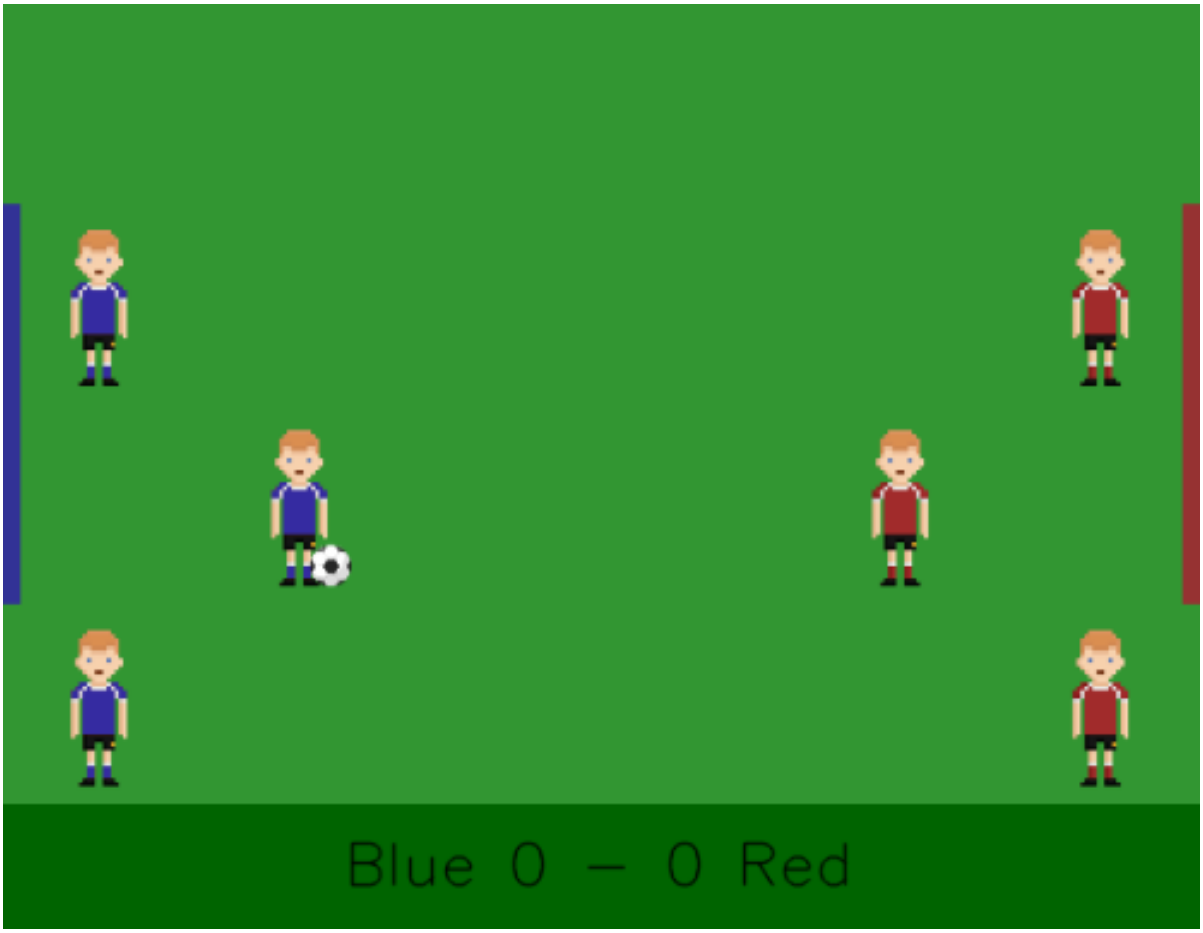
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

Génie Mathématique
Mars 2020

Projet de fin d'études

Apprentissage par renforcement multi-agents



Etudiant : David Albert
Encadrants : Jean-Phillipe Kotowicz, Laurent Vercouter

Table des matières

Introduction

Les systèmes multi-agents (SMA), d'une part, et l'apprentissage par renforcement, d'autre part, sont des sujets de recherche qui intéressent beaucoup la communauté scientifique de nos jours. Chacun d'eux s'inspire, à leur façon, du comportement animal pour développer des méthodes informatique permettant de simuler des comportements intelligents.

Tout d'abord, les systèmes multi-agents (SMA), par le biais d'un grand nombre d'agents peu complexes, tentent de faire émerger un comportement collectivement intelligent. C'est l'organisation des agents au sein d'un groupe qui va permettre d'améliorer les capacités de chacun.

Ensuite, nous savons très bien concevoir des agents capables de résoudre des problèmes ou d'accomplir des tâches spécifiques dans un cadre donné. Cependant, la conception de ce type d'agents peut nécessiter une certaine expertise humaine et donc avoir un coût de production élevé. De plus, ces agents sont en général propice à un seul type de tâche. Un nouvel agent devra donc être conçu pour chaque tâche et ceci peut s'avérer fastidieux. C'est pourquoi nous chercherons à concevoir des agents capables de s'adapter à une nouvelle situation. En d'autre termes, nous cherchons à donner aux agents la capacité d'apprendre le comportement à adopter. Il s'agit là du domaine de l'apprentissage par renforcement qui sera le centre de notre étude.

Dans ce projet de fin d'études, nous explorons les méthodes liées à l'apprentissage par renforcement multi-agents. Plus précisément, nous aborderons les techniques modernes d'apprentissage par renforcement dans le cas d'un agent seul et discuterons des cas où ces méthodes peuvent être adaptées au cas de plusieurs agents.

Le rapport est découpé en différentes parties. Les trois premières parties sont exclusivement théoriques et permettent de bien comprendre les enjeux et les solutions proposés pour que des agents évoluant dans un environnement donné apprennent à se comporter de manière adéquate. Ensuite une quatrième partie sera consacrée à la présentation de la framework python développée lors de ce projet et une cinquième et dernière partie présentera certains des tests réalisés avec l'API.

Partie I

Apprentissage par renforcement mono-agent

L'apprentissage par renforcement représente de nos jours près d'un tiers des recherches en intelligence artificielle. Ces dernières années, ce type de méthode a convaincu la communauté scientifique qu'il avait un grand potentiel à explorer. En effet, les techniques récentes ont montré leur capacité à résoudre des problèmes complexes mais aussi une grande capacité à pouvoir être réutilisée dans de nombreux domaines d'applications. Un exemple bien connu est la victoire d'AlphaGo (modèle développé par DeepMind) qui a battu le coréen Lee Se-Dol, champion du monde de jeu de Go.

1 Introduction à l'apprentissage par renforcement

L'apprentissage par renforcement est le fait d'apprendre un comportement à adopter dans une situation donnée. Cela revient à tenter de comprendre les conséquences de ses actions. Les conséquences, positives ou non, sont perçues sous la forme d'un simple signal de récompense. Dans cette section nous introduisons le formalisme de l'apprentissage par renforcement.

1.1 Système d'apprentissage par renforcement

Un système d'apprentissage par renforcement se compose de trois parties.

- **L'environnement** : Comme son nom l'indique il s'agit du contexte dans lequel l'agent (défini ci-après) évolu.
- **L'agent :** Il s'agit d'une entité pouvant agir selon un comportement qui lui est propre. Ses actions son étroitement liées à sa perception de l'environnement.
- **La récompense :** Le dernier élément d'un système d'apprentissage par renforcement est la récompense. La récompense est un signal perçu par l'agent et émis par l'environnement.

Citons, à titre d'exemples, quelques systèmes d'apprentissage par renforcement.

Ex 1. Une voiture autonome (l'agent) circulant (les actions) sur un circuit (l'environnement) avec comme récompense le nombre de tours de circuit qu'elle a réalisé. L'ensemble des états de ses capteurs forment l'observation.

Ex 2. Un chien (l'agent) qui reçoit de la nourriture (récompense) quand il fait les tâches que son maître lui demande (l'environnement). Dans ce cas, l'observation est ce qu'il a vu et entendu.

1.2 Les processus de Markov

Le but des méthodes d'apprentissage par renforcement est de contrôler le comportement d'un agent. Il faut donc modéliser cet agent dans son environnement. L'environnement étant incertain, il paraît raisonnable de faire appel à une modélisation stochastique du système. Ainsi, l'outil sur lequel se basent la majorité des algorithmes d'apprentissage par renforcement est le processus de décision Markovien (MDP) et ses extensions. Il s'agit d'une classe de processus de Markov, eux-même appartenant aux processus stochastiques.

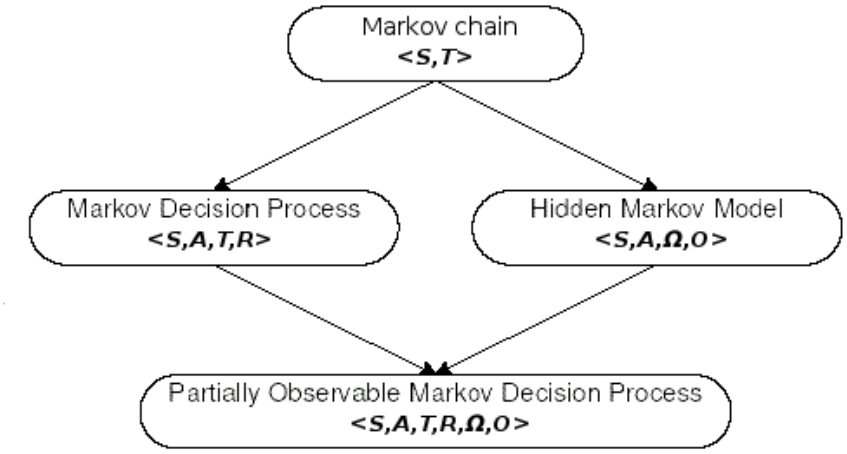


FIGURE I.1 – Différents processus de Markov

1.2.1 Processus stochastiques

Definition 1.1. ***Processus stochastique***
Un **processus stochastique** $(X_t)_{t\in T}$ est une famille de variables aléatoires X_t à valeurs dans un espace E indexée par le temps T.

En général nous considèrerons les *processus aléatoires en temps discret*. Dans ce cas $T = \mathbb{Z}^+ = \{1, 2, 3, ...\}$. Pour la suite, la notation S_t sera utilisée à la place de X_t pour mieux caractériser le fait que la variable aléatoire en question décrit l'état du système.

Nous pouvons citer plusieurs familles de processus aléatoires tels que : les marches aléatoires, les processus gaussiens ou encore les processus de Markov. Dans notre cas, nous ne nous intéresserons qu'à la dernière famille dont les dérivées sont utilisées pour mettre au point les algorithmes d'apprentissage par renforcement. Cette famille de processus, dits de Markov, est composée de l'ensemble des processus aléatoires vérifiant la propriété de Markov.

Propriété 1.1. ***Propriété de Markov***
La probabilité de transition d'un état s à un état s' est indépendante du passé :

$$\mathbb{P}\left(S_{t+1} = s' | S_0 = s_0, S_1 = s_1, ..., S_t = s_t\right) = \mathbb{P}\left(S_{t+1} = s' | S_t = s_t\right)$$

$\rightarrow$ On dit aussi que : "Le futur ne dépend que de l'état présent".

1.2.2 Chaîne de Markov (MC)

Le processus de Markov le plus simple correspond à un simple graphe d'états, doté d'une fonction de transition probabiliste. On parle dans ce cas de chaîne de Markov.

Plus formellement, une chaîne de Markov est d'écrite par un tuple $< \mathcal{S}, \mathcal{T} >$ où :

- $\mathcal{S}$: espace d'états possibles
- $\mathcal{T} : \mathcal{S} \times \mathcal{S} \rightarrow [0,1]$: fonction de transition probabiliste, i.e. $\mathbb{P}(S_{t+1} = s' \,|S_t = s), \ \ \forall (s, s')$

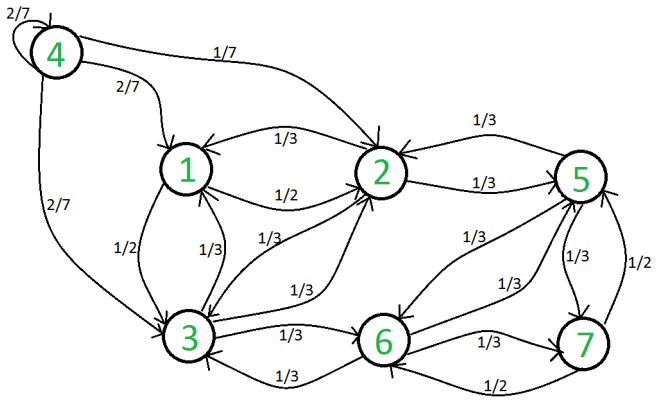


FIGURE I.2 – Exemple de chaîne de Markov

1.2.3 Processus de décision de Markov (MDP)

Les Processus de Décision de Markov (MDP) ont été introduit dans les années 1960 par Richard E. Bellman. Ce sont les processus stochastiques qui nous intéressent le plus dans cette partie car ils forment un outil mathématique très utilisé en apprentissage par renforcement. Ils permettent de formaliser la notion d'agent intéragissant avec l'environnement ainsi que la notion de récompenses (jusqu'alors non prise en compte dans les chaînes de Markov).

Formellement, un MDP est décrit par un tuple $< \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} >$ avec :

- $\mathcal{S}$: espace d'états
- $\mathcal{A}$: espace d'actions
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0,1]$: fonction de transition probabiliste, i.e. $\mathbb{P}(S_{t+1} = s' \,|S_t = s, A_t = a)$
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathbb{R} \rightarrow [0,1]$: récompense probabiliste (cas le plus général)

En pratique, on aura souvent une fonction de récompense déterministe qui dépend soit de l'état, soit du couple (état, action) ou encore du triplet (état, action, état suivant). On notera alors $\mathcal{R}(s, a, s')$.

Remarque : Les chaînes de Markov sont des MDPs particulières dans lequel il n'y a qu'une action possible ($card(\mathcal{A}) = 1$) et $\mathcal{R}(s, a, s') = cst \ \ \forall (s, a, s') \in \mathcal{S} \times \mathcal{A} \times \mathcal{S}$.

1.2.4 Modèle de Markov caché (HMM)

Un Modèle de Markov caché est similaire aux chaînes de Markov à la différence près que l'état du système n'est pas connu. On a, à la place, accès à une observation qui est liée par des lois probabilistes aux états.

Un Modèle de Markov Caché est donc défini par un tuple $< \mathcal{S}, \mathcal{T}, \Omega, \mathcal{O} >$ avec :

- $\mathcal{S}$: espace d'états possibles
- $\mathcal{T} : \mathcal{S} \times \mathcal{S} \rightarrow [0,1]$: fonction de transition probabiliste

  $\Rightarrow$ i.e. $\mathcal{T}(s, s') = \mathbb{P}(S_{t+1} = s' \,|S_t = s), \ \ \forall (s, s')$
- Ω : espace d'observations
- $\mathcal{O} : \Omega \times \mathcal{S} \rightarrow [0,1]$: fonction d'observation probabiliste, i.e. $\mathcal{O}(o, s) = \mathbb{P}(O_t = o \,|S_t = s)$

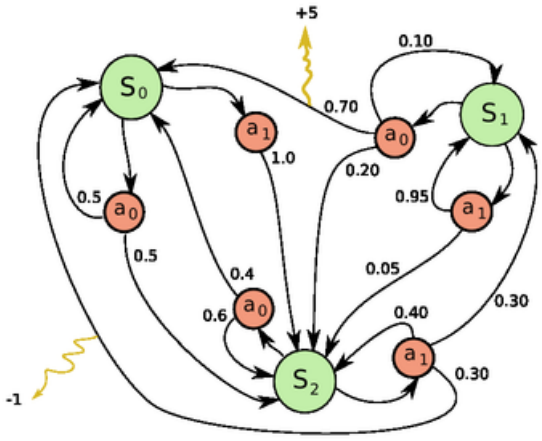


FIGURE I.3 – Exemple de processus de décision markovien (MDP)

1.2.5 Autres processus de Markov courants

D'autres extensions des processus de Markov de base (MC, MDP, HMM) sont parfois utilisées dans la pratique. A titre d'exemple on peut citer :

- **POMDP** : Mélange de HMMs et MDPs, les MDPs Partiellement Observables ajoutent l'idée que l'agent n'a de son environnement qu'une perception partielle. Il n'a accès qu'à une observation et non pas à l'état complet du système.
- **MMDP** : Les MDP Multiples sont une variante des MDPs adaptée au cas des systèmes comprenant plusieurs agents. Ces derniers sont moins présents dans la littérature que les **DEC-MDPs** (MDPs décentralisés) et les **jeux de Markov**. Nous présentons ces cas dans la partie III.1.

1.3 Résolution théorique d'un problème de décision de Markov

Dans cette section, nous nous reposons sur le formalisme des processus de décision markoviens présentés précédemment et introduisons des notions supplémentaires nécessaires à la planification dans les MDPs. Les MDPs basiques font de fortes suppositions sur le processus de décision. D'une part, il ne peut pas y avoir plus d'une entité qui prend des décisions et, d'autre part, cette entité doit avoir accès à l'état actuel du système. Ces suppositions sont parfois trop restrictives pour certaines applications mais permettent tout de même d'aboutir à de bons résultats dans beaucoup d'autres cas.

1.3.1 Définir l'objectif

Dans un système d'apprentissage par renforcement, nous pouvons poser l'heuristique suivante :

Propriété 1.2. ***Heuristique de l'objectif***
Tout objectif peut être formulé comme le fait de ***maximiser la récompense cumulée*** au cours du temps.

Ainsi, il n'y a pas besoin de définir l'objectif de manière explicite. Il nous suffit simplement de définir une fonction de récompense r adéquate et un critère de performance G_t.

1.3.2 Critère de performance

Le critère de performance G_t n'est autre qu'une aggrégation particulière de récompenses. De nombreux critères de performances peuvent être utilisés mais en général nous utilisons le gain cumulé comme critère de performance :

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$$

avec $\gamma \in [0, 1]$ et la possibilité d'avoir soit $T = +\infty$ ou $\gamma = 1$ (mais pas les deux).

Le paramètre γ est appelé *discount factor*. Dans le cas où $\gamma < 1$, il permet d'accorder plus d'importance aux récompenses à court terme. Ce processus est naturel chez l'animal qui préfèrera en général une récompense à court terme.

1.3.3 Problème de décision markovien

Un problème de décision markovien est un ensemble constitué : d'un MDP et d'un critère de performance G_t associé. Résoudre un problème de décision markovien consiste à trouver un comportement π (aussi appelé *politique*) qui, s'il est exécuté dans le Processus de Décision de Markov, optimise le critère G_t.

1.3.4 Fonction de valeur

Nous avons vu que dans un MDP la fonction de transition d'états $\mathcal{T}$ est en général stochastique. Ainsi, pour une même politique π et un même état initial, nous pouvons avoir un gain G_t différent à chaque exécution. Il nous faut donc introduire une manière d'évaluer une politique qui ne repose pas sur une seule trajectoire mais sur toutes. C'est ce qu'on appelle la fonction de valeur.

Mathématiquement parlant, la fonction de valeur associée à une politique π est définie comme le gain espéré en démarrant à un état $s \in \mathcal{S}$ et en suivant la politique π. Cette fonction représente donc à quel point être dans un état est bien pour l'agent pour un comportement donné. Les deux fonctions de valeur classiques sont les suivantes :

- Fonction de **valeur d'état** : $\forall s \in \mathcal{S}\ , \quad v_\pi(s) = \mathbb{E}_\pi\{G_t \mid S_t = s\}$
- Fonction de **valeur d'action** : $\forall (s, a) \in \mathcal{S} \times \mathcal{A}\ , \ \ q_\pi(s, a) = \mathbb{E}_\pi\{G_t \mid S_t = s, A_t = a\}$

Dans la suite de la partie théorique, nous utiliserons la fonction de valeur v_π. Cependant, tout ce qui sera présenté fonctionne également avec q_π.

Etant donné la définition de la fonction de valeur, nous pouvons redéfinir la résolution d'un problème de décision markovien comme le fait de trouver un comportement qui maximise la fonction de valeur. Un tel comportement est appelé ***politique optimale*** et on le note π_*.

$$\pi_*(s) = arg\max_\pi \ v_\pi(s)$$

1.3.5 Equations de Bellman

Nous pouvons réécrire la fonction de valeur telle que suit :

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}\left\{G_t \mid S_t = s\right\} \\
&= \mathbb{E}_\pi\left\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... \mid S_t = s\right\} \\
&= \mathbb{E}_\pi\left\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid S_t = s\right\} \\
&= \sum_{a,s',r} p\left(s, a, r, s'\right)\left[r + \gamma \mathbb{E}_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid S_{t+1} = s'\right\}\right] \\
&= \sum_a \pi\left(a \mid s\right) \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma v_\pi(s')\right] \quad (Bellman\ equation\ (i))
\end{aligned}$$

Cette première équation de Bellman nous donne une relation entre la valeur à un état s et les valeurs aux états suivants s'. De la même manière, nous pouvons déduire les **4 principales équations de**

Bellman :

$$(i) \quad v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s',r} p\left(s', r \mid s, a\right) \left[r + \gamma v_\pi(s')\right]$$

$$(ii) \quad v_*(s) = \max_{a \in \mathcal{A}(s)} \sum_{s',r} p\left(s', r \mid s, a\right) \left[r + \gamma v_*\left(s'\right)\right]$$

$$(iii) \quad q_\pi(s,a) = \sum_{s',r} p\left(s', r \mid s, a\right) \left[r + \gamma \sum_{a'} \pi\left(a' \mid s'\right) q_\pi(s', a')\right]$$

$$(iv) \quad q_*(s,a) = \sum_{s',r} p\left(s', r \mid s, a\right) \left[r + \gamma \max_{a'} q_*\left(s', a'\right)\right]$$

Ces équations sont la base théorique de nombreuses méthodes de résolution de problèmes de décision que nous verrons par la suite.

Nous allons maintenant nous intéresser aux différentes approches permettant d'optimiser une politique d'action. On peut en général mettre en avant trois approches différentes.

Dans un premier cas, nous pouvons utiliser un modèle de l'environnement si celui-ci est connu ou que nous pouvons facilement en avoir une approximation. Dans le cas contraire, nous devons élaborer des méthodes itératives pour construire de manière approchée un comportement optimal. Pour se faire, il y a deux possibilités qui sont : soit d'estimer la fonction de valeur optimale et de choisir l'action qui mène à la valeur la plus élevée, ou alors d'optimiser directement la politique d'action.

Nous présentons dans la suite du rapport ces différentes approches.

2 Utilisation d'un modèle de l'environnement *(Model-based)*

Une première approche de résolution du problème est d'utiliser le modèle de l'environnement. Dans le cas où le modèle de l'environnement est parfaitement connu, Bellman introduit dans ses travaux des méthodes de planification directe, aussi connu sous le nom de *programmation dynamique*. Cependant, le modèle de l'environnement n'est pas toujours connu. Une seconde méthode consiste donc à estimer un modèle de l'environnement si cela est possible. Il faut ensuite appliquer les méthodes de programmation dynamique classiques sur l'estimation du modèle de l'environnement.

2.1 Planification dans les MDPs

Lorsque le modèle de l'environnement est parfaitement connu (MDP connu). Nous ne parlons pas d'apprentissage mais plutôt de planification. Dans le cas où l'espace d'états et l'espace d'actions sont finis, il existe une solution optimale au problème de décision markovien. Plusieurs méthodes existent pour trouver cette solution.

2.1.1 Evaluation de la politique

Pour déterminer la solution optimale liée au MDP associé, nous avons besoin d'évaluer si un comportement donné est bon ou pas. Pour cela, nous devons estimer la fonction de valeur associée à la politique d'action.

L'évaluation itérative de la politique est montré dans l'algorithme 1 ci-après. Il est une conséquence directe de l'équation de Bellman (i).

Pour rappel, l'équation de Bellman (i) est donné par :

$$v_\pi(s) = \sum_a \pi\left(a \mid s\right) \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma v_\pi(s')\right]$$

La mise à jour itérative de la politique se fait de la façon suivante :

$$v_{k+1}(s) = \sum_a \pi\left(a \mid s\right) \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma v_k(s')\right]$$

On peut montrer que, pour un MDP fini, on a $v_k \underset{k \to +\infty}{\longrightarrow} v_\pi$ (sous condition que v_π existe).

Algorithm 1 Policy Evaluation

Require: a finite MDP $< \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} >$

procedure POLICYEVALUATION(π, γ, ϵ)
  $V(s) \leftarrow 0 \quad (\forall s \in \mathcal{S})$
  repeat
    $V' \leftarrow V$
    for all $s \in \mathcal{S}$ **do**
      $V(s) \leftarrow \sum_a \pi \ (a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma V'(s')\right]$
  until $\|V - V'\| < \epsilon$
  return V

2.1.2 Amélioration de la politique

Maintenant que nous savons évaluer n'importe quelle politique, il nous faut une stratégie pour construire la politique d'action optimale. Une méthode est l'algorithme d'itération sur les politiques. Celui-ci utilise le fait que $\pi'(s) = \arg\max_a \sum_{s',r} p\left(s', r \mid s, a\right) \left[r + \gamma v_\pi \ \left(s'\right)\right]$ est une meilleur politique

que π. Ainsi, en alternant, *évaluation* et *amélioration* de la politique nous aboutissons à l'algorithme 2 qui construit une politique d'actions optimale pour un MDP fini.

Algorithm 2 Policy Iteration

Require: a finite MDP $< \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} >$

procedure POLICYIMPROVEMENT(v_π, γ)
  for all $s \in \mathcal{S}$ **do**
    $\pi'(s) \leftarrow \arg\max_a \sum_{s',r} p\left(s', r \mid s, a\right) \left[r + \gamma v_\pi \ \left(s'\right)\right]$
  return π'

procedure POLICYITERATION(γ)
  Init π and π' randomly
  repeat
    $\pi \leftarrow \pi'$
    $V \leftarrow PolicyEvaluation(\pi, \gamma)$
    $\pi' \leftarrow PolicyImprovement(V, \gamma)$
  until $\pi = \pi'$
  return π

Le problème avec l'algorithme d'itération sur les politiques est qu'à chaque étape, nous devons évaluer la politique et que cela est assez coûteux en temps de calcul. Pour faire face à ce problème, l'algorithme 3 utilise l'équation d'optimalité de Bellman (iii) pour mettre directement à jour la fonction de valeur sans construire à chaque fois une nouvelle politique.

Algorithm 3 Value Iteration

Require: a finite MDP $< \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} >$

procedure VALUEITERATION(v_π, γ, ϵ)
  $V(s) \leftarrow 0 \quad (\forall s \in \mathcal{S})$
  $V' \leftarrow V$
  repeat
    for all $s \in \mathcal{S}$ **do**
      $V' \leftarrow V$
      $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a) \left[r + \gamma V'(s')\right]$
  until $\|V - V'\| < \epsilon$
  return V

Ce nouvel algorithme est bien plus efficace et converge vers la fonction de valeur optimale d'après l'équation d'optimalité de Bellman (pour un MDP fini).

Nous avons dans cette première partie présenté brièvement deux algorithmes de planification dans les MDPs. Ces algorithmes sont très performant mais nécessitent de fortes hypothèses. En effet, nous devons avoir accès au modèle complet du système. Les algorithmes d'itération sur la politique et d'itération sur la valeur sont des exemples de **programmation dynamique**.

2.2 Estimer un modèle de l'environnement

Dans certains cas plus complexes, le modèle de l'environnement n'est pas directement accessible. Quand cela est possible, il peut être intéressant de l'estimer. Pour se faire, une méthode consiste à garder en mémoire toutes les occurences du modèle (s, a, s', r) et à construire les probabilité associées à chaque transition. Après avoir fait cela suffisamment de fois, nous pouvons utiliser les méthodes de programmation dynamique sur notre approximation de l'environnement pour construire une politique sous-optimale.

3 Méthodes d'estimation de la fonction de valeur *(Value-based)*

Dans cette section, nous nous intéressons aux méthodes d'apprentissage en ligne qui cherchent à approcher la fonction de valeur optimale v_* ou q_*. Ces méthodes utilisent simplement l'expérience passée sous la forme de séquences de type *état/action/récompense/état suivant* pour estimer la fonction de valeur. Dans cette section, nous supposons que le processus d'évolution de l'environnement est un MDP. Dans le cas contraire, les méthodes présentées ci-après pourront être utilisées mais elles convergeront vers une politique sous-optimale.

3.1 Méthode de Monte-Carlo

Tout comme dans la partie *Planification*, nous allons nous intérroger sur comment évaluer v_π. Une possibilité consiste à revenir à la définition :

$$v_\pi(s) = \mathbb{E}_\pi\left(G_t \mid S_t = s\right) \approx \frac{1}{N}\sum_{i=1}^{N} G_{t_i}$$

Dans le cas où la tâche est épisodique, une idée est d'utiliser une moyenne mobile pour évaluer itérativement le gain moyen. Une mise à jour de la forme suivante en résulte :

$$v_t(S_t) \leftarrow (1-\alpha_t)\, v_{t-1}(S_t) + \alpha_t\, G_t = v_{t-1}(S_t) + \alpha_t\left(\mathbf{G_t} - v_{t-1}(S_t)\right)$$

3.1.1 Choisir α_t

La question maintenant est de savoir comment choisir le paramètre α_t.

Propriété 3.1. *Conditions de Robbins-Monro*

$$(1) \quad \sum_{t=1}^{T} \alpha_t \underset{T\to\infty}{\longrightarrow} \infty$$
$$(2) \quad \sum_{t=1}^{\infty} \alpha_t^2 < \infty$$

On peut montrer que $v_t(s) \underset{\infty}{\to} v_\pi(s)$, si α_t vérifie la condition de Robbins-Monro. Ainsi, un choix adéquate est $\alpha_t = \frac{1}{t}$.

3.2 Méthodes *on-line*

La méthode Monte-Carlo nécessite d'attendre la fin de l'épisode pour faire la mise à jour. Si la tâche est continue, cette méthode ne s'applique donc pas. Pour faire face à cela, on utillise de méthodes mettant à jour la fonction de valeur après chaque séquence d'échantillons du type $S_t,\ A_t,\ R_{t+1},\ S_{t+1}$.

3.2.1 TD-learning

Le cas de base consiste à utiliser l'estimateur de v_π donné par $R_{t+1} + \gamma v(S_{t+1})$ au lieu de G_t. C'est une estimation moins précise de v_π mais elle ne nécessite pas d'avoir accès à l'épisode entier. Ainsi, cette méthode de mise à jour se fait en ligne (***on-line***). La règle de mise à jour de cette méthode, appelée TD-learning, est la suivante :

$$v_{t+1}(S_t) \leftarrow v_t(S_t) + \alpha_t\left[\,\mathbf{R_{t+1}} + \gamma \mathbf{v_t}(\mathbf{S_{t+1}}) \; - v_t(S_t)\right]$$

La méthode TD peut être vue comme une combinaison de la méthode Monte-Carlo (moyenne mobile) avec l'approximation itérative des méthodes de programmation (algorithme d'itération sur les valeurs 3). Sous certaines conditions, il y a convergence de la méthode vers v_π.

Les méthodes SARSA et Q-learning se distinguent de la méthode TD-learning car elles utilisent la fonction de valeur q et donc sont basées sur les équations de Bellman (iii) et (iv). Ces deux méthodes se distinguent par leur règle de mise à jour qui sont présentées dans la suite.

3.2.2 SARSA

La règle de mise à jour de q pour la méthode SARSA est la suivante :

$$q_{t+1}(s,a) = q_t(S_t, A_t) + \alpha_t(\ \mathbf{R_{t+1}} + \gamma \mathbf{q_t}(\mathbf{S_{t+1}}, \mathbf{A_{t+1}}) - q_t(S_t, A_t))$$

Et $q_t \underset{t\to\infty}{\longrightarrow} q_\pi$ sous les mêmes conditions que TD-learning.

3.2.3 Q-learning

Quant à l'algorithme du Q-learning, sa règle de mise à jour est :

$$q_{t+1}(s,a) = q_t(S_t, A_t) + \alpha_t(\ \mathbf{R_{t+1}} + \gamma \max_{\mathbf{a'}} \mathbf{q_t}(\mathbf{S_{t+1}}, \mathbf{a'}) - q_t(S_t, A_t))$$

Et $q_t \underset{t\to\infty}{\longrightarrow} q_*$ sous les mêmes conditions que TD-learning. Cet algorithme est très intéressant car il permet d'estimer q_* au lieu de q_π. Cela nous permet, dans le cas d'un MDP fini, d'avoir directement la politique d'action optimale par $\pi\,(a \mid s) = \mathbb{1}_{a=\, arg\max_{a\in\mathcal{A}} q_*(s,a)}$.

Ces méthodes n'utilisent que la séquence suivante pour mettre à jour leur fonction de valeur. Cependant, en fonction de l'environnement, il peut être astucieux que la mise à jour ne prenne pas en compte uniquement la dernière séquence d'action/récompense/état mais n séquences consécutives. C'est pourquoi la généralisation de chacune de ces méthodes est intéressante.

3.3 Généralisation : n-step learning

On rappelle la formulation théorique du gain total : $G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$.

Les méthodes TD, SARSA et Q-learning peuvent être généralisées au cas où on ne considère plus uniquement la récompense suivante mais les n récompenses suivantes (voir figure I.4b). Dans le cas du TD-learning, contrairement à la méthode dites *1-step*, la cible n'est plus $R_{t+1} + \gamma v(S_{t+1})$ mais la valeur $G_t^{(n)}$. Un estimateur de la valeur $v_\pi(S_t)$ est $G_t^{(n)}$ et est égal à :

$$G_t^{(n)} = r_{t+1} + \gamma r_{t+2} + ... + \gamma^{n-1} r_{t+n} + v_{t+n-1}(S_{t+n})$$

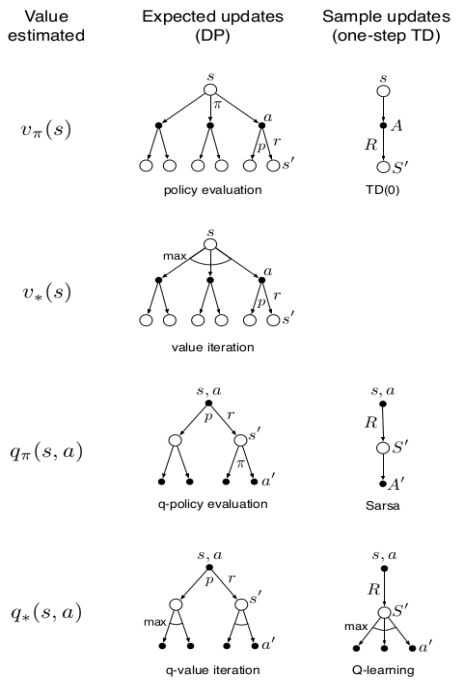
La mise à jour de la fonction de valeur se fait donc de la façon suivante :

$$v(S_t) \leftarrow v(S_t) + \alpha(\ \mathbf{G_t^{(n)}} - v(S_t))$$

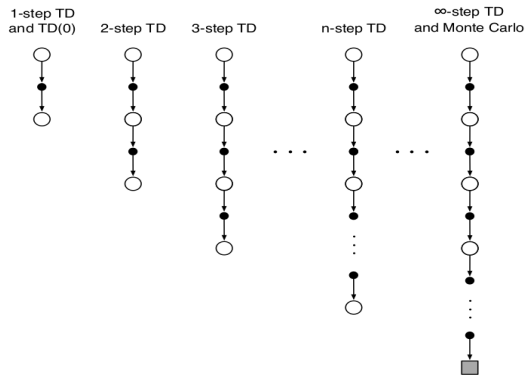
Remarque 1 : On peut considérer la méthode Monte-Carlo comme étant une méthode de type ∞-step TD-learning.

Remarque 2 : Plus n est grand et plus le biais est faible mais plus la variance est grande.

De la même manière, il existe des généralisations des méthodes SARSA et Q-learning que nous ne détaillerons pas dans ce rapport.



(a) Méthodes d'estimation de la fonction de valeur



(b) n-step TD-learning

FIGURE I.4 – Schéma explicatif

3.4 Exploration vs Exploitation

En apprentissage par renforcement, un point central de tout algorithme dit *sans modèle* est l'**exploration**. Nous venons de voir l'algorithme du Q-learning. Cet algorithme converge vers q_* mais la convergence peut être lente si le comportement de la politique est inefficace.

Prenons l'exemple d'un agent dans un labyrinthe. L'agent reçoit une récompense de +1 quand il atteint la sortie du labyrinthe. Si l'agent suit un comportement aléatoire à chaque épisode, il mettra beaucoup d'itérations avant de trouver la sortie. La fonction de valeur mettra donc du temps avant de pouvoir être mise à jour. Dans ce cas, il paraît intelligent d'utiliser l'expérience passée pour générer des échantillons plus utiles, c'est à dire des échantillons générés d'une part avec des actions choisies aléatoirement (**exploration**) et d'autre part avec des actions suivant la politique actuelle (**exploitation**).

3.4.1 ϵ-greedy

L'exploration dites $\epsilon - greedy$ est la méthode d'exploration la plus utilisée. Elle consiste à suivre une politique aléatoire avec la probabilité ϵ et de suivre la politique actuelle (choisir l'action qui maximise Q(s,a)) avec la probabilité $1 - \epsilon$. Pour contrôler l'exploration, nous choisirons $\epsilon \in [0, 1]$.

3.4.2 Distribution de Boltzmann

Une seconde méthode d'exploration consiste à choisir une action aléatoire selon la distribution de Boltzmann. Plus précisemment, nous choisirons une action proportionnelle à $e^{\beta \times Q_t(S_t,a)}$. Pour contrôler l'exploration, nous choisirons $\beta \in \mathbb{R}^+$. Le comportement est purement aléatoire lorsque $\beta = 0$ et $\beta = +\infty$ revient à suivre comportement déterministe.

3.5 Approximation de la fonction de valeur par réseau de neurones

Les méthodes vues jusqu'alors sont applicables dans le cas où il y a un nombre n fini d'actions et un nombre m fini d'états possibles. Dans ce cas particulier et à condition que les valeurs n et m ne soient pas trop grandes, nous pouvions garder en mémoire les valeurs associées à chaque état. Cependant, dans le cas où le nombre d'états est relativement élevé ou que l'espace $\mathcal{S}$ est continu, nous cherchons une approximation de la fonction de valeur. Différents modèles d'approximation de fonctions existent. Dans notre cas, nous utiliserons des réseaux de neurones artificiels car ils permettent d'approcher un grand nombre de fonctions non linéaires.

Dans ce cas, la fonction de valeur stockée en mémoire est une fonction paramétrique, de paramètres $\theta \in \mathbb{R}^N$, que l'on note v_θ (respectivement q_θ). Le but est d'ajuster les paramètres θ pour que $q_\theta \approx q_*$ et ainsi pouvoir estimer la valeur optimale d'états que nous n'avons pas encore explorés.

On ne considère ici que le cas du 1-step Q-learning pour lequel l'estimateur de la fonction de valeur est donné par $r_{t+1} + \gamma \max_{a'} q_\theta(S_{t+n}, a')$.

La règle de mise à jour s'écrit désormais :

$$q_{\theta_{t+1}}(S_t) = q_{\theta_t}(S_t) + \alpha_t \left[r_{t+1} + \gamma \max_{a'} q_\theta(S_{t+n}, a') - q_{\theta_t}(S_t, A_t) \right] \nabla_\theta q_\theta(S_t, A_t)$$

Cette mise à jour se résume sous la forme d'un problème de minimisation de la fonction :

$$\mathcal{L}(\theta) = \frac{1}{2} \left[r_{t+1} + \gamma \max_{a'} q_\theta(S_{t+n}, a') - q_\theta(S_t, A_t) \right]^2$$

Dans la pratique, on pourra utiliser un algorithme de descente de gradient pour minimiser cette fonction.

3.5.1 Deep Q-Learning

L'algorithme DQN a été introduit par une équipe de DeepMind. Les auteurs du papier [6] utilisent un réseau de neurones convolutif comme approximation de la fonction de valeur optimale q_*. Ainsi, l'algorithme en question apprend à jouer à des jeux Ataris en utilisant uniquement l'image et non pas des caractéristiques prédéfinies. De plus, le papier présente quelques astuces importantes telles que l'utilisation d'un buffer d'expérience ou encore l'utilisation d'un réseau de neurones cible.

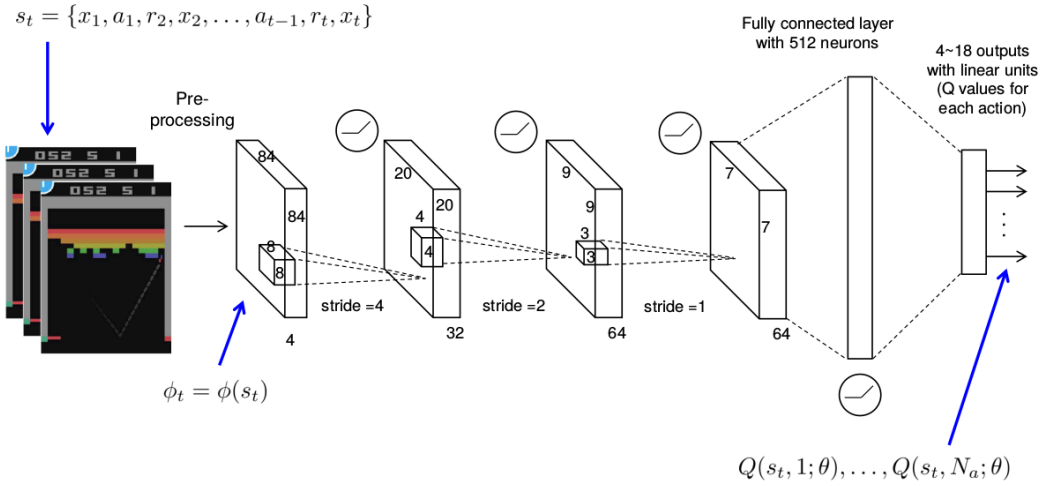


FIGURE I.5 – Architecture du modèle DQN

3.5.2 Les apports de DQN

L'algorithme DQN fonctionne notamment grâce à deux astuces qui permettent de stabiliser fortement l'apprentissage.

- Mieux utiliser l'expérience : Experience Replay

Une première astuce est ce qu'on appelle l'Experience Replay. L'experience replay est une astuce purement statistique. Elle permet d'utiliser de manière efficace les informations des échantillons tirés. En effet, le principe consiste à réutiliser des échantillons passés qui ont été sauvegardés sous la forme (s_t, a_t, r_t, s_{t+1}) pour apprendre avec les mêmes échantillons de multiple fois. Cette pratique permet d'accélérer l'apprentissage car au lieu de mettre à jour le réseau pour chaque échantillon une fois, il sera mis à jour plusieurs fois pour un même échantillon. De plus, nous prenons des lots d'échantillons de manière aléatoire dans le buffer. De cette manière, nous évitons la corrélation des échantillons.

- L'utilisation d'un modèle cible

Dans DQN, deux réseaux de neurones sont utilisés. Ceux-ci ont la même architecture mais ne partagent pas les mêmes paramètres. Le but est de mettre à jour toujours le même réseau Q et d'utilisé le second $\hat{Q}$ pour calculer la valeur cible $r_{t+1} + \gamma \max_{a'} q_\theta(S_{t+n}, a')$ sans que celle-ci ne change constamment à cause des mises à jour de Q.

4 Application : Lunar Lander

Pour tester notre implémentation de DQN (voir Partie IV), nous avons utilisé l'environnement gym [9] nommé *LunarLander-v2*. Le but est d'apprendre une politique d'action permettant au module d'alunir. L'observation consiste en six valeurs continues (position $\times 2$, vitesse $\times 2$, angle et vitesse angulaire) et deux valeurs discrètes qui indiquent si la patte droite (resp. gauche) touche le sol. D'autre part, il y a quatre actions possibles qui consistent à soit ne rien faire, soit à actionner l'un des trois propulseurs (gauche, principal ou droite).

L'apprentissage s'est fait sur 100 000 itérations avec les paramètres suivants :

- Modèle dense : 2 couches cachées de 64 et 32 neurones
- *Taux d'apprentissage* : 0.001 et γ : 0.99
- *Exploration* : $\epsilon - greedy$ dégressif de 1. à 0.1
- *Fréquence de mise à jour du modèle cible* : 1000
- *Taille du buffer d'expérience* : 5000

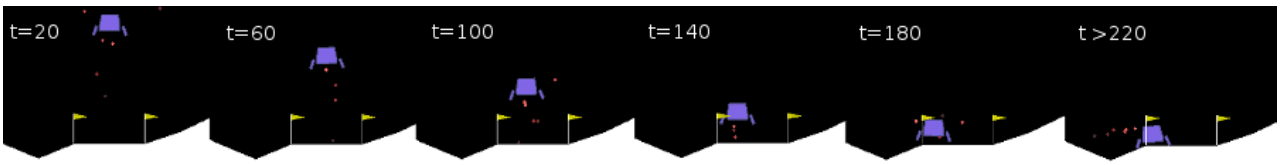


FIGURE I.6 – Exemple d'exécution de la politique apprise

On remarque dans le test de la figure I.6 que la politique a bien appris à poser le module lunaire approximativement entre les drapeaux et de façon "sécurisée". Dans un premier temps, le module élabore une descente rapide. Ensuite, aux alentours de l'itération 140, il ralentit pour éviter un atterrissage brutal qui engendrerait une récompense très négative. Finalement, le module n'a pas totalement réussi à se poser entre les drapeaux. Il tente donc désespéremment ($t > 220$) de se recentrer en actionnant sans cesse le propulseur gauche.

5 Méthodes d'optimisation directe de la politique *(Policy-based)*

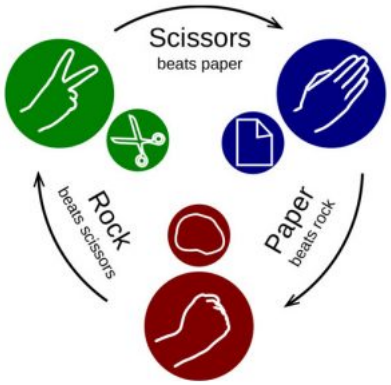
Toutes les méthodes vues jusqu'alors ne sont applicables que dans le cas de problèmes de décision de Markov. Cela implique, par exemple, que nous ayons accès à chaque instant à l'état de l'environnement. Le problème c'est, qu'en réalité, il est très rare d'avoir accès à l'état exact du système. Nous avons introduit dans la première partie une extension des MDPs qui correspond bien à ce cas : les **POMDPs**. Dans le formalisme des POMDPs, l'agent n'a pas accès directement à l'état du système mais a simplement une observation de celui-ci.

5.1 Le problème

Dans le cas où l'agent n'a accès qu'à une observation de son environnement, il est fort probable qu'il y ait deux états pour lesquels l'observation soit la même. Dans ce cas là, comment savoir quelle action choisir sachant que l'on peut être indépendemment dans l'un ou l'autre état.

5.1.1 Exemple du "Pierre-Papier-Ciseaux"

Le jeu "Pierre-Papier-Ciseaux" est un jeu qui se joue à deux. Les deux joueurs choisissent simultanément un des trois coups possibles en le symbolisant de la main. Les règles sont les suivante : la pierre bat les ciseaux, les ciseaux battent la feuille, la feuille bat la pierre.



Ce jeu est très simple, et pourtant, si on utilise une des méthodes vues prédemment, nous ne pourrons aboutir à la stratégie optimale. Ceci vient du fait que les méthodes vues jusqu'alors permettent d'obtenir uniquement des politiques déterministes, qui sont optimales dans le cas d'un MDP. Or, dans ce cas, nous n'avons pas du tout accès à l'état de l'environnement, qui serait de connaître le choix de l'adversaire. La stratégie optimale à adopter serait de choisir à chaque manche une action (pierre, papier ou ciseaux) de façon purement aléatoire. Il nous faut donc un moyen de construire des politiques stochastiques.

5.2 Méthodes de gradient de la politique

On considère la politique $\pi_\theta(a \mid s)$ dépendant de paramètres θ (par exemple un réseau de neurones). L'objectif va être de mettre à jour les paramètres θ de façon à maximiser l'espérance du gain $J(\theta) = \mathbb{E}_{\pi_\theta}[G_1]$ au début de la tâche. Le problème d'optimisation est donc le suivant :

Trouver $\hat{\theta}$ tel que $\hat{\theta} = arg\max_\theta \mathbb{E}_{\pi_\theta}[G_1]$

Or, d'après le ***théorème de gradient de politique***, on a :

$$\nabla J(\theta) = \sum_{t=1}^{\infty} \mathbb{E}_{\pi_\theta}\left[\gamma^{t-1} q_{\pi_\theta}(S_t, A_t) \nabla_\theta \log \pi_\theta(A_t \mid S_t)\right] \tag{I.1}$$

Dans le cas de tâches épisodiques, cela donne :

$$\nabla J(\theta) = \sum_{t=1}^{T} \mathbb{E}_{\pi_\theta}\left[\gamma^{t-1} q_{\pi_\theta}(S_t, A_t) \nabla_\theta \log \pi_\theta(A_t \mid S_t)\right] \tag{I.2}$$

Tous les algorithmes d'apprentissage direct de la politique sans modèle de l'environnement utilisent ce théorème pour mettre à jour les paramètres θ par montée de gradient stochastique. La principale différence entre ces algorithmes est la façon dont est approximé q_{π_θ}. Le problème de ces algorithmes est qu'ils engendrent généralement une grande variance.

5.2.1 Utilisation d'une valeur de référence

Pour diminuer la variance, il est d'usage de soustraire à $q_{\pi_\theta}(S_t, A_t)$ une valeur de référence $b(S_t)$, fonction de l'état. On peut montrer que le gradient n'est pas affecté par l'ajout de cette valeur de référence. En effet, on a :

$$\mathbb{E}_{\pi_\theta}\left[b(S_t)\nabla_\theta \log \pi_\theta\left(A_t \mid S_t\right)\right] = 0$$

Donc,

$$\begin{aligned}\nabla J\left(\theta\right) &= \sum_{t=1}^{T} \mathbb{E}_{\pi_\theta}\left[\gamma^{t-1} q_{\pi_\theta}(S_t, A_t)\nabla_\theta \log \pi_\theta\left(A_t \mid S_t\right)\right] \\ &= \sum_{t=1}^{T} \mathbb{E}_{\pi_\theta}\left[\gamma^{t-1}\left\{q_{\pi_\theta}(S_t, A_t) - b\left(S_t\right)\right\}\nabla_\theta \log \pi_\theta\left(A_t \mid S_t\right)\right]\end{aligned}$$

Un bon choix de valeur de référence est $b(S_t) = v_{\pi_\theta}\left(S_t\right) = \mathbb{E}_{\pi_\theta}\left(q_{\pi_\theta}(S_t, A_t) \mid S_t\right)$. La fonction de valeur étant la moyenne probabiliste du gain, la variance est sûre de diminuer.

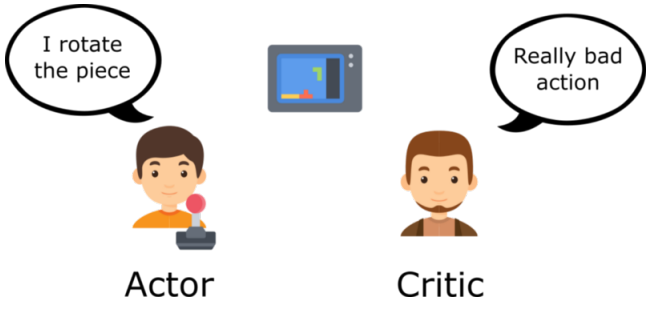
5.3 L'algorithme REINFORCE

L'algorithme REINFORCE, aussi connu sous le nom d'algorithme de gradient de politique de Monte-Carlo, utilise G_t comme estimation de $q_{\pi_\theta}\left(S_t, A_t\right)$.

5.4 L'algorithme Actor-Critic

L'algorithme Actor-Critic, quant à lui, utilise une méthode vue dans la section précédente pour estimer $q_{\pi_\theta}\left(S_t, A_t\right)$. Ainsi, cet algorithme est souvent considéré comme une combinaison des méthodes basées sur la valeur et des méthodes de gradient de la politique.

Interprétation : Le nom *Actor-Critic* (en français Acteur-Critique) vient de l'interprétation que l'on peut faire de ce procédé d'apprentissage. L'idée est la suivante. D'une part, l'acteur choisi et exécute des actions. C'est le rôle de notre politique π_θ. D'autre part, le critique observe le comportement de l'acteur et lui indique si l'action effectuée est bonne. C'est le rôle de notre fonction de valeur q ou v. En quelque sorte, l'acteur s'améliore grâce au jugement émis par le critique.



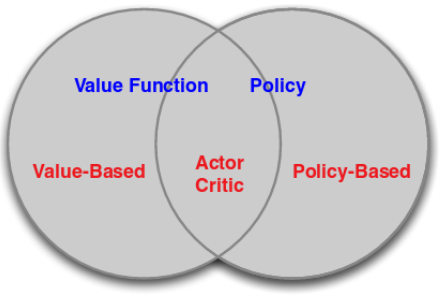


FIGURE I.7 – Relation entre *Value-based* RL et *Policy-based* RL

5.5 L'algorithme Deterministic Gradient Policy (DPG)

D'autres algorithmes de gradient de politique existent. Nous n'en détaillons ici qu'un seul qui nous servira dans la partie apprentissage multi-agents. C'est l'algorithme de gradient de politique déterministe (DPG). Comme son nom l'indique, il dérive de l'équation (I.2) dans le cas de politiques déterministes.

Dans ce cas, on notera la politique déterministe $\mu_\theta : \mathcal{S} \rightarrow \mathcal{A}$ et on peut montrer que :

$$\nabla_\theta J(\theta) = \mathbb{E}_{\mu_\theta} \left\{ \nabla_\theta \mu_\theta(s) \, \nabla_a q^\mu(s,a)|_{a=\mu_\theta(s)} \right\}$$

L'équation ci-dessus nous indique qu'il est nécessaire de pouvoir dériver par rapport aux actions. Ceci implique que l'espace d'action $\mathcal{A}$ doit être continue. Cet algorithme est donc intéressant car il permet d'étendre les méthodes précédentes au cas de politiques déterministes et d'espace d'actions continu.

5.5.1 Exploration dans un espace continu

Apprendre à choisir une action dans un espace continu peut être très utile dans des applications réelles telle qu'en robotique. Cependant, cela apporte des problématiques supplémentaires. Comment explorer de nouvelles actions ? Dans les cas discrets nous utilisons en général une action aléatoire parmis les p actions possibles. Est-il possible d'utiliser la même méthode dans le cas continu ? Oui, mais cela n'est pas la méthode appropriée. En faisant cela, notre agent aurait un comportement anarchique et il serait peu probable qu'il répète plusieurs fois une série d'action intéressante. Une meilleure approche est d'utiliser un processus de Ornstein-Uhlenbeck pour l'exploration. Ce processus aléatoire est un processus de type Gauss-Markov, c'est-à-dire qu'il vérifie à la fois les propriétés des processus de Markov et celles des processus gaussiens. Il est en général utilisé pour décrire le mouvement d'une particule massive dans un fluide, celle-ci ayant une mouvement aléatoire mais relativement structuré.

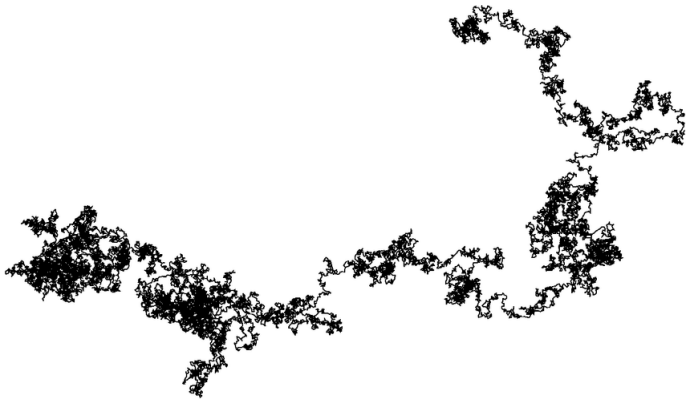


FIGURE I.8 – Exemple de mouvement brownien

Le processus de Ornstein–Uhlenbeck x_t, avec le terme supplémentaire de dérapage μ, est défini par l'équation différentielle stochastique :

$$dx_t = \theta \times (\mu - x_t)\, dt + \sigma dW_t$$

où W_t est un processus de Wiener, θ et σ sont des paramètres strictement positifs. En pratique, le processus est discrétisé de la manière suivante :

$$x_{t+1} - x_t = \theta \times (\mu - x_t) + \sigma W_t$$

With $W_t \sim \mathcal{U}_{[0,1]}$.

Partie II

Les systèmes multi-agents

1 Introduction aux systèmes multi-agents

Un système multi-agents (SMA) est un système composé d'un ensemble d'agents qui intéragissent entre eux dans un environnement commun. Les SMA ont été mis en avant grâce aux études des comportements sociaux et notamment l'*intelligence en essaim* et l'*intelligence collective*.

1.1 L'intelligence en essaim

L'intelligence en essaim désigne le fait qu'une population soumise à des règles simples fasse apparaître un comportement collectif qui semble complexe et intelligent. Les individus au sein de la population agissent selon des règles fixées et simples. Pourtant, du point de vue du groupe, on distingue un comportement collectif intelligent.

Un exemple classique de ce type de comportement est celui des colonies de fourmis. Ces petits insectes sont capables de trouver le chemin le plus court jusqu'à une source de nourriture sans même avoir de notion des distances. Pour se faire, elles vont utiliser des règles simples.

1.1.1 Exemple des fourmis

On considère, comme dans la figure II.1, quatre chemins possibles menant à de la nourriture. Le principe est que lorsqu'une fourmi trouve de la nourriture, elle revient à la fourmilère en déposant sur le chemin du retour une trace chimique (des phéromones). Les fourmis sont attirées par les phéromones et sont donc incitées à emprunter ce chemin. Or, dans le cas du chemin le plus court, les fourmis le choisissant auront plus vite fait de le marquer fortement que les fourmis passant par un autre chemin. Le résultat est que les fourmis seront plus attirées par ce chemin et le choisiront donc plus souvent. Ainsi, le phénomène s'amplifie jusqu'à ce que toutes les fourmis l'emprunte sans réellement savoir pourquoi (comportement global intelligent).

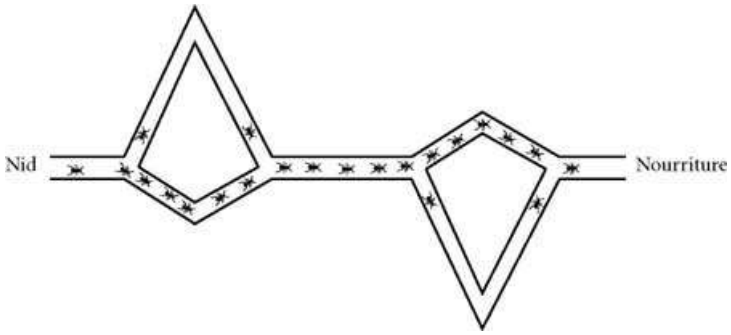


FIGURE II.1 – Exemple des colonies de fourmis

1.2 L'intelligence collective

L'intelligence collective, tout comme l'intelligence en essaim, a pour but de faire apparaître une intelligence globale. La différence se fait à l'échelle de l'individu. Dans l'intelligence en essaim les individus sont soumis aux mêmes règles et se comportent de façon automatique. On parle d'intelligence collective quand chaque agent peut avoir une connaissance différente du monde et va favoriser l'interaction avec les autres pour mettre en place un comportement collectif intelligent.

Les systèmes collectifs sont plus ou moins sophistiqués suivant la nature des agents mais partagent des caratéristiques communes qui sont :

- des règles simples que suivent les agents
- des interactions qui représentent les liens entre agents
- une information locale et limitée du point de vue des agents
- un but utile à l'ensemble des individus qui incite à collaborer

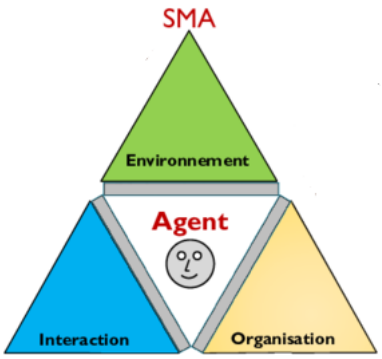
1.2.1 Exemple du football

L'exemple de l'équipe de football est un bon exemple d'intelligence collective. Dans ce cas, chaque agent a un rôle et une vision du jeu qui lui est propre. L'objectif est de mettre en place une stratégie collective basée sur la communication entre les coéquipiers pour marquer le plus de buts. L'exemple de la compétition Robocup est présenté dans la figure II.2.

FIGURE II.2 – Competition *Robocup*

1.3 Les systèmes multi-agents

La motivation derrière les systèmes multi-agents est donc que certains problèmes peuvent être résolus par un ensemble d'agents de faible complexité plutôt que par un seul très complexe. Ainsi, dans un SMA, chaque agent n'a souvent qu'une information incomplète ou des capacités insuffisantes à la résolution du problème dans son intégralité. C'est en coopérant avec les autres que le problème va pouvoir être résolu.



L'étude des comportements collectif est le principal intérêt des SMA. En particulier, ce domaine étudie les interactions avec l'**environnement**, les **interactions** entre agents, et l'**organisation** collective.

Comme montré sur la figure II.3, l'architecture d'un système multi-agents est composée de quelques éléments.

- Un **environnement** (représenté par le plan gris)
- Plusieurs **agents** (représenté en vert) composés chacun :
  - d'un système de **décision**
  - d'un modèle de **perception** et compréhension de son environnement (modèle cognitif)
  - d'un système de communication (**Interaction**)
- Des relations entre agents (représenté par les lignes entre les agents)

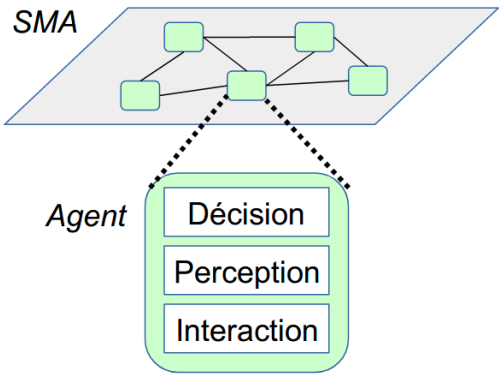


FIGURE II.3 – Architecture d'un système multi-agents

2 Les agents

Pour avoir un système multi-agents performant, il faut des agents performants. Ainsi, nous avons besoin de définir et catégoriser les agents.

2.1 Définir un agent

Commençons donc par définir un agent.

Definition 2.1. *Agent au sens général*
Un agent est un système mécanique, biologique ou logiciel qui interagit avec son environnement.

Cette définition a la particularité de rester suffisamment générale et d'être simple. Cependant, pour plus de précision, il faut se pencher sur la définition au sens de Ferber.

Definition 2.2. *Agent au sens de Ferber*
On appelle agent une entité physique ou logiciel :

a. qui est capable d'agir dans un environnement,
b. qui peut communiquer directement avec d'autres agents,
c. qui est mue par un ensemble de tendances (sous la forme d'objectifs individuels ou d'une fonction de satisfaction, voire de survie, qu'elle cherche à optimiser),
d. qui possède des ressources propres,
e. qui est capable de percevoir (mais de manière limitée) son environnement,
f. qui ne dispose que d'une représentation partielle de cet environnement (et éventuellement aucune),

g. qui possède des compétences et offre des services,
h. qui peut éventuellement se reproduire,
i. dont le comportement tend à satisfaire ses objectifs, en tenant compte des ressources et des compétences dont elle dispose, et en fonction de sa perception, de ses représentations et des communications qu'elle reçoit..

Cette seconde définition d'un agent a l'avantage de présenter une grande partie des caractéristiques dont peut être doté un agent.

2.2 Agent réactif vs agent cognitif

On distingue deux catégories d'agents, les agents réactifs et les agents cognitifs. La distinction entre ces deux catégories est parfois subjective et dépend de la définition que l'on donne. En général, on distingue agent cognitif et réactif par la représentation du monde dont dispose l'agent. Ainsi, un agent réactif ne requiert pas de représentation explicite du monde. Son comportement est en général guidé par un ensemble de règles qui lui permettent de choisir l'action à exécuter en fonction d'un stimuli (son observation du monde). Un agent cognitif, quand à lui, maintiendra une représentation explicite du monde et potentiellement du passé. Son comportement sera donc fondé sur une information plus complète et la prise de décision sera en général plus sophistiquée.

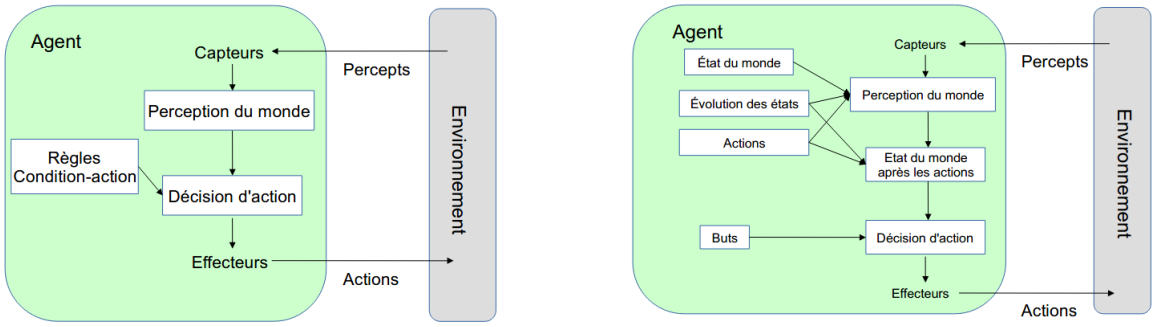


FIGURE II.4 – Architecture d'un agent réactif (gauche) et d'un agent cognitif (droite)

3 Type d'agents utilisés

Dans notre cas d'étude, qui est l'apprentissage par renforcement, il n'est pas forcément évident de catégoriser nos agents comme étant purement réactif ou cognitif. En effet, le comportement de ceux-ci n'est pas simplement basé sur un ensemble de règles prédéfinies mais sur un apprentissage d'un comportement optimal. Cependant, ce n'est pas non plus un agent cognitif car, d'après la définition, cela signifierait qu'il utilise une représentation explicite de ses connaissances et de son environnement. Or, l'agent évoluant dans un système d'apprentissage par renforcement n'utilise que son observation pour décider de son comportement et n'utilise pas d'informations supplémentaires explicites.

Ainsi, le fait que nos agents prennent des décisions uniquement en se basant sur leur observation de l'environnement fait qu'il s'agit d'agents réactifs.

Partie III

Apprentissage par renforcement multi-agents (MARL)

1 Introduction à l'apprentissage multi-agents

Dans les parties précédentes, nous avons introduit deux concepts. D'abord, les systèmes multi-agents utilisent plusieurs agents simples pour simuler un comportement général qui paraît complexe et intelligent. Dans ce cas, les agents considérés sont conçus plus ou moins à la main et agissent selon des règles prédéfinies sans chercher à améliorer leur comportement. D'un autre côté, les méthodes d'apprentissage par renforcement tentent de modifier le comportement d'un agent pour que celui-ci résolve le problème sans que l'homme ne définisse de règles à suivre.

Dans cette dernière partie théorique, nous étudions quelques approches existantes pour fusionner les avantages de ces deux domaines. C'est-à-dire que nous chercherons, à l'image des systèmes multi-agents, à mettre en oeuvre plusieurs agents relativement simples et à ce que leur comportement respectif permette un comportement global intelligent. Cependant, nous ne voudrons pas avoir à utiliser des règles prédéfinies par l'homme pour le comportement des agents. Ainsi, à l'image des méthodes de RL, nous souhaiterons que nos agents apprennent par essais successif un comportement optimal.

2 Contrôle des SMA

Il y a deux façons de contrôler le comportement des agents dans le cas d'un système de plusieurs agents. On peut faire en sorte que le contrôle se fasse de façon centralisée ou décentralisée.

2.1 Contrôle centralisé

Dans le cas d'un contrôle centralisé, une entité centrale est chargée de calculer la politique optimale jointe et de prendre les décisions pour tous les agents.

Un tel type de contrôle n'est généralement pas envisageable dans les applications réelles. En effet, il est quasiment impossible en pratique de mettre en place une entité capable de communiquer avec tous les agents sans être limitée par des contraintes physiques.

2.2 Contrôle décentralisé

Dans le cas inverse, le contrôle est dit décentralisé et ce sont les agents qui prennent les décisions. Cependant, le calcul de la politique peut, quand à lui, être réalisé de manière centralisée. Dans ce cas, une entité centrale est chargée de calculer la politique jointe au moment de l'apprentissage. Au moment de l'exécution, les politiques locales seront envoyées aux agents chargés de les exécuter sans interactions avec l'entité centrale. On peut également tout réalisé de manière décentralisé.

3 Les processus de Markov multi-agents

Nous avons vu dans la première partie que les méthodes d'apprentissage par renforcement mono-agent partent du principe que le système évolue sous la forme d'un processus de décision markovien partiellement ou totalement observable. Dans le cas multi-agents, nous avons besoin de revoir la définition des MDPs pour prendre en compte le fait d'avoir plusieurs agents avec potentiellement plusieurs objectifs. De nombreux modèles existent pour cela.

3.1 MDP multi-agents (MMDP)

L'extension la plus simple des MDPs au cas muti-agents, est le *processus de décision de Markov multi-agents*. Un MMDP est un système dans lequel chaque agent a une connaissance complète de l'état du système.

On le définit par un tuple $\langle \mathcal{S}, \boldsymbol{\mathcal{A}}, \mathcal{T}, \mathcal{R} \rangle$:

- $\mathcal{S}$: espace d'états
- $\boldsymbol{\mathcal{A}} = \mathcal{A}_1 \times ... \times \mathcal{A}_n$: ensemble d'espaces d'actions propre à chaque agent
- $\mathcal{T} : \mathcal{S} \times \boldsymbol{\mathcal{A}} \times \mathcal{S} \to [0, 1]$: fonction de transition de l'état s à l'état s' en exécutant l'action jointe $\boldsymbol{a}$

  $\to \ \mathcal{T}(s, \boldsymbol{a}, s') = \mathbb{P}(S_{t+1} = s' \,|S_t = s, A_t^{(1)} = a_1, ..., A_t^{(n)} = a_n) = p(s' \mid s, a_1, ..., a_n)$
- $\mathcal{R} : \mathcal{S} \times \boldsymbol{\mathcal{A}} \times \mathcal{S} \times \mathbb{R} \to [0, 1]$: récompense probabiliste liée au choix de l'action jointe $\boldsymbol{a}$

Dans le cas des MMDP, les agents doivent avoir accès à l'état complet de l'environnement. Ainsi, ce formalisme se prête bien au cas d'une politique centralisée. L'optimisation de la politique jointe est assez similaire aux méthodes de planification dans les MDPs vues dans la partie planification.

3.2 MDP Décentralisés (DEC-POMDP)

L'extension la plus naturelle des processus de décision mono-agent au cas de plusieurs avec une prise de décision décentralisée est sans doute le model POMDP *décentralisé*. Dans ce formalisme, on suppose la présence de plusieurs agents prenant des décisions de sorte qu'ils influencent le processus de manière simultanée. Chaque agent a sa propre observation et son propre espace d'action.

Le DEC-POMDP est décrit par un tuple $\langle \mathcal{S}, \boldsymbol{\mathcal{A}}, \mathcal{T}, \mathcal{R}, \boldsymbol{\Omega}, \mathcal{O} \rangle$ avec :

- $\mathcal{S}, \boldsymbol{\mathcal{A}}, \mathcal{T}, \mathcal{R}$: idem que pour les MMDPs
- $\boldsymbol{\Omega} = \Omega_1 \times ... \times \Omega_n$: ensemble d'espaces d'observations propre à chaque agent
- $\mathcal{O} : \Omega \times \boldsymbol{\mathcal{A}} \times \mathcal{S} \to [0, 1]$: fonction d'observation probabiliste

  $\to \ \mathcal{O}(o, \boldsymbol{a}, s') = \mathbb{P}(O_{t+1} = o \,|A_t = a, S_{t+1} = s') = p(o \mid a, s')$

En considérant l'action jointe comme étant une seule et unique action, on se retrouve dans le cas d'un POMDP, appelé POMDP sous-jacent au DEC-POMDP. Les *DEC-MDPs* sont, quant à eux, un cas particulier des DEC-POMDPs pour lequel l'état peut être retrouvé de manière sûre grâce à l'observation jointe $\boldsymbol{o} = (o_1, o_2, ..., o_n)$ de tous les agents (i.e. $p(s \mid \boldsymbol{o}) = 1$).

3.2.1 DEC-POMDP-Com

Un aspect important des systèmes multi-agents et pourtant non mis en évidence jusqu'à présent est la communication. Il est évident que l'échange d'informations est primordial pour optimiser un comportement coopératif qui nécessite la coordination des agents. Dans le cas général, on peut voir la communication comme un cas particulier d'action et donc les modèles précédents ne modélisent

pas explicitement la communication. Cependant, dans certains cas, il peut être utile d'exprimer la communication de manière explicite. C'est pour cela qu'ont été introduits les DEC-POMDP-Com ; décrit par le tuple $\langle \mathcal{S}, \boldsymbol{\mathcal{A}}, \mathcal{T}, \Sigma, \mathcal{R}, \boldsymbol{\Omega}, \mathcal{O} \rangle$:

- $\mathcal{S}, \boldsymbol{\mathcal{A}}, \mathcal{T}, \boldsymbol{\Omega}, \mathcal{O}$: idem que pour les DEC-POMDPs
- Σ : ensemble fini de messages $\sigma \in \Sigma$ ($\boldsymbol{\sigma}$ représente le message joint)
- $\mathcal{R} : \mathcal{S} \times \boldsymbol{\mathcal{A}} \times \Sigma^n \times \mathcal{S} \times \mathbb{R} \to [0, 1]$: récompense probabiliste liée au choix de l'action jointe $\boldsymbol{a}$ et du message joint $\boldsymbol{\sigma} \to \mathcal{R}(s, \boldsymbol{a}, \boldsymbol{\sigma}, s')$

Ce formalisme nécessite l'apprentissage de deux politiques distinctes : une pour les actions et une pour la communication. De plus, on peut montrer qu'il est équivalent aux DEC-POMDPs.

Type de contrôle	Centralisé		Décentralisé	
Etat global collectivement totalement observable	Oui	Non	Oui	Non
Formalisme	MDP MMDP	POMDP	DEC-MDP	DEC-POMDP
			MTDP	

FIGURE III.1 – Relations entre les différents modèles collaboratifs.

3.3 Jeux de Markov et POSGs

On remarque que dans le cas du formalisme DEC-POMDP, la récompense est commune à tous les agents. Ce formalisme peut donc convenir dans le cas d'un système collaboratif mais ne convient pas au cas d'agents ayant des objectifs concurrents (système compétitif). Les **jeux de Markov** (ou jeux stochastiques) s'appuient sur le fait que chaque agent a une récompense qui lui est propre.

Les jeux stochastiques sont donc définis par un tuple $\langle \mathcal{S}, \boldsymbol{\mathcal{A}}, \mathcal{T}, \boldsymbol{\mathcal{R}} \rangle$:

- $\mathcal{S}, \boldsymbol{\mathcal{A}}, \mathcal{T}$: idem que pour les MMDPs
- $\boldsymbol{\mathcal{R}} = \langle \mathcal{R}_1, ..., \mathcal{R}_n \rangle$: $\mathcal{R}_i$ est la fonction de récompense propre à l'agent i

Les jeux stochastiques partiellement observables (POSG) sont l'extension des jeux stochastiques au cas où les agents n'ont accès qu'à une observation de l'environnement.

3.4 Autres processus de décision multi-agents

Il existent de nombreuses extensions aux processus multi-agents présentés ci-dessus. Nous ne les détaillerons pas mais nous pouvons citer par exemples certaines extensions des DEC-POMDPs telles que les *MTDP* (qui incluent les croyances comme partie du modèle) ou les *interactive POMDPs*.

Pour construire des algorithmes, nous devons choisir un formalisme d'étude. Les deux formalismes les plus courants sont les DEC-POMDPs et les jeux stochastiques. Dans ce rapport, nous nous focaliserons sur les méthodes utilisant le formalisme des jeux stochastiques.

4 Théorie des jeux classique

Du point de vue de l'apprentissage par renforcement, les jeux stochastiques sont l'extension des MDPs au cas de plusieurs agents avec plusieurs objectifs. Nous pouvons également considérer le problème du point de vue de la théorie des jeux. En effet, selon une approche par la théorie des jeux, les jeux stochastiques sont des jeux matriciels à plusieurs étapes. Pour comprendre cela, A. Murkov représente ce concept par la figure III.2 dans le chapitre consacré aux SG du livre *PDMIA* [1]. Dans le

cas de la résolution d'un jeu stochastique nous utiliserons souvent des éléments de la théorie des jeux.

Cette section introduit les notions nécessaires de ce domaine d'étude.

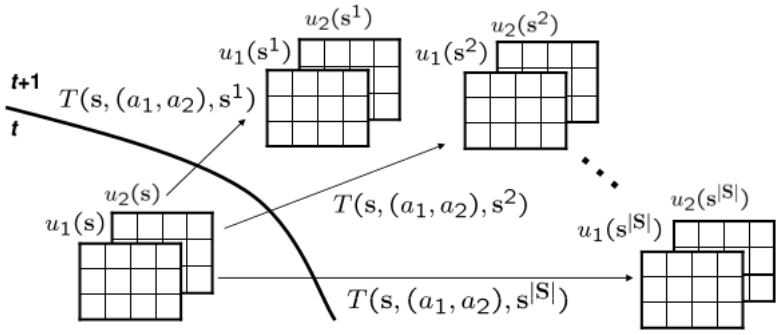


FIGURE III.2 – Jeu stochastique à deux joueurs : transitions possibles entre le tour t et le tour $t+1$ lorsque les joueurs jouent l'action conjointe (a_1, a_2). Chaque état peut être vu comme un jeu en forme normale

4.1 Les types de jeux

Quand on parle de jeux, nous cherchons en général à savoir de quel type de jeux il s'agit. En effet, il y a différentes catégories selon le caractère coopératif ou non du jeu, s'il se joue de manière simultanée ou alternée, s'il est répété ou non, etc. Nous n'expliquerons ici que les catégories qui nous intéressent le plus pour notre sujet.

Jeux coopératifs et compétitifs : Le fait que le jeu soit coopératif ou non est très important dans notre cas car nous ne serons pas forcément amenés à utiliser le même formalisme pour l'un ou l'autre. Dans le cas des *jeux coopératifs*, les agents se comportent au mieux pour l'intérêt général. Pour y arriver, ils ont besoin de moyens de communications pour élaborer une stratégie collective. Dans ce cas, le formalisme des *Dec-POMDPs* est adéquate mais celui des *jeux stochastiques* peut aussi convenir. Les *jeux non coopératifs*, quant à eux, font référence à la classes des jeux dans lesquels les agents cherchent à maximiser leur récompense individuelle. Ils n'ont en général pas besoin de communiquer avec les autres pour accomplir leur mission à bien. On parle de *jeux compétitifs* quand un jeu est non coopératif et que les agents ont des buts qui s'opposent. Le formalisme qui convient le mieux dans ce cas est celui des *jeux stochastiques*.

Jeux sous forme normale ou extensive : On dit qu'un jeu est sous forme normale si tous les joueurs effectuent leur action de manière simultanée. Dans le cas contraire, on dira que le jeu est sous forme extensive ou qu'il est alterné. Par exemple, les jeux de plateaux (échecs, dames, go) sont en général des jeux alternés tandis que les jeux de rapidité (Kem's, Dobble, Twinit) sont plus souvent simultanés. Dans notre cas, nous nous intéresserons au cas d'agents intéragissant avec l'environnement de manière simultanée (jeux en forme normale).

4.2 Trouver les meilleures stratégies

Dans le cadre d'un jeu, chaque joueur (agent) doit élaborer une stratégie (politique d'action) adéquate à la maximisation de son gain. Dans le cadre de jeux simultanés et non coopératifs, les joueurs n'ont pas connaissance des actions des autres joueurs mais peuvent, à priori, avoir une idée de leur stratégie. A cet effet, ils vont devoir choisir une action rationnelle. C'est à dire une action qui saura exploiter au mieux les faiblesses des autres joueurs pour maximiser le gain individuel. Dans le cas optimal, le joueur prévoit bien les stratégies adverses et joue sa meilleure réponse.

Definition 4.1. *Meilleure réponse*
La meilleure réponse est la stratégie qui produit le résultat immédiat le plus favorable au joueur considéré, étant donné les stratégies des autres joueurs.

Definition 4.2. *Equilibre de Nash*
Un équilibre de Nash est la situation dans laquelle chaque joueur prévoit correctement le choix des autres et joue sa meilleure réponse à cette prévision. Dans ce cas, aucun joueur n'a intérêt à changer seul de stratégie. Il faut que d'autres joueurs changent leur stratégie en même temps pour espérer récolter une meilleure récompense.

Remarque : Un équilibre de Nash n'est pas forcément unique. Dans le cas du *Dilemme du prisonier* (figure III.3), il y a un seul équilibre de Nash qui correspond à la stratégie jointe ("Ne pas avouer", "Ne pas avouer"). Dans ce cas, aucun des deux prisoniers n'a intérêt à changer de position s'il le fait seul. Pourtant, s'ils se coordonnaient, ils pourraient obtenir un gain meilleur tous les deux en choisissant d'"Avouer".

		Prisonnier 2	
		Avouer	*Ne Pas Avouer*
Prisonnier 1	*Avouer*	(5, 5)	(0, 7)
	Ne Pas Avouer	(7, 0)	(1, 1)

FIGURE III.3 – Exemple du *Dilemme du prisonier*

5 Les difficultés de l'apprentissage multi-agents

La théorie des jeux et les systèmes multi-agents apportent des outils essentiels pour étudier les intéractions entre agents. D'un autre côté, nous avons une théorie bien fondée pour le contrôle d'un agent dans son environnement. Nous aimerions adapter les différentes théories pour l'apprentissage de comportements coopératifs ou compétitifs entre agents. Un problème persiste : les méthodes d'apprentissage par renforcement mono-agent s'adaptent mal au cas multi-agents pour différentes raisons.

5.1 Non-stationnarité de l'environnement

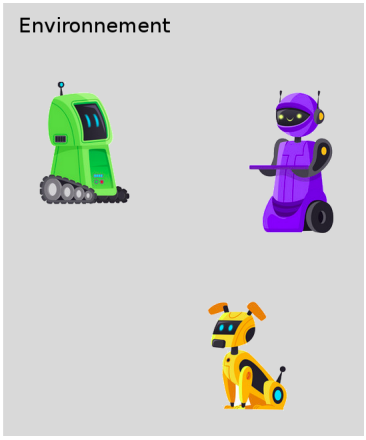
Tout d'abord, le problème central est un problème de non-stationnarité. Dans le cadre des MDPs nous considérions que le processus ne change pas au cours du temps (MDP stationnaire). C'est-à-dire que les probabilités de transitions $\mathcal{T}(s,a,s') = \mathbb{P}\left(S_{t+1} = s' \mid S_t = s, A_t = a\right)$ ne dépendent pas de la variable de temps t. Cette propriété est nécessaire pour garantir la convergence des algorithmes.

Dans le cas multi-agents, la propriété de stationnarité de l'environnement reste vérifiée si on considère l'action jointe mais du point de vue de chaque agent l'environnement semblera non-stationnaire. En effet, du point de vue d'un agent i, on a la probabilité jointe suivante :

$$p\left(s', a_1, ..., a_n, s\right) = \underbrace{p\left(s' \mid s, a_1, .., a_n\right)}_{stationnaire} \times \pi_1\left(a_1 \mid s\right) \times .. \times \underbrace{\pi_i\left(a_i \mid s\right)}_{connue} \times .. \times \pi_n\left(a_n \mid s\right)$$

La probabilité de transition $\mathcal{T}\left(s, a_1, ..., a_n, s'\right) = p(s' \mid s, \mathbf{a})$ est stationnaire mais dépend des actions des agents. De cette façon, si tous les agents apprennent, alors les politiques π_k dépendent du temps t et sont non stationnaires. Du point de vue de notre agent i qui n'a pas accès aux actions des autres, l'évolution de l'environnement dépendra de $\mathcal{T}$ et des $\pi_{k\in[1,..,n]-\{i\}}$ et sera donc non-stationnaire.

Pour illustrer cela, nous nous réfèrons à la figure ci-contre. On suppose un problème à quatre agents situés dans un même environnement, tous les agents apprenant. Comme les agents *vert*, *violet* et *jaune* sont en plein apprentissage, leur comportement évolue. Or ils font partie de l'environnement si l'on considère le point de vue de l'agent *rose*. Cette évolution des comportements des autres agents se traduit par une loi de transition de l'environnement non stationnaire.



Cette difficulté est centrale car elle enlève la garantie de convergence et rend l'apprentissage très fastidieux. De plus, elle rend inutilisable certaines techniques fondamentale de l'apprentissage par renforcement mono-agent tel que l'*expérience replay*. En effet, il est inutile de rejouer des transitions qui utilisait des anciennes politiques. Ainsi, la plupart des méthodes d'apprentissage multi-agents tentent de faire face à cette difficulté en augmentant la connaissance des agents par la connaissance des actions des autres joueurs. Cette connaissance est fournit uniquement pendant la phase d'apprentissage et peut être transmise de manière plus ou moins explicite.

6 Résoudre un jeu stochastique

De la même façon que dans le cas mono-agent, nous devons définir ce que nous entendons par comportement optimal. Dans le cas d'un jeu stochastique, on souhaite en général que chaque agent (ou joueur) sélectionne, à chaque étape, sa meilleure réponse. Ainsi, la résolution d'un jeu stochastique consiste à **trouver un équilibre de Nash**.

Dans cette dernière partie théorique, nous présentons quelques méthodes classiques de résolution des jeux stochastiques.

6.1 Minimax-Q et les dérivées

Une approche classique pour résoudre un jeu de Markov consiste à adapter l'algorithme du Q-learning pour trouver un équilibre de Nash dans la matrice en forme normale correspondante à chaque état du système. L'algorithme de base de cette approche et celui que nous avons étudié est l'algorithme **Minimax-Q**. Cette algorithme, introduit par Littman [3], est propice aux jeux à deux joueurs avec des objectifs contradictoires. Le principe consiste à tenir pour chacun des deux agents une fonction de valeur $Q_i(s,a,o)$ où a est l'action de l'agent en question et o est l'action de l'adversaire. La mise à jour de la fonction de valeur se faisant de la manière suivante :

$$Q_i(s,a,o) = Q_i(s,a,o) + \alpha[r_t + \gamma \underbrace{\max_a \min_o Q_i(s',a,o)}_{Valeur_i(s)} - Q_i(s,a,o)]$$

Grâce à cette fonction nous pourrons ainsi évaluer à chaque instant l'action la moins risquée au sens du $minimax$. C'est-à-dire qu'à l'exécution, l'agent suivra la politique déterministe suivante :

$$\pi_i(s) = Equilibre_i(G(s)) = arg\max_a \min_o Q_i(s,a,o)$$

L'algorithme complet est présenté en Annexe. On pourrait interpréter cette méthode comme un Q-learning pessimiste où $Q_i(s,a) = \min_o Q_i(s,a,o)$. C'est-à-dire un Q-learning pour lequel chaque agent prédit que son adversaire va choisir l'action parfaite pour le faire perdre.

Il existe plusieurs variantes du Minimax-Q. Les plus connues sont les algorithmes *Nash-Q* et *CE-Q*. Nash-Q ne diffère du Minimax-Q que par la façon dont il calcule l'$Equilibre$ et la $Valeur$.

6.2 Approche par descente de gradient

Les méthodes Minimax-Q et Nash-Q reposent sur une fonction de valeur pour construire une politique à priori déterministe. De la même façon qu'avec un seul agent, nous pouvons également améliorer directement la politique de chaque agent par une descente de gradient. L'algorithme de base le plus intéressant est sans doute l'algorithme de **Policy-Hill-Climbing** (*PHC*) introduit par Bownling et Veloso dans [4]. Cette méthode a l'avantage de ne pas nécessiter la connaissance de la politique de l'adversaire.

L'idée derrière cet algorithme est assez similaire aux méthodes *Acteur-Critique* vues dans la patie I. En effet, pour chaque agent i, nous allons mettre à jour la fonction de valeur q_i (le critique) à la façon du *Q-learning* :

$$Q_i\left(s, a_i\right) = Q_i\left(s, a_i\right) + \alpha[r_t + \gamma \max_{a_i'} Q_i\left(s', a_i'\right) - \ Q_i\left(s, a_i\right)]$$

Ensuite, il faut ajuster la politique π_i (l'acteur) en augmentant la probabilité de choisir l'action ayant la valeur la plus élevée de δ.

$$\pi_i(s,a) = \pi_i(s,a) + \begin{cases} \delta & si \quad a = argmax_{a'} Q(s,a') \\ \frac{-\delta}{|A_i|-1} & sinon \end{cases}$$

Cet algorithme est détaillé en Annexe. Il garantie de converger vers un comportement rationnel pour l'agent i à condition que les autres agents convergent vers des politiques stationnaires. Ainsi, si tous les agents apprennent en même temps, l'algorithme peut ne pas converger.

6.2.1 Win or Learn Fast (WoLF)

Le concept de *WoLF*, introduit dans [4], est une stratégie d'apprentissage qui stipule que si un agent gagne alors il mettra moins vite à jour son modèle. A l'inverse s'il perd, il le mettra plus rapidement à jour. Cette stratégie à permis de développer une extension intéressante de PHC qui est *WoLF-PHC*. Le but est de laisser plus de temps aux agents pour s'adapter aux changements de stratégie du joueur considéré. Pour cela, deux coefficients sont utilisés : δ_{lose} (resp. δ_{win}) pour mettre à jour les politiques des perdants (resp. des gagnants).

6.3 Multi-agents acteur-critique

Pour terminer cet état de l'art, nous présentons l'algorithme *MADDPG* de Lowe et al. [5]. Nous avons vu précédemment l'algorithme *(WoLF-)PHC* qui modélise la politique directement. Cependant, depuis 2001, de nouvelles méthodes d'ajustement direct de la politique ont été proposées pour le cas mono-agent (voir Partie I). L'algorithme **MADDPG** adapte l'une de ces méthodes récentes au cas de n agents : l'algorithme *Deep Deterministic Policy Gradient*.

L'idée consiste, au même titre que *PHC*, à évaluer pour chaque agent la fonction de valeur d'action associée et de mettre à jour directement les politiques des agents. Cependant, dans ce cas, nous considérons un espace d'état continu et nous utilisons donc des approximations de q_i et π_i. On notera θ_i les paramètres du modèle d'approximation de q_i et μ_i ceux de π_i.

De plus, pour atténuer l'effet de non-stationnarité, les auteurs considèrent des politiques d'action déterministes, continues et qui sont connues par tous les agents lors de l'apprentissage. Le but étant que les agents puissent inférer de manière exacte l'action des autres joueurs lors d'une transition

d'états. Ainsi, la fonction q_i pourra utiliser toute la connaissance nécessaire pour évaluer l'action exécutée par l'agent i dans la circonstance donnée et mettre à jour la politique de façon cohérente. Le schéma III.4 décrit ce concept d'apprentissage centralisé avec le critique q_i qui utilise l'ensemble des couples observation/action, $(o_i, a_i)\ \forall i = 1, .., n$, mais d'exécution décentralisée avec l'acteur π_i qui n'utilise que l'information locale (o_i, a_i).

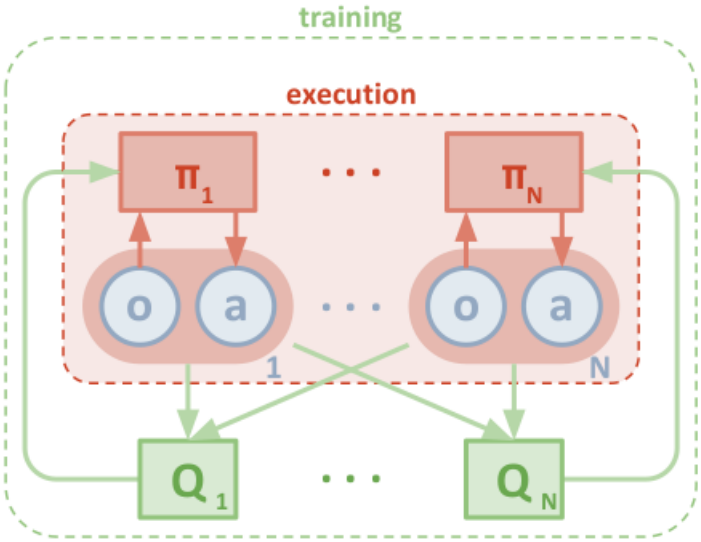


FIGURE III.4 – Apprentissage centralisé et exécution décentralisée

Algorithm 1: Multi-Agent Deep Deterministic Policy Gradient for N agents

for episode $= 1$ to M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial state $\mathbf{x}$
  for $t = 1$ to max-episode-length **do**
    for each agent i, select action $a_i = \boldsymbol{\mu}_{\theta_i}(o_i) + \mathcal{N}_t$ w.r.t. the current policy and exploration
    Execute actions $a = (a_1, \ldots, a_N)$ and observe reward r and new state $\mathbf{x}'$
    Store $(\mathbf{x}, a, r, \mathbf{x}')$ in replay buffer $\mathcal{D}$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    for agent $i = 1$ to N **do**
      Sample a random minibatch of S samples $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$ from $\mathcal{D}$
      Set $y^j = r_i^j + \gamma Q_i^{\boldsymbol{\mu}'}(\mathbf{x}'^j, a'_1, \ldots, a'_N)|_{a'_k = \boldsymbol{\mu}'_k(o_k^j)}$
      Update critic by minimizing the loss $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_N^j)\right)^2$
      Update actor using the sampled policy gradient:

$$\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i}\boldsymbol{\mu}_i(o_i^j)\nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_i, \ldots, a_N^j)\big|_{a_i = \boldsymbol{\mu}_i(o_i^j)}$$

    end for
    Update target network parameters for each agent i:

$$\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$$

  end for
end for

Partie IV

Multi-agent reinforcement learning framework

1 L'idée

La longue partie théorique précédente n'a pas seulement eu comme objectif de présenter les différentes méthodes d'apprentissage par renforcement. Elle a également mis en avant le fait que toutes ces méthodes ont des caractéristiques communes et que plusieurs méthodes découlent en général d'un cas plus global. C'est cet aspect, en plus du fait de pouvoir entraîner en ligne divers agents avec des méthodes d'apprentissage potentiellement différentes, qui a motivé le développement de la librairie *marl* en question.

L'objectif de cette librairie python est de fournir la possibilité à l'utilisateur de déclarer et d'entraîner simplement un ensemble d'agents évoluant de manière simultanée dans un environnement. Nous souhaitons également avoir la possibilité de faire intervenir à la fois des agents apprenant selon des méthodes d'apprentissage mono et multi-agent mais également des agents non entraînables. Il peut, dans ce dernier cas, s'agir d'agents utilisant des politiques déjà entraînées, de bots ou encore d'un humain.

Le code source du projet est disponible en ligne (voir Code 1).

2 Description d'un environnement adéquat

L'environnement est un point central pour l'apprentissage. La libraire *marl* repose sur le formalisme des jeux de Markov. Ainsi, dans le cas multi-agents, nous considérons que la récompense est spécifique à chaque agent et non commune à tous. De plus, on ne prévoit pas de moyen de communication explicite. Si cela est nécessaire, la communication devra devra être intégrée de façon implicite sous forme de transmission d'une partie de l'action d'un agent à l'observation d'un autre. Dans le but de travailler avec un autre formalisme, il faudra adapté, dans la limite du possible, l'environnement.

Pour être conforme à notre implémentation, l'environnement doit suivre des règles simples mais précises. En effet, l'environnement doit suivre la structure des environnements Gym [9]. C'est-à-dire que la classe doit :

- Posséder les attributs :
  - $observation\_space$ (gym.Spaces) : Il définit l'espace d'observation du ou des agents.
  - $action\_space$ (gym.Spaces) : Il définit l'espace d'action du ou des agents (sera utilisé pour tirer une action aléatoire).
- Réimplémenter les méthodes :

– $reset()$: Réinitialise l'environnement à un état initial. Cette méthode sera appelée à chaque début d'épisode d'apprentissage.
– $step(action)$: Pour une action donnée (potentiellement l'action jointe de tous les agents), met à jour l'environnement. Cette méthode retourne quatre éléments qui sont dans l'ordre : l'observation suivante, la récompense perçue, un booléen pour signifier que l'épisode est fini ou non et des potentielles informations complémentaires.
– $render()$: Affiche l'environnement.

Un grand nombre d'environnements Gym mono-agent existent déjà mais il en existent beaucoup moins qui font intervenir plusieurs agents.

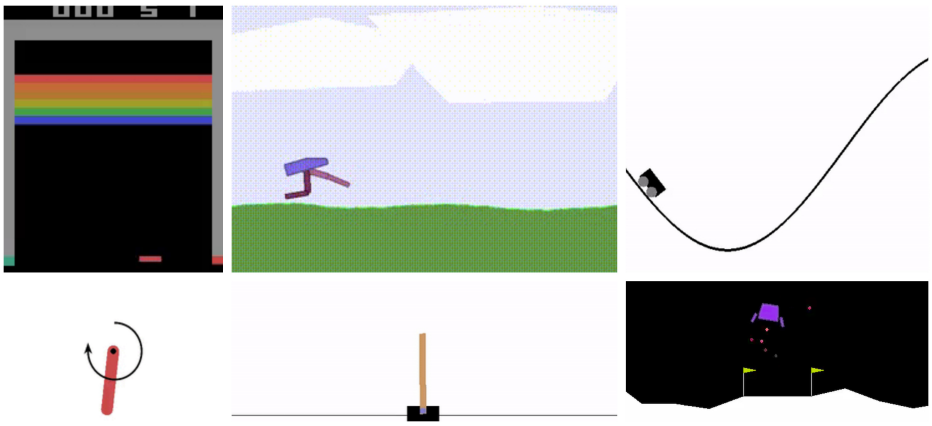


FIGURE IV.1 – Exemples d'environnements classiques mono-agent

3 Détails d'implémentation

Les algorithmes finalement implémentés et testés sont ceux présentés dans le tableau suivant.

Single-agent algorithms

Q-learning	DQN	Actor-Critic	DDPG	TD3
✓	✓	✓	✓	✗

Multi-agent algorithms

minimaxQ	PHC	JAL	MAAC	MADDPG
✓	✓	✗	✓	✓

3.1 Architecture logiciel

Un début d'architecture logiciel a été réalisé pour expliquer le lien entre les différentes parties de l'implémentation. Les diagrammes réalisés à cet effet sont en Annexe.

3.2 Documentation

Pour aider l'utilisateur à bien comprendre le fonctionnement de l'API et comment l'utiliser, nous avons décidé de générer une documentation automatique via l'utilitaire Sphinx [10]. La documentation générée est disponible en ligne à l'adresse : `https://blavad.github.io/marl`.

4 Exemples d'utilisation

Pour montrer que la librairie est relativement simple d'utilisation. Nous présentons ici deux exemples : un dans le cas d'un agent seul et un dans le cas d'un système composé de deux agents.

4.1 Exemple 1 : Deep Q-learning pour un agent seul

Dans ce premier exemple, nous montrons le code correspondant à l'exemple d'application de DQN présenté à la première partie. Nous voyons que nous pouvons déclarer et customiser de manière relativement simple l'agent et lancer un apprentissage et un test sur un environnement en une seule ligne.

```
import marl
from marl.agent import DQNAgent
from marl.model.nn import MlpNet

import gym
# Choose an environment
env = gym.make("LunarLander-v2")

obs_s = env.observation_space
act_s = env.action_space

#Choose a network model
mlp_model = MlpNet(8, 4, hidden_size=[64, 32])

# Declare your agent
dqn_agent = DQNAgent(   mlp_model, obs_s, act_s, experience="ReplayMemory-5000",
                        exploration="EpsGreedy", lr=0.001, name="MonModuleLunaire")

# Train the agent for 100 000 timesteps
dqn_agent.learn(env, nb_timesteps=100000)

# Test the agent for 10 episodes
dqn_agent.test(env, nb_episodes=10)
```

4.2 Exemple 2 : Minimax-Q learning pour deux agents

Dans ce deuxième exemple, nous montrons un code permettant de déclarer et entraîner deux agents utilisant l'algorithme Minimax-Q de manière simultanée sur l'environnement *soccer* (voir code 2).

```
import marl
from marl.agent import MinimaxQAgent
from marl.exploration import EpsGreedy

from soccer import DiscreteSoccerEnv
# Environment available here "https://github.com/blavad/soccer"
env = DiscreteSoccerEnv(nb_pl_team1=1, nb_pl_team2=1)

obs_s = env.observation_space
act_s = env.action_space

# Custom exploration process
```

```
expl = EpsGreedy(eps_deb=1.,eps_fin=.3)

# Create two minimax-Q agents
q_agent1 = MinimaxQAgent(   obs_s, act_s, act_s, exploration=expl,
                            gamma=0.9, lr=0.001, name="SoccerJ1")

q_agent2 = MinimaxQAgent(   obs_s, act_s, act_s, exploration=expl,
                            gamma=0.9, lr=0.001, name="SoccerJ2")

# Create the trainable multi-agent system
mas = MARL(agents_list=[q_agent1, q_agent2])

# Assign MAS to each agent
q_agent1.set_mas(mas)
q_agent2.set_mas(mas)

# Train the agent for 100 000 timesteps
mas.learn(env, nb_timesteps=100000)

# Test the agents for 10 episodes
mas.test(env, nb_episodes=10, time_laps=0.5)
```

Partie V

Expérimentations

Les expérimentations ont été réalisées dans le but de tester les différentes parties de la framework *marl*. Nous avons testé certains algorithmes mono-agent sur des environnements des modules *Box2D* et *Classic Control* de la plateforme *Gym*. Dans le cas multi-agents, la plateforme de base ne fournissant pas d'environnements appropriés, nous avons dû implémenter notre propre environnement multi-agents.

1 L'environnement *Soccer*

L'environnement *Soccer* a été implémenté lors du projet pour fournir la possibilité de tester les méthodes d'apprentissage multi-agents. Il existe différentes configurations qui permettent de tester des cas d'usage différents.

1.1 Discrete Soccer

Un premier type d'environnement est un environnement avec espace d'état fini. Dans ce cas, le terrain de jeu est représenté par une grille de dimensions $h \times w$. A un instant t, chaque joueur occupe une position unique de la grille.

Description détaillée :

→ Grille $h \times w$ (ici 4×5)

→ Action : $a \in \{none, front, back, left, right\}$

→ Deux types d'observations

- Etat exact : $s \in [0, ..., n \times (hw)^n]$
- Map : $s \in \{0,1\}^h \times \{0,1\}^w \times \{0,1\}^3$

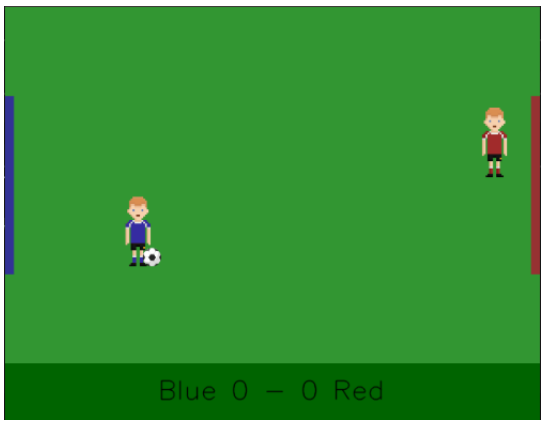


FIGURE V.1 – Exemple de l'environnement *Discrete Soccer* dans le cas de 2 joueurs

1.2 Continuous Soccer

Pour combler le manque de réalisme qu'offre le jeu de football sous la forme de grille, nous avons étendu l'environnement *DiscreteSoccer* au cas continue.

Description détaillée :

→ Action : $a \in \{none, front, back, left, right\}$

→ Observation : $o \in [-1, 1]^{5+2\times(n-1)}$

L'observation dans ce cas est constitué $5 + 2(n-1)$ valeurs entre -1 et 1 avec n représentant le nombre de joueurs sur le terrain. Il y a donc $2(n-1)$ valeurs pour la distance relative avec les différents joueurs, deux valeurs pour la distance relative avec le ballon, deux pour les distances relative avec le but adverse et une dernière pour la distance en x avec notre but.

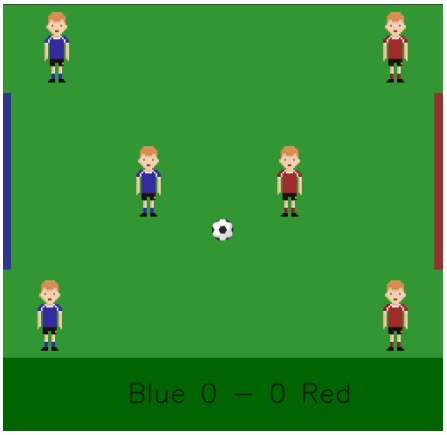


FIGURE V.2 – Exemple de l'environnement *Continuous Soccer* dans le cas de 6 joueurs

2 Comparaisons de méthodes

Pour diminuer le temps passer à entraîner les différents modèles, l'environnement *Discrete Soccer* avec un terrain de dimension 2×2 et deux joueurs a été choisi pour les tests. La figure V.3 représente cet environnement.

Nous avons ainsi pu tester et comparer les algorithmes minimax-Q, Q-learning et PHC. Du fait que le nombre d'états de notre environnement augmente de manière quadratique avec la dimension du terrain et de manière exponentielle avec le nombre de joueur, nous avons vite constaté qu'il était pas envisageable d'utiliser un terrain plus grand pour l'algorithme du minimax-Q. Cette restriction n'est pas forcément vrai pour les algorithmes PHC et Q-learning classiques. En effet, la méthode PHC a montré être également converger vers une politique d'action intéressante dans le cas de terrains de dimensions 3×3 et 4×4.

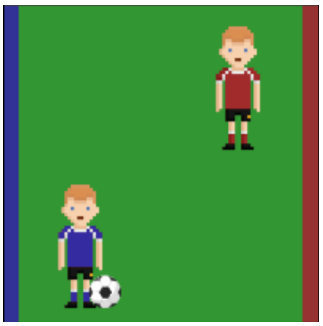


FIGURE V.3 – Environnement de test

2.1 Résultats

Pour la réalisation des tests, les agents ont été entraînés pendant 100000 steps avec un taux d'apprentissage de 0.01. Tous des agents entraînés, faisaient face à un adversaire utilisant la même méthode d'apprentissage que lui. Ce n'est que lors de la phase de test que nous avons confronté des agents entraînés avec différentes méthodes.

Chaque test consiste en 1000 matchs. Un match est considéré comme nul si aucun agent n'a marqué après 20 steps. Nous ne tenons pas compte du temps mis avant de marquer un but. Les résultats transcrits pourcentage sont présentés dans la figure ci-dessous.

Minimax-Q vs Random
- Minimax-Q : 93.6 %
- Random : 9.4 %
- Match nul : $\leq 2\%$

Minimax-Q vs Q-learning
- Minimax-Q : 100.0 %
- Q-learning : 0.0 %
- Match nul : $\geq 70\%$

Q-learning vs Random
- Q-learning : 95.9 %
- Random : 4.1 %
- Match nul : $\leq 1\%$

MinimaxQ vs PHC
- Minimax-Q : 58.2 %
- PHC : 41.8 %
- Match nul : 1 %

PHC vs Random
- PHC : 82.3 %
- Random : 17.7 %
- Match nul : $\leq 10\%$

FIGURE V.4 – Comparaison des différentes méthodes d'apprentissage

2.2 Interprétation

Les tests précédents prouvent que chaque algorithme d'apprentissage a permis de converger vers des politiques de jeux intéressante. En effet, dans tous les cas, les agents entraînés battent de manière quasi certaine l'agent jouant selon une politique aléatoire. Il est intéressant également de remarquer que la confrontation des deux méthodes basées sur l'estimation de la valeur Q aboutit à un très grand nombre de match nul. Cela vient du fait que chaque agent a une politique d'action déterministe qui consiste à ne pas prendre de risque de perdre la balle qui a ne pas marquer. On comprend donc qu'une politique d'action déterministe, même si elle est bonne, engendre un jeu prédéfini et ennuyeux. L'algorithme PHC, fait intervenir une partie d'aléatoire dans son jeu qui engendre de l'intérêt dans le match.

Conclusion

Le projet de fin d'étude a pour objectif de se familiariser avec un domaine d'étude récent qui nous intéresse. Dans certains cas, il peut permettre d'avoir une première approche avec sujet que nous souhaiterions étudier dans notre carrière future.

Au cours de six mois nous avons donc pu explorer la thématique de l'apprentissage par renforcement multi-agents. Cette thématique n'est pas nouvelle mais a récemment refait surface suite aux progrès réalisés en terme d'apprentissage mono-agent. L'objectif initial était de réaliser une étude bibliographique pour bien comprendre les différents enjeux, les problématiques et les solutions envisagées pour ce sujet. Ainsi en fin de projet nous pourrions réaliser quelques tests simples selon une méthode préalablement choisie. Finalement, nous avons suivi notre objectif en réalisant cet état de l'art et en tentant d'introduire différentes approches de résolution du sujet. Une partie implémentation a également été réalisée. Cette partie n'a pas simplement eu comme but de réimplémenter les méthodes existantes mais elle nous a permis de réfléchir plus longuement sur le lien entre la théorie et la pratique. L'objectif de l'implémentation était de réaliser une framework simple d'utilisation mais très extensible. Ainsi, de nombreuses questions ont été mentionnés tel que le fait de savoir ce que nous pouvions qualifier d'agents ou non.

Finalement, ce projet a permis de mettre en pratique de nombreuses compétences acquises tout au long de notre formation mais également d'en développer des nouvelles autant du côté théorique qu'appliqué. De plus, la possibilité de travailler sur ce sujet en collaboration avec deux enseignants-chercheurs qui m'ont suivis et permis de découvrir le monde de la recherche a été un atout considérable. Merci à eux de m'avoir encadré tout au long de ce projet.

Codes

- **Code 1** - Multi-Agent Reinforcement Learning API
  https://github.com/blavad/marl
- **Code 2** - Soccer environment
  https://github.com/blavad/soccer
- **Documentation** - Documentation de l'API *marl*
  https://blavad.github.io/marl

Bibliographie

[1] *Processus Décisionnels de Markoven Intelligence Artificielle* [Communauté PDMIA, 2008]
http://researchers.lille.inria.fr/~munos/papers/files/bouquinPDMIA.pdf

[2] *Reinforcement Learning : An Introduction* [Sutton & Barto, 2018]
http://www.incompleteideas.net/book/the-book-2nd.html

[3] *Markov games as a framework for multi-agent reinforcement learning* [Littman, 1994]
https://openreview.net/forum?id=SkNAa9-dZB

[4] *RationalandConvergentLearninginStochasticGame* [Bowling & Veloso, 2001]
http://www.cs.cmu.edu/~mmv/papers/01ijcai-mike.pdf

[5] *Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments* [Lowe et al., 2017]
https://arxiv.org/abs/1706.02275

[6] *Playing Atari with Deep Reinforcement Learning* [Mnih et al., 2013]
https://arxiv.org/abs/1312.5602

[7] *Cours Systèmes Multi-agents* [L. Vercouter]
http://pagesperso.litislab.fr/lvercouter/teaching/

[8] *Multi-Agent Particle Environment* [Lowe et al., 2017]
https://github.com/openai/multiagent-particle-envs

[9] *OpenAI Gym* [OpenAI]
https://github.com/openai/gym

[10] *Sphinx - Python Documentation Generator* [Georg Brandl]
https://www.sphinx-doc.org/en/master

Annexes

Algorithmes Partie I

Algorithm 4 Deep Q Learning with Experience Replay

procedure DQN($n\_episodes, T, C, \epsilon, \gamma$)
  Initialize replay memory D to capacity N
  Initialize action-value function Q_θ with random weights θ
  Initialize target action-value function $\hat{Q}_{\theta^-}$ with weights $\theta^- = \theta$
  for $episode = 0,\ n\_episodes$ **do**
    Initialize state s_1
    for $t = 0,\ T$ **do**
      With probability ϵ select a random action a_t,
      otherwise select $a_t = arg\max_a(Q_\theta(s_t, a))$
      Execute action a_t and observe reward r_t and state s_{t+1}
      Store transition (s_t, a_t, r_t, s_{t+1}) in D
      Set

$$y_j = \begin{cases} r_j \ \ if\ episode\ terminates\ at\ step\ j+1 \\ r_j + \gamma \max_{a'} \hat{Q}_\theta(s_{t+1}, a') \ \ otherwise \end{cases}$$

      Perform a gradient descent step on $(y_j - Q_\theta(s_j, a_j))^2$
      Every C steps reset $\hat{Q} = Q$
      Set $s_{t+1} = s_t$

Algorithmes Partie III

Algorithme 4.2 : Algorithme de la base pour les algorithmes Minimax-Q et Nash-Q pour un joueur i

pour tous les s, $\mathbf{a}$ *et* $j \in Ag$ **faire** Initialiser $Q_j(s, \mathbf{a})$ par des valeurs arbitraires.
Mettre dans s l'état courant.
Construire le jeu $G(s)$ à partir des valeurs Q_j dans s, $\forall j \in Ag$.
Choisir une politique $\pi_i(s) = Equilibre_i(G(s))$.
répéter
  Jouer la politique $\pi_i(s)$ avec une certaine exploration.
  Observer l'action conjointe et la mettre dans $\mathbf{a}$.
  Observer les récompenses de chacun et les mettre dans $\mathbf{r}$.
  Observer le nouvel état et le mettre dans s'.
  Construire le jeu $G(s')$.
  Choisir une politique $\pi_i(s') = Equilibre_i(G(s'))$.
  pour chaque *joueur* j **faire** Mettre à jour la valeur $Q_j(s, \mathbf{a})$ en utilisant l'équation appropriée
  $s \leftarrow s'$, $t \leftarrow t + 1$.
jusqu'à $t > T$
pour $s \in S$ **faire**
  Construire la matrice $G(s)$ comme précédemment.
  $\pi_i(s) = Equilibre_i(G(s))$.
retourner $\pi_i(s)\ \forall s$.

Algorithme 4.5 : Algorithme PHC pour les SG pour un joueur i, adapté de [BOW 02a]

Initialiser $\alpha \in (0, 1]$, $\delta \in (0, 1]$, $\gamma \in (0, 1)$.
pour tous les $s \in S$ *et* $a_i \in A_i$ **faire** Initialiser $q_i(s, a_i) \leftarrow 0$.
pour tous les $a_i \in A_i$ **faire** Initialiser la politique courante $\pi_i^{a_i}(s) \leftarrow \frac{1}{|A_i|}$.
L'état courant $s \leftarrow s^0$.
répéter
  Choisir une action a_i selon la stratégie $\pi_i(s)$.
  Jouer a_i avec une certaine exploration.
  Observer le nouvel état s' et la récompense obtenue R_i.
  Mettre à jour $q_i(s, a_i)$ en utilisant la règle suivante :

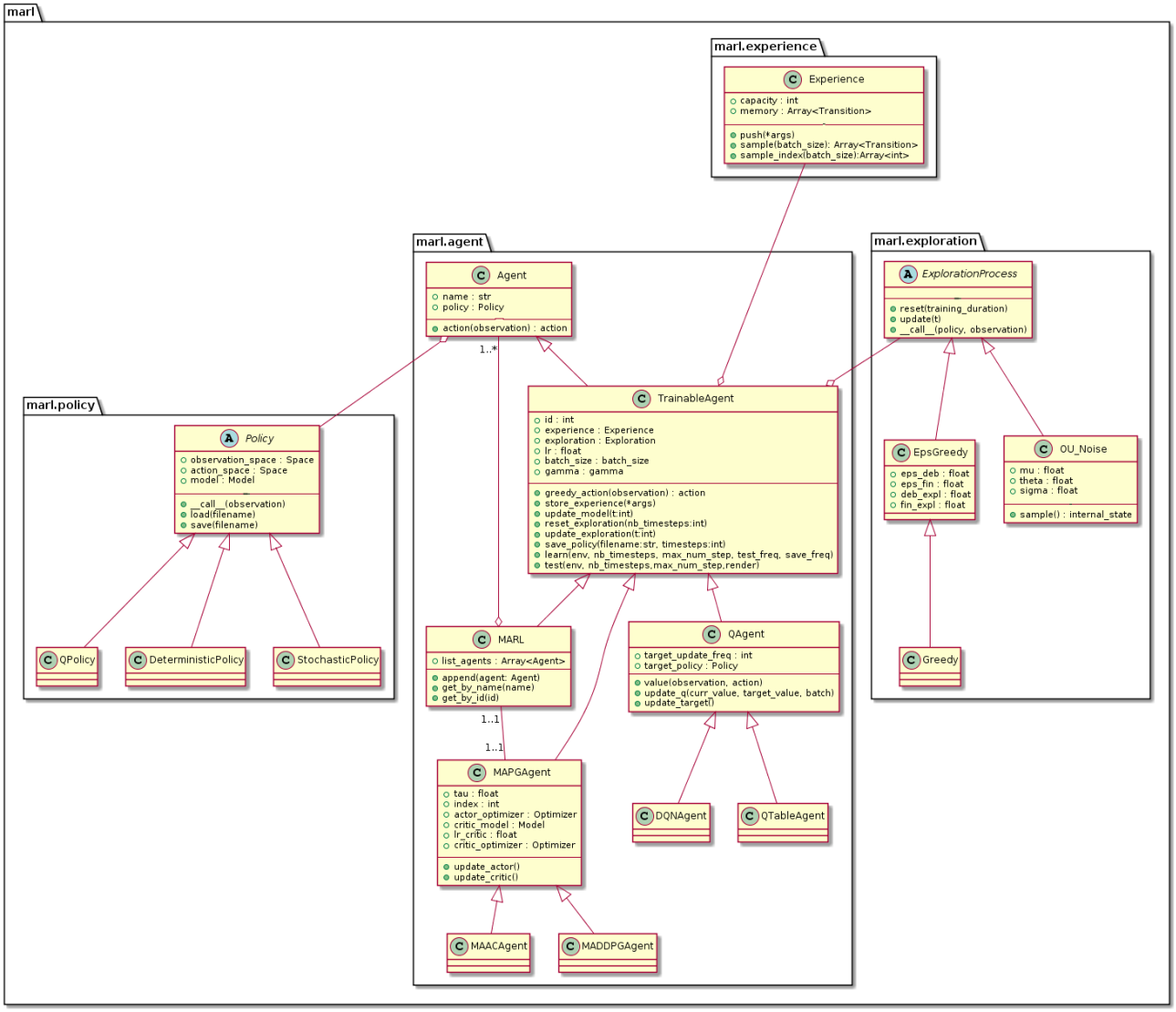
$$q_i(s, a_i) \leftarrow (1 - \alpha) q_i(s, a_i) + \alpha \left(R_i + \gamma \max_{a_i'} q_i(s', a_i') \right).$$

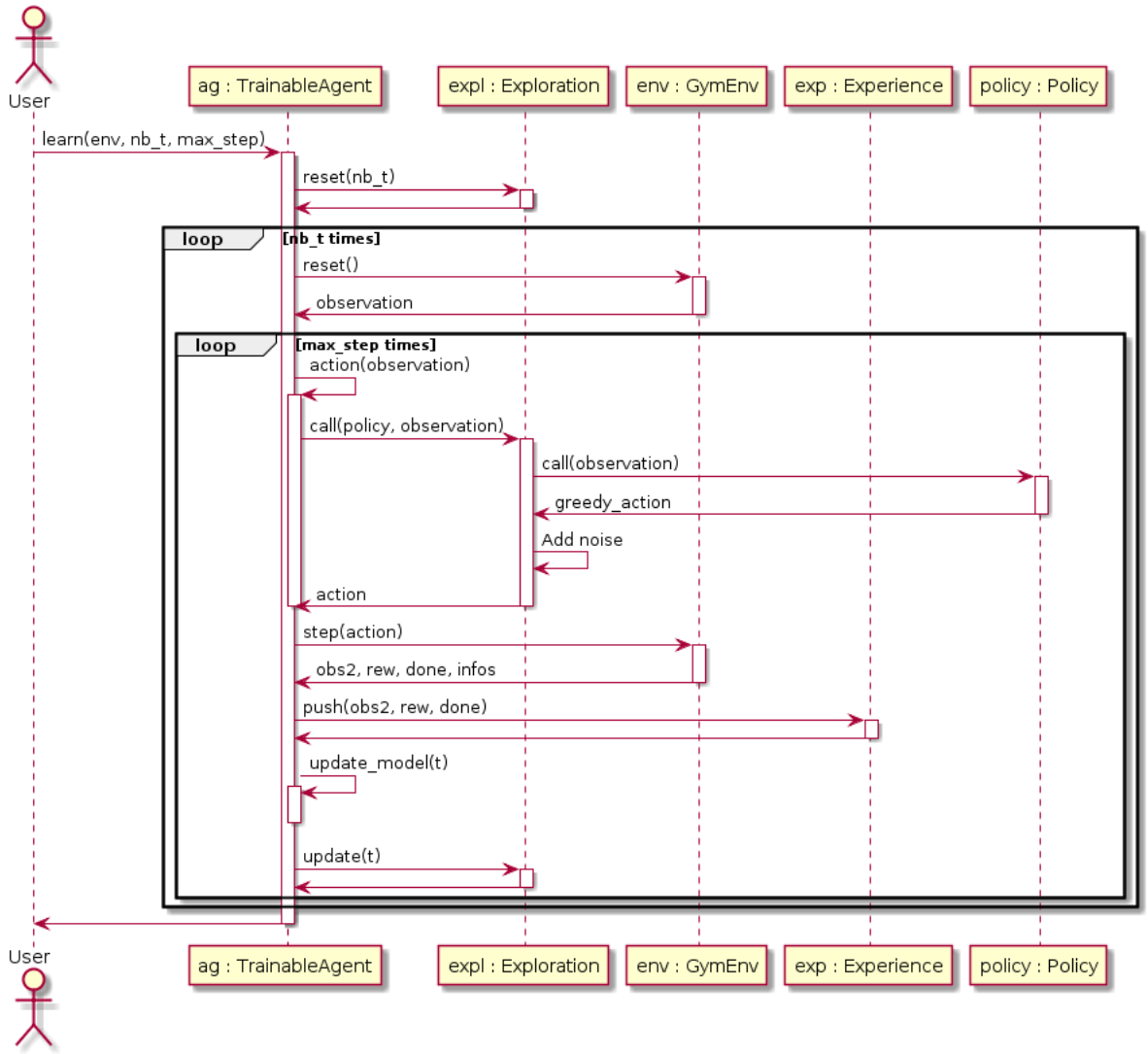
  Mettre à jour la stratégie courante, $\pi_i(s)$, en utilisant la règle suivante :

$$\begin{aligned} \pi_i^{a_i}(s) &\leftarrow \pi_i^{a_i}(s) + \Delta_{sa_i}, \text{ où} \\ \Delta_{sa_i} &= \begin{cases} -\delta_{sa_i} & \text{si } a_i \neq \operatorname{argmax}_{a_i'} q_i(s, a_i') \\ \sum_{a_i' \neq a_i} \delta_{sa_i'} & \text{sinon ;} \end{cases} \\ \text{avec} \quad \delta_{sa_i} &= \min \left(\pi_i^{a_i}(s), \frac{\delta}{|A_i| - 1} \right), \end{aligned}$$

  en se limitant à la distribution de probabilité légale.
  Mettre à jour l'état courant $s \leftarrow s'$.
  $n \leftarrow n + 1$.
jusqu'à $n > N$
retourner $\pi_i(s)\ \forall s$.

Architecture logiciel




User
ag : TrainableAgent
expl : Exploration
env : GymEnv
exp : Experience
policy : Policy
learn(env, nb_t, max_step)
reset(nb_t)
loop
[nb_t times]
reset()
observation
loop
[max_step times]
action(observation)
call(policy, observation)
call(observation)
greedy_action
Add noise
action
step(action)
obs2, rew, done, infos
push(obs2, rew, done)
update_model(t)
update(t)
User
ag : TrainableAgent
expl : Exploration
env : GymEnv
exp : Experience
policy : Policy