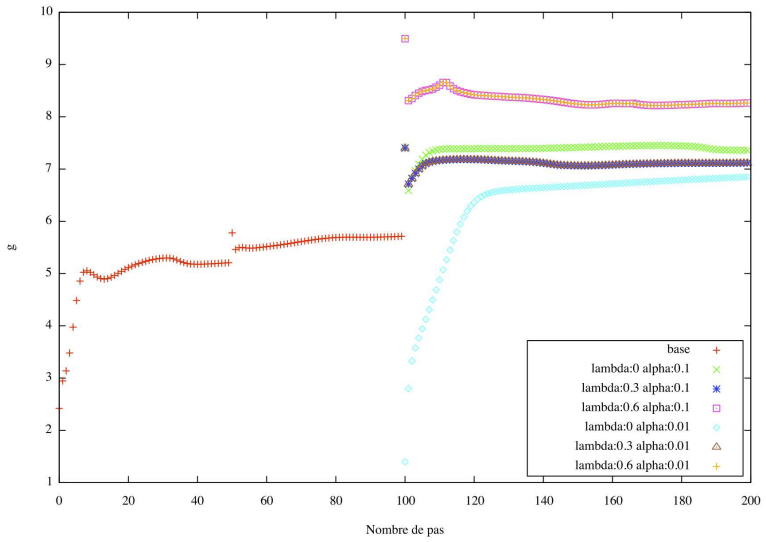

FIGURE 4.16 – Valeurs de g au cour de l'apprentissage (avec base)

Nous avons comparé les résultats obtenu en faisant varier les paramètres α et λ (voir figures 4.16 et 4.17. Ce critère nous a permis de choisir de travailler avec $\lambda = 0.6$ et $\alpha = 0.01$.

On rappelle que, les deux méthodes permettant d'apprendre suite à l'ajout d'actions que nous souhaitons comparer sont :

- (1) reprendre un apprentissage depuis le départ : travail *sans base* ;
- (2) utiliser la base de connaissances déjà acquises et continuer à apprendre en adaptant notre structure d'apprentissage (ajout de réseaux et copie des poids) : travail *avec base* ;

Dans la simulation mise en place, le robot apprend avec un jeu d'exemples, les exemples ont été passés au robot pour un total de 200 000 pas d'apprentissage environs. Dans le cas (1), où on travaille directement avec les 5 actions, on a commencé par apprendre avec $\lambda = 0.6$ pendant 100 000 pas, puis on a appris avec $\lambda = 0$ pendant 100 000 pas. Dans le cas (2), on commence par travailler avec 3 actions : on apprend durant 50 000 pas avec $\lambda = 0.6$, puis pendant 50 000 pas avec $\lambda = 0$. On ajoute ensuite les deux actions supplémentaires, puis on apprend pendant 25 000 pas avec $\lambda = 0.6$ et on continue pendant 75 000 pas avec $\lambda = 0$ (voir figure 4.18).

Nous avons commencés par essayer d'estimer la valeur de la politique finale obtenue pour (1) et (2) à l'aide d'un outil que nous avons développé. Cet outil permet d'obtenir, pour un jeu d'exemple donné, la meilleure action à faire (suivi de la politique gloutonne sur les valeurs actuelles de Q) en chaque état parcouru. Comme nous

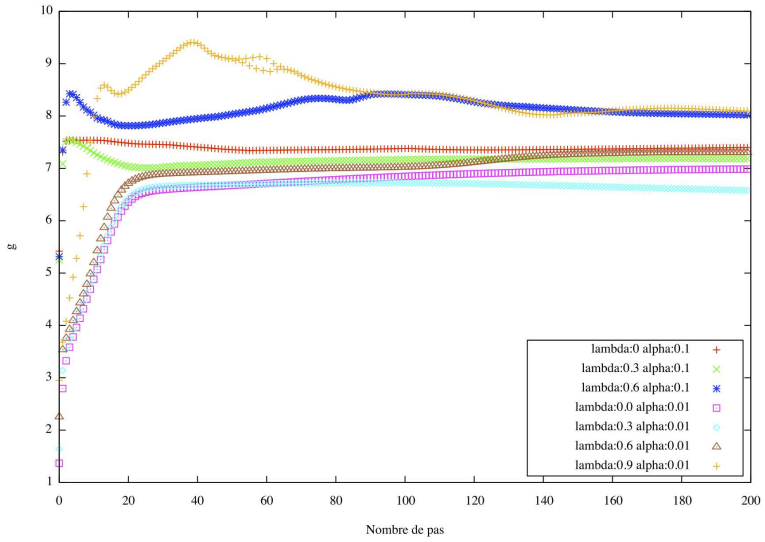

FIGURE 4.17 – Valeurs de g au cour de l'apprentissage (sans base)

Type d'apprentissage/ Pas	50 000	50 000	25 000	75 000
Avec Base	$\lambda = 0.6$	$\lambda = 0$	$\lambda = 0.6$	$\lambda = 0$
Sans base	$\lambda = 0.6$		$\lambda = 0$	

FIGURE 4.18 – Déroulement des simulations

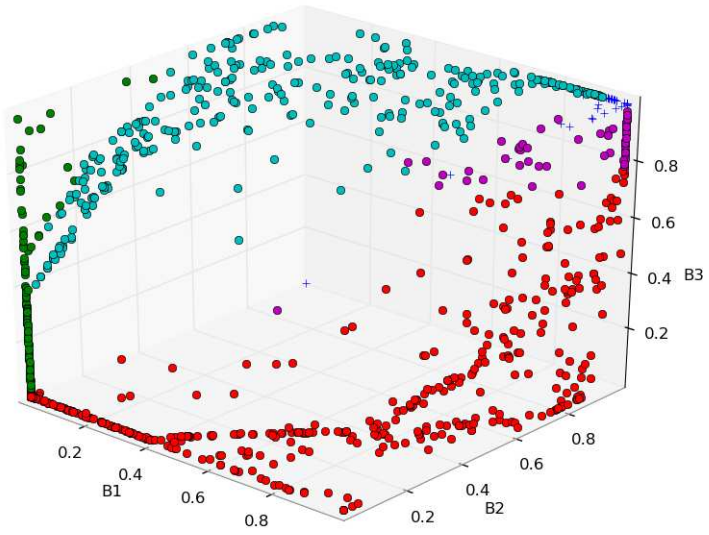

FIGURE 4.19 – Evaluation de la politique finale sur le jeu d'exemples (avec base)

pouvons le remarquer sur les figures 4.19 et 4.20 la politique finale pour (1) et (2) semblent très proches. On remarquera que cette politique semble efficace, lorsque le robot détecte du bleu sur la gauche, il tourne à gauche (resp. droite) et quand il est aligné sur le centre du losange ($B_0, B_1, B_2 \approx 1$) il ne bouge plus.

Comme ce précédent critère de comparaison ne nous a pas permis de comparer nos deux méthodes d'apprentissage, nous avons choisi de les comparer selon un nouveau critère : l'évaluation de la politique courante tous les 25 000 pas d'apprentissage. L'évaluation de la politique s'est faite sur le robot : pendant 100 pas d'apprentissage, en utilisant la politique gloutonne dérivée des valeurs actuelles de Q, on a comptabilisé le nombre de récompenses obtenues. Pour que le robot ne se bloque pas dans une situation dans laquelle il est récompensé en continu (action "STOP"), toutes les 5 récompenses positives, nous avons déplacé aléatoirement le robot dans son environnement. Afin de limiter le bruit dans les résultats, chaque évaluation a été répété quatre fois, on a ensuite fait une moyenne pour obtenir le résultat. Les résultats obtenu lors de ces tests 4.21 semblent assez concluants, la méthode (2) permet d'obtenir plus de récompenses, à terme, que la méthode (1).

4.3.2 Phase 2 : Rapprochement d'un objet coloré

Nous n'avons pas eu le temps de faire des simulations réelles sur cette phase.

Dans cette phase le robot doit se rapprocher d'objets colorés. Nous avons choisi de

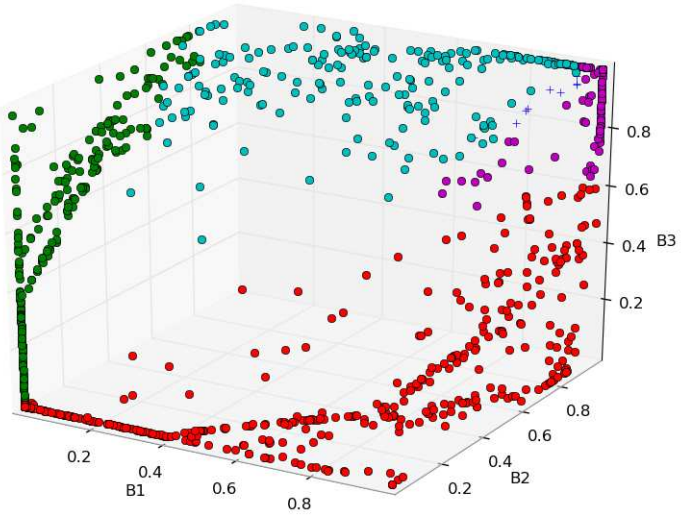

FIGURE 4.20 – Evaluation de la politique finale sur le jeu d'exemples (sans base)

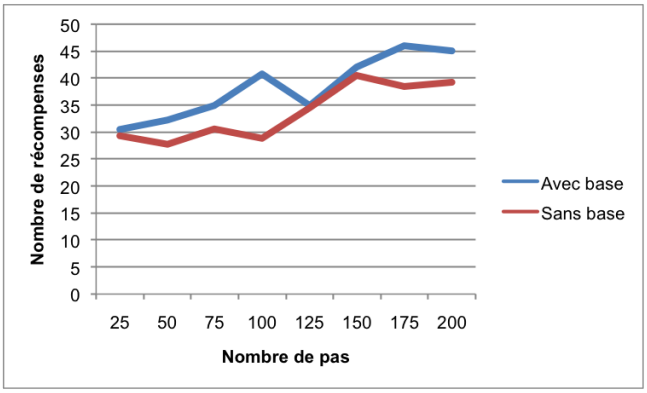

FIGURE 4.21 – Evaluation de la politique sur le robot

travailler avec des objets en forme d'ovoïde aplati (moins formellement, la forme devait se rapprocher de celle d'une toupille). Ce type de forme est particulièrement intéressant car il permet de conserver les avantages (parcours du domaine d'états) que nous avions trouvé à la forme du losange dans la phase précédente, mais en nous plaçant dans un contexte ou le robot peut tourner autour de l'objet.

Il est envisagé de faire évoluer les capacités de perceptions du robot en travaillant maintenant avec cinq bandes verticales réparties aux largeurs suivantes : 0,80,160,240,320 (voir figure 4.22).

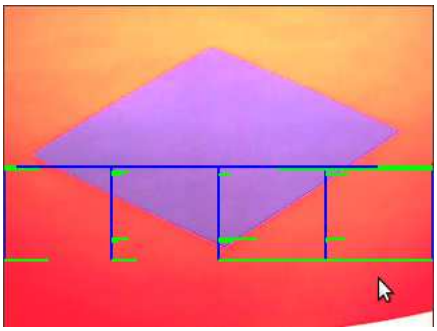

FIGURE 4.22 – Capture avec 5 bandes (l'environnement de travail n'est pas le bon)

Nous avons décidés d'ajouter les actions suivantes :

– a_6 : avancer (vitesse de rotation des roues : 1000 tics/s) ;

– a_7 : reculer à droite (vitesse de rotation des roues : 1000 tics/s) ;

La fonction de récompense du robot n'a pas été exprimée de façon explicite. Cependant, elle était supposée s'approcher fortement de la fonction de la phase 1 en ajoutant une condition pour que le robot ne se rapproche pas trop de la forme colorée (pour éviter les collisions). Il était prévu d'augmenter la valeur de la récompense dans cette phase par rapport à la récompense qui était attribuée au robot dans la phase 1 et qui avait pour valeur 1.

4.3.3 Phase 3 : Détection de formes

Dans cette partie nous avons prévu d'apprendre au robot à se rapprocher de triangulaires colorées.

Nous avons envisagé de rajouter les actions "avancer en tournant" dans cette phase, mais nous ne sommes pas convaincu qu'elles soient utiles.

La fonction de récompense aurait du être faite de telle façon à ce que le robot ne se rapproche que des triangles (et pas des autres formes de la bonne couleur). Nous comptions encore augmenter la valeur de la récompense qui est octroyée au robot.

Chapitre 5

Organisation du projet

5.1 Méthode de travail

Notre stage s'étant déroulé dans le domaine de la recherche, nous avons adopté une méthode de travail bien spécifique. En effet, le fil directeur du stage était connu, cependant les objectifs à court terme étaient en évolution constante, en fonction des idées que nous avions pour répondre à la problématique posée. Nous avons travaillés en collaboration avec nos encadrants, sur la base d'une réunion par semaine durant laquelle, nous abordions l'avancement dans la tâche en cours et les perspectives d'avancement possibles.

De manière générale, la réflexion vis-à-vis de la problématique a été faite de manière continue durant le stage, le développement des différentes composantes qui nous ont permis d'effectuer nos expérimentations a été plus segmentée. Chaque composante qui a été implémentée a été testée sur un problème "typique", ce qui nous a permis de s'assurer de son bon fonctionnement. La plupart des algorithmes que nous avons utilisés étaient paramétrables, une partie du travail a été de trouver les bon paramètres pour assurer un fonctionnement optimal.

Dans notre démarche de recherche, nous avons passé un certain temps à nous documenter sur les outils dont nous pouvions disposer pour répondre à la problématique. En cours, nous n'avions eu qu'une brève introduction à certain concepts de l'apprentissage par renforcement, une partie du travail a aussi été d'apprendre tous ces nouveaux concepts. Il nous a aussi fallu nous familiariser avec le domaine de la robotique et ses spécificités.

5.2 Gestion du temps

Nous avons commencés par faire une introduction sur l'apprentissage par renforcement. Durant cette introduction, nous avons lu un grand nombre d'articles et appris de nouveaux concepts sur ce domaine. Nous nous avons ensuite développés un certain nombre d'outils que nous avons testés par le biais de mini-projets (Ratmaze, Approximateur de fonctions, Mountain Car). Pour finir nous avons fait un grand nombre de simulations réelles en utilisant les outils que nous avons développé. Chaque simulation nous permettait d'obtenir des statistiques que nous avons du étudier en détail. La figure 5.1 donne le détail.

Tâche	Date de début	Date de fin
Initiation A/R et projet Ratmaze	29/03/10	09/04/10
Installation et test URBI / Robot	12/04/10	24/04/10
Bibliothèque de réseaux de neurones et tests	26/04/10	07/05/10
Bibliothèque Q-Learning	10/05/10	21/05/10
Projet Mountain Car et statistiques	25/05/10	11/06/10
Initiation à URBI SDK et Q-Learning en URBI Script	14/06/10	25/06/10
UObject réseau de neurones, sauvegarde des réseaux	28/06/10	01/07/10
Tests avec camera, jeu d'exemples, paramétrage	05/07/10	16/07/10
Apprentissage mixé et paramétrage	19/07/10	30/07/10
Simulation et test d'évolutions	02/08/10	20/08/10
Rapport de stage	23/08/10	27/08/10

FIGURE 5.1 – Gestion du temps

5.3 Outils utilisés

Nous disposions durant le stage d'une machine équipant GNU/Linux. Pour développer chaque composant qui nous a permis d'effectuer nos expérimentations, nous avons principalement programmé en C++ et en Urbi Script. Nous avons aussi appris à utiliser python et gnuplot pour créer des scripts qui nous ont permis d'obtenir un affichage exploitable de nos statistiques de simulations.

Concernant la gestion des sources, nous avons utilisé un système de contrôle de version, SVN, de manière systématique. Nous avons utilisé le serveur SVN mis à disposition par l'INRIA. Nous avons aussi tenu un carnet de bords ou nous notions précisément ce qui était fait chaque jour, ainsi qu'un décompte des heures.

Chapitre 6

Conclusion

Rappelons tout d'abord l'objet du stage : nous cherchions une façon d'apprendre à un robot à effectuer une tâche complexe en s'appuyant sur les principes de l'apprentissage développemental. Une question s'est alors posée : lorsque qu'un robot a acquis des connaissances dans une configuration donnée (tâches, capacités perceptives, capacités d'actions), comment transférer et utiliser ces connaissances dans une nouvelle configuration ?

Nous avons formulés (dans la section 2) un certain nombre de solutions envisageables pour répondre à cette problématique. Après avoir choisi quelques solutions à tester dans la pratique, nous les avons les avons expérimentées (dans la section **??**). Les simulations que nous avons effectuées se sont avérées probantes, l'apprentissage incrémental a été plus performant que l'apprentissage en repartant de zéro.

Cependant, nous n'avons effectués qu'un faible nombre d'expérimentations, et pour s'assurer des performances de la méthode que nous avons choisi de tester, il faudrait en réaliser plus. Nous rappelons aussi que la latence a été un grand problème dans nos expérimentation et qu'il se peut que cela ait en partie biaisé nos résultats.

Un certain nombre de scénarios et de critères d'évaluation ont été formulés pour tester les performances de l'apprentissage incrémental, il ne resterait donc plus qu'à les mettre en oeuvre pour les tester. Nous avons aussi réfléchi à des solutions pour régler le problème de la latence dans l'environnement de travail. Les deux solutions qui semblent les plus prometteuses sont : (1) d'embarquer le système d'apprentissage sur le robot et (2) de tenir compte de la latence dans le système d'apprentissage (plus précisément, en l'ajoutant au domaine des états S).

Il reste encore des problèmes à résoudre, parmi ceux-ci, le problème du changement de but en cours d'apprentissage parait le plus difficile. Les outils permettant de tester nos hypothèses étant en place, il ne reste plus qu'à affiner les solution envisagées dans la partie 3 et les expérimenter.

Bibliographie

Andrew G. Barto, Richard S. Sutton, and Christopher J. C. H. Watkins. Sequential decision probelms and neural networks. In *NIPS*, pages 686–693, 1989.

Richard Bellman. Dynamic programming. *Princeton University Press*, 1957.

O. Buffet, A. Dutech, and F. Charpillet. Incremental reinforcement learning for designing multi-agent systems. In *Fifth International Conference on Autonomous Agents, Agents'01 (poster session)*, 2001.

Olivier Buffet. *Une double approhe modulaire de l'apprentissage par renforcement pour des agents intelligents adaptatifs*. PhD thesis, Université Henri Poincaré, Nancy 1, September 2003. LORIA.

Kary Främling. Replacing eligibility trace for action-value learning with function approximation. In *European Symposium on Artificial Neural Networks*, 1997.

Ronald Howard. *Dynamic Programming and Markov Processes*, volume 3. MIT Press and Wiley, 1960.

J. Lettvin, H. Maturana, W. McCulloch, and W. H. Pitts. What the frog's eye tells the frog's brain. *Proceedings of the IRE*, 2(11) :1940–1951, 1959.

Max Lungarella, Giorgio Metta, Rolf Pfeifer, and Giulio Sandini. Developmental robotics : a survey. *Connection Science*, 15(4) :151–190, December 2003.

M. Puterman. *Markov Decision Processes : discrete stochastic dynamic programming*. John Wiley & Sons, Inc. New York, NY, 1994.

Satinder Singh, Richard S. Sutton, and P. Kaelbling. Reinforcement learning with replacing eligibility traces. In *Machine Learning*, pages 123–158, 1996.

R. Sutton and A. Barto. *Reinforcement Learning : An Introduction*. Bradford Book, MIT Press, Cambridge, MA, 1998.