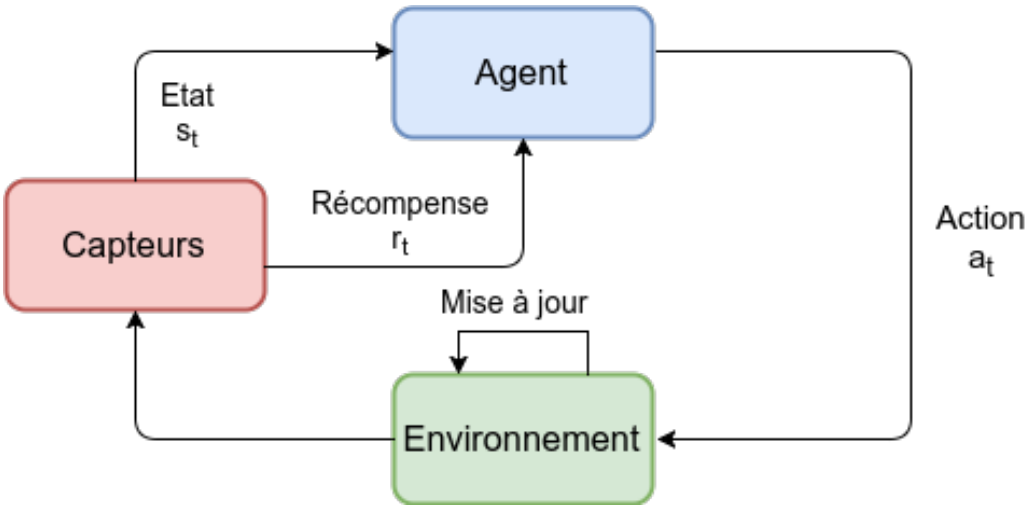
Apprentissage par renforcement

Algorithme du Q-learning

Octobre 2017 - Janvier 2018



Etudiants :
Cyril Antoun
Victor Le Maistre
Carlos Miranda

Enseignant encadrant :
Laurent Vercouter

Table des matières

1 Introduction

1.1 Cadre du projet

1.1.1 Cadre général

Ce projet semestriel est réalisé dans le cadre de la formation de Génie Mathématiques de l'Institut National des Sciences Appliquées de Rouen. Il s'agit ici de se documenter sur le sujet ainsi que d'appréhender et d'appliquer des connaissances sur l'intelligence artificielle. Plus précisement, ce projet a pour objectif d'étudier l'algorithme du *Q-learning* (algorithme d'apprentissage par renforcement) et d'en produire une implémentation dans le langage de programmation Java pour un jeu choisi librement.

L'équipe a décidé d'entreprende l'implémentation d'une intelligence artificielle optimisant le score du jeu Super Mario Bros. En ce qui concerne le déroulement du projet, voir l'annexe 1 (diagramme de Gantt). En plus, étant un sujet nouveau pour les membres du groupe, nous avons décidé que tous devaient participer aux différentes étapes et tâches du projet. Ainsi, on n'a pas défini une répartition des tâches explicitement.

1.2 Définitions et formalismes

Définition (Agent)**.** Un agent intelligent, ou agent rationnel, est une entité autonome pouvant intéragir avec son environnement par le biais de capteurs et d'effectueurs, dans le but de réaliser un objectif. Dans le cadre de ce projet, il s'agit d'un programme pouvant faire des choix sans besoin d'intéraction humaine.

Définition (Etat)**.** On appelle état de l'agent une description multivariable de la situation de l'agent dans l'environnement à un temps donné. Dans le cadre de ce projet, on fait une modélisation en temps discret, donc on a un ensemble discret d'états s_t avec $t \in [0; T]$, T un temps maximal.
Notation : On notera cet ensemble S et l'état de l'agent à un temps donnée s_t.

Définition (Action)**.** Une action permet à l'agent de passer d'un état à un autre. Il est important de remarquer que, en accord avec cette définition, le choix de l'agent de ne rien faire compte comme une action.
Notation : On notera l'ensemble des actions A et une action à un temps donné a_t.

Définition (Récompense)**.** Une récompense est une valeur (scalaire ou vectorielle) servant à l'agent pour évaluer le rendement d'une action ou d'une suite d'actions. Il faut bien noter qu'une récompense peut être négative, dans le cas où on veuille pénaliser le choix d'action de l'agent. Le choix des magnitudes

des récompenses est un des facteurs clés à la bonne modélisation du problème.
Notation : On notera l'ensemble des récompenses R et une récompense à un temps donnée r_t.

Définition (Politique)**.** La politique, ou police, d'un agent est une fonction modélisant le choix de l'action. Cette fonction donne la probabilité d'effectuer une action lorsque l'agent est dans un état donné.
Notation : On note une politique π telle que :

$$\pi : \begin{array}{l} S \times A \to [0,1] \\ a|s \mapsto \mathbb{P}\left(a_t = a | s_t = s\right) \end{array}$$

où $a|s$ correspond à l'événement *choisir d'effectuer l'action a sachant qu'on est à l'état s.*

Définition (Processus de Décision Markovien)**.** Un Processus de Décision Markovien (PDM) est un modèle mathématique des systèmes de prise de décision séquentielle. La caractéristique principale de ces processus est qu'ils sont munis de la propriété de Markov, c'est pourquoi ils sont aussi appelés processus *sans mémoire*. Dans le cadre de ce projet, on utilisera la *propriété de Markov, dite* faible élémentaire, qui s'exprime comme suit :

$$\begin{array}{l} \text{Pour tout } n \geq 0, \text{ pour toute suite d'états } (i_0, ..., i_{n-1}, i, j) \in E^{n+2}, \\ \mathbb{P}\left(X_{n+1} = j | X_0 = i_0, ..., X_{n-1} = i_{n-1}, X_n = i\right) = \mathbb{P}\left(X_{n+1} = j | X_n = i\right) \end{array}$$

Cette propriété servira dans le cadre de la modélisation et de l'algorithme du *Q-learning.*

Définition (Apprentissage automatique en ligne et hors ligne)**.** L'apprentissage automatique peut se faire en ligne ou hors ligne :

- *en ligne* l'apprentissage est fait en temps réel, dès que les informations sont disponibles. Il existe un flux de données que l'agent doit interpréter et celui-ci doit réagir en conséquence.
- *hors ligne* l'apprentissage est fait sur une banque de données statique. On dispose de toutes les données depuis le début, ce qui permet des approches différentes (notamment statistiques).

1.3 Histoire et principe

Histoire de l'apprentissage par renforcement

L'apprentissage par renforcement est un domaine du *Machine Learning* inspiré par la psychologie du comportement. Il est question de placer un agent autonome dans un environnement avec lequel l'agent pourra interagir pour recevoir des récompenses, le but étant de maximiser la récompense totale au cours du temps.

La version discrète et aléatoire de ce problème, les Processus de Décision Markoviens, furent introduits par Richard Bellman en 1957, dans le cadre du contrôle optimal des systèmes dynamiques. Les premiers algorithmes ne sont apparus qu'à la fin des années 80s. Le *Temporal difference learning* (TD-learning, 1988) de Richard Sutton et le *Q-learning* (1992) de Chris Watkins sont les deux premiers algorithmes publiés d'apprentissage par renforcement.

Depuis ses débuts, l'apprentissage par renforcement est en lien direct avec la psychologie du comportement et intéresse alors les neurobiologistes et les psychologues. En effet, plusieurs expériences et découvertes ont mis en évidence une importante similitude entre la façon dont les vertébrés apprennent à choisir leurs actions et les algorithmes d'apprentissage par renforcement.

Aujourd'hui, avec l'intérêt croissant que suscitent l'intelligence artificielle et ses applications, les méthodes d'apprentissage par renforcement se développe très rapidement. De plus, le parallèle avec le fonctionnement du cerveau permet d'explorer nouvelle formes d'apprentissage par renforcement, ou tout simplement d'améliorer les méthodes actuelles.

Principe de la méthode

L'apprentissage par renforcement consiste à munir un agent autonome d'une police et de capteurs et de le placer dans un environnement avec lequel celui-ci peut intéragir. L'agent effectue une action ayant une influence sur l'environnement. Les capteurs reçoivent de l'information sur l'environnement (une représentation de l'environnement ou des données à exploiter) et produisent un état et une récompense pour l'agent.

Le but principal dans un problème d'apprentissage par renforcement est de choisir des actions de façon à maximiser une récompense quantitative au cours de l'expérience. Pour cela, on construit une politique optimale au travers d'expériences itérées.

Modélisation

Le modèle d'apprentissage par renforcement consiste en trois ensembles :

- S, l'ensemble d'états de l'agents dans l'environnement
- A, l'ensemble d'actions que l'agent peut effectuer
- R, l'ensemble des récompenses (valeurs scalaires positives ou négatives) que l'agent peut obtenir suite au choix d'une action dans un état

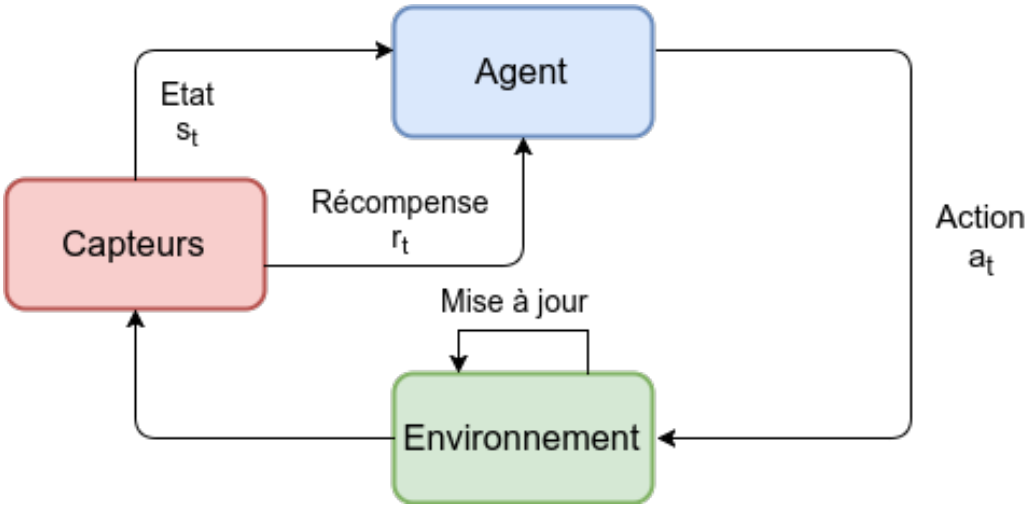


FIGURE 1 – Apprentissage par renforcement

L'expérience est subdivisée avec un pas de temps t. La détermination de ce pas de temps est essentiel au bon fonctionnement de l'algorithme. En effet, si on choisit un pas de temps trop grand, l'agent ne percevra pas suffisament d'information, tandis que si le pas de temps est trop petit, l'agent pourrait capter plusieurs fois la même information, ou agir plusieurs fois avant que l'environnement ne tienne compte de l'action, les récompenses perçues étant alors fausses.

A chaque pas de temps t de l'algorithme, l'agent perçoit sont état $s_t \in S$ et l'ensemble des actions réalisables depuis cet état $A(s_t) \subset A$. Ensuite, l'agent choisit une action $a \in A(s_t)$ grâce à sa stratégie de décision. L'environnement réagit à cette action et les capteurs de l'agent produisent un nouvel état s_{t+1} et une récompense r_{t+1}. La valeur de cette récompense r_{t+1} caractérise l'utilité de l'action a_t choisie à l'état s_t et permet l'amélioration de la police de l'agent.

L'apprentissage par renforcement permet à l'agent de construire une politique π permettant de maximiser le total des récompenses :

- Dans le cas de PDM avec état terminal, il s'agit de maximiser $r = \sum_t r_t$.
- Dans le cas de PDM sans état terminal, il s'agit de maximiser $r = \sum_t \gamma^t r_t$, avec γ un facteur d'actualisation (*discount factor*, compris entre 0 et 1) qui permet de pondérer l'importance des récompenses futures.

2 Description du cadre de travail

2.1 Introduction au jeu

Super Mario Bros. est un jeu vidéo de plates-formes développé par *Nintendo* pour son *Nintendo Entertainment System* (NES). Successeur du jeu *Mario Bros.* publié en 1983, *Super Mario Bros.* fut publié au Japon et en Amérique du Nord en 1985. Dans le jeu, le joueur contrôle Mario et son frère, Luigi, à travers le *Mushroom Kingdom* (parfois traduit Royaume Champignon) dans le but de porter secours aux habitants du royaume mais plus particulièrement à la princesse *Toadstool* (postérieurement appelée *Peach*) à l'égard de l'antagoniste *Bowser*.

Composition des niveaux

Le joueur devra guider Mario à travers plusieurs niveaux. Chaque niveau est composé généralement par des plates-formes, des blocs, des ennemis et des tuyaux :

- Les plates-formes permettent d'avancer, d'atteindre des endroits élevés, et très souvent constituent le seul point d'appui de Mario.
- Les blocs sont des structures destructibles qui peuvent contenir des récompenses telles que des pièces, des champignons et des fleurs.
- Les ennemis viennent en plusieurs formes. Au début, on rencontre typiquement des *Goombas* et des *Koopas*, mais après le joueur sera attaqué par toute une panoplie d'ennemis, parfois indestructibles.
- Les tuyaux permettent à Mario de se déplacer et d'atteindre des parties cachées du nvieau mais sont aussi utilisés par des ennemis particuliers, les *Piantas*.

Mécanique du jeu

La mécanique du jeu est assez simple. Les niveaux sont à défilement parallaxe et à temps limité. Le joueur a le choix parmi 5 actions (pouvant être composées) :

- Déplacement : Gauche, droite
- Saut
- Courir
- S'accroupir

Le but de chaque niveau est de maximiser la quantité de points obtenus, tout en traversant le niveau le plus rapidement possible car des points bonus sont attribués en fonction du temps restant à la fin du niveau.

Il existe des niveaux spéciaux (e.g. sous-marins, parties sur un nuage, etc) dans lesquels la mécanique de mouvement change partielle ou totalement.

Finalement, après un groupe de niveaux, Mario devra affronter un ennemi unique à détruire (*boss*).

Dans notre projet, on a utilisé un outil simulant l'environnement du jeu. Ainsi, si bien que dans le jeu original il y ait des niveaux sous-marins, des boss, et bien d'autres éléments particuliers, nous ne les tiendront pas en considération pour le déroulement du projet vu que ceux-ci sont absents de l'environnement simulé de jeu.

2.2 Outil MarioAI

Entre 2010 et 2012 a eu lieu une compétition de programmation d'Intelligence Artificielle appelée *Mario AI Champhionship*[1] (aujourd'hui *Platformer AI Championship*[2]). Le but de la compétition était de développer le meilleur contrôleur (agent intelligent) pour un hommage codé en Java de *Super Mario Bros.* La compétition était divisée en trois étapes :

- *Gameplay* : une version simplifiée du jeu original
- *Learning* : version où l'agent pourra s'entraîner sur un plusieurs niveaux sans se soucier des vies et du temps
- *Level Generation* : génération automatique de niveaux avec certains critères

Pour la réalisation de cette compétition, les organisateurs ont mis à disposition des participants un outil de simulation de l'environnement du jeu *Super Mario Bros.* Ainsi, les participants n'auraient qu'à implémenter l'agent à travers une interface. Interface de laquelle nous nous sommes servis pour notre projet. Cependant, vu que la compétition a été discontinuée en 2012, les sources officielles ne sont plus disponibles sur le site. Alors, on a dû les retrouver de nos propres moyens et on ne peut pas être sûrs qu'il s'agisse exactement du même environnement (constat de quelques bugs pouvant avoir un impact sur l'agent).

1. Site officiel : `http://www.marioai.org`
2. Site officiel : `https://sites.google.com/site/platformersai`

3 Présentation de l'algorithme

3.1 Principe

L'algorithme du *Q-learning* s'inscrit dans le domaine de l'apprentissage par renforcement. Ainsi, le but est de trouver une police d'action optimale. Dans ce cas, la police est une fonction d'action-état qu'on notera Q. L'apprentissage de cette fonction permet à l'agent d'évaluer la récompense potentielle.

Le coeur de l'algorithme réside dans une boucle qui calcule une nouvelle *q-valeur* (valeur de la politique à un état suite à une action donnés) à chaque itération. Ces itérations peuvent être regroupées en épisodes, e.g. du début à la fin d'un jeu (mort ou arrivée à la fin), avec les $q - valeurs$ étant conservées entre épisodes. Cependant, au début, on n'a pas de valeurs de Q, donc on initialise les valeurs arbitrairement. Plusieurs choix sont possibles : on peut initialiser toutes les $q - valeurs$ au même nombre (e.g. 0), on peut utiliser une distribution uniforme ou normale centrée-réduite pour initialiser les valeurs de façon pseudo-aléatoire, etc. Evidemment, l'initialisation dépend significativement de l'implémentation de l'algorithme, c'est pourquoi on fait un point dans la sous-partie suivante et dans chaque sous-partie de la partie **Détails d'implémentation**.

Plusieurs paramètres entrent en jeu pour le calcul des valeurs de Q :

- Le facteur d'actualisation, noté γ, à valeurs entre 0 et 1, sert à pondérer les récompenses à venir. En effet, la valeur de γ détermine l'importance des récompenses futures dans la valeur de l'état actuel, i.e. plus γ s'approche de 1, plus importantes seront les récompenses futures.
- La vitesse d'apprentissage, notée α, à valeurs entre 0 et 1, sert à déterminer en quelle mesure la nouvelle valeur de l'état remplace l'ancienne valeur.

Ainsi, la valeur de la fonction à un instant $t + 1$ pour un état s_t suite à une action a_t récompensée par r_{t+1} est donnée par :

$$Q_{t+1}(s_t, a_t) \leftarrow (1 - \alpha)Q_t(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_{a \in A} Q_t(s_{t+1}, a))$$

On remarquera que, normalement, les q-valeurs associées à un état final ne sont pas mises à jour et conservent alors leurs valeurs initiales. Ainsi, il est prudent d'initialiser toutes les q-valeurs des états finaux au même nombre.

3.2 Influence et choix des paramètres

Le **facteur d'actualisation** γ, à valeurs entre 0 et 1, détermine l'importance qu'on donne aux récompenses à venir. Comme on peut le voir dans la formule d'actualisation de la police, γ influe sur la valeur donnée au terme

$\max_a Q_t(s_{t+1}, a)$. Ce terme correspond à la récompense maximale sur l'ensemble des états suivants accessibles depuis l'état courant (d'après l'estimation actuelle de la police). Ainsi, lorsque γ est proche de 0, on ne tient pas compte de cette valeur qui sert à évaluer l'utilité du passage à l'état s_{t+1}, et on privilégie la valeur de la récompense obtenue dans l'immédiat (r_t). Au contraire, si γ est proche de 1, l'agent favorisera la valeur d'utilité donnée par la police contre la récompense immédiate.

Le **nombre d'itérations** de l'algorithme est important, mais il est évident que plus ce nombre est important, plus l'algorithme approchera la solution optimale. Cependant, la convergence de l'algorithme dépend d'autres paramètres, notamment de la vitesse d'apprentissage.

La **vitesse d'apprentissage** α, à valeurs entre 0 et 1, détermine l'importance qu'on donne aux résultats précédents. La valeur peut dépendre de l'état et l'action choisis, dans ce cas on notera $\alpha \equiv \alpha(s_t, a_t)$. Comme on peut le voir dans la formule d'actualisation de la police, α pondère la somme des deux termes $Q_t(s_t, a_t)$, la valeur actuelle de la police pour un couple état-action à un instant t, et $r_t + \gamma \max_a Q_t(s_{t+1}, a)$, la nouvelle valeur d'utilité. Ainsi, lorsque α est proche de 0, on ne tient pas compte de la nouvelle valeur et l'agent apprend très lentement, tandis que lorsque α est proche de 1, on tient plus compte de la nouvelle valeur, oubliant ce que l'agent a appris par le passé.

Dans un environnement déterministe, le choix $\alpha = 1$ est optimal car une information doit toujours produire la même réaction. Dans un environnement stochastique, l'algorithme converge sous certaines conditions dépendantes de la vitesse d'apprentissage. Dans sa thèse, Watkins formule les conditions suffisantes suivantes sur $\alpha(s_t, a_t)$ pour avoir convergence de l'algorithme (i.e. produire une police optimale en temps fini) :

1. $\lim\limits_{t \to +\infty} \alpha(s_t, a_t) = 0$
2. $\alpha(s_t, a_t)$ décroît de façon monotone
3. $\sum\limits_{t=1}^{+\infty} \alpha(s_t, a_t) = \infty$

Le dernier paramètre à prendre en compte est l'ensemble des **q-valeurs initiales**. L'influence de ces q-valeurs sur l'expérience est restreinte. En effet, elles ne joueront pas un rôle important dans les dernières actualisations des q-valeurs, mais au début de l'expérience.

On peut choisir d'initialiser les q-valeurs à la même valeur pour avoir des chances équiprobables d'explorer les différentes actions possibles lors de la première itération. Néanmoins, cette approche peut poser des problèmes, car si depuis le début l'agent trouve une solution au problème, un chemin à récompenses positives, il n'explorera pas les autres options lors des itérations suivantes.

On peut choisir d'initialiser les q-valeurs avec une distribution de probabilités (e.g. uniforme, normale) dans le but d'introduire du pseudo-aléatoire à l'expérience. Cependant, le même problème se pose, d'autant plus que dans des cas extrêmes, on peut avoir des *bons* états pour l'agent initialisés avec une valeur faible et des *mauvais* états initialisés avec un valeur élevée. Ainsi, les *bons* états pourraient ne pas être visités jusqu'à ce que les valeurs des *mauvais* états soient réduits. Ainsi, cela pose un problème d'exploration.

Pour rémédier à ce problème, on introduit un paramètre d'exploration et une stratégie de décision.

3.3 Paramètre d'exploration

Dans les problèmes d'apprentissage par renforcement, comme dans autres problèmes d'apprentissage automatique, l'agent peut trouver une solution alors que celle-ci n'est pas optimale (solution suboptimale) et rentrerait dans une boucle infinie. De ce fait, comme l'algorithme ne permet de choisir que l'action dont l'utilité semble optimale à l'instant courant, l'agent n'explore pas d'autres options et pourrait ne pas trouver la solution recherchée. C'est pourquoi on introduit un paramètre d'exploration.

Il s'agit d'une valeur, notée ϵ, entre 0 et 1, dictant la fréquence à laquelle l'agent choisira une action au hasard ou la valeur donnée par l'évaluation de la police. L'introduction de ce paramètre ajoute du pseudo-aléatoire au choix de l'action, donc cela peut mener à des mauvais choix par l'agent, mais permet aussi d'échapper au problème expliqué ci-dessus.

En fonction du choix d'implémentation, une valeur proche des bornes donne plus ou moins d'importance à l'exploration ou à l'exploitation des résultats de l'algorithme d'apprentissage. Ainsi, on peut jouer sur la valeur de ce paramètre en fonction du but recherché dans une phase d'exécution. Par exemple, dans notre projet, il y a deux phases d'exécution : l'apprentissage et l'évaluation ; il est judicieux d'attribuer une valeur différente à ϵ dans chaque phase dans le but de faire que l'agent explore plus dans la phase d'apprentissage que dans la phase d'évaluation.

Il existe plusieurs stratégies pour le choix d'exploration (ϵ-*greedy*, ϵ-*decreasing*, etc). On a choisi d'utiliser la stratégie formellement plus basique (ϵ-*greedy*) qui consiste à explorer si un nombre généré pseudo-aléatoirement est inférieur à ϵ ou à choisir l'action supposée optimale sinon. La stratégie ϵ-*decreasing* qui diminue la valeur de ϵ au fur et à mesure de l'apprentissage pouvait aussi être intéressante, mais on a choisi d'utiliser ϵ-*greedy* en changeant les valeurs de ϵ uniquement entre les deux modes d'exécution car on n'a pas su déterminer une méthode d'actualisation de ϵ ni de mesurer l'impact.

3.4 Algorithme

L'algorithme dans sa forme la plus basique s'écrit :

1. Initialiser $Q_0(s_i, a_i)$ arbitrairement
2. Répéter (pour chaque épisode) :
   2.1. Récupérer l'état courant s_t
   2.2. Répéter (pour chaque sous-division de l'épisode) :
      2.2.1. Choisir une action a_t en fonction de l'état s_t en utilisant une stratégie de décision (e.g. ϵ-*greedy*)
      2.2.2. Effectuer l'action a_t
      2.2.3. Intégrer l'observation composée d'une récompense r_t et du nouvel état s_{t+1} grâce à la fonction d'actualisation

$$Q(s_t, a_t) \leftarrow (1 - \alpha)Q(s_t, a_t) + \alpha(r + \gamma \max_{a \in A_{t+1}} Q(s_{t+1}, a))$$

      2.2.4. Affecter à s_t le nouvel état s_{t+1}

   jusqu'à ce que s soit un état final.

Toutefois, il existe des variations. Par exemple, on peut actualiser α et γ en fonction de plusieurs critères (temps, nombre de fois qu'on effectue une action depuis un état, etc), avant le calcul de la nouvelle q-valeur.

3.5 Cadre d'application

Il y a de nombreux domaines d'applications :

1. Dans les usines, le réapprovisionnement est une problématique récurrente. Il y a souvent de nombreuses contraintes et plusieurs fournisseurs ce qui rend la tâche très compliquée. Un algorithme de Q-learning permet par exemple de pouvoir gérer ces problèmes d'approvisionnements en proposant les demandes à réaliser en fonction de chaque situation !
2. Dans le domaine de la finance, certains algorithmes de Q-Learning apprennent à optimiser un porte-feuille financier en adoptant les meilleurs décisions possibles.
3. Le Q-learning est aussi utilisé dans le domaine de la robotique. Le caractère discret de l'ensemble d'actions et états des robots permet l'application directe du Q-learning au choix d'action (notamment pour le déplacement).

4 Modélisation du jeu

4.1 Ensemble d'états

Afin de bien définir l'ensemble d'états, il faut choisir quelles informations de l'environnement l'agent doit intégrer. Nous décomposons tout d'abord la fenêtre observable depuis Mario AI en carrés. Puis nous définissons l'état de l'agent de la manière suivante :

1. Le statut de Mario : 0 (petit), 1 (grand), 2 (mode feu)
2. La direction de la vélocité de Mario : il y a 8 directions et l'état lorsque Mario ne bouge pas. On a donc 9 valeurs allant de 0 à 8
3. Si Mario est coincé : 0 ou 1
4. Si Mario est sur le sol : 0 ou 1
5. Si Mario peut sauter : 0 ou 1
6. Si Mario vient de se cogner contre une créature : 0 ou 1
7. Ennemis très proches dans une portée 3*3 (ou 4*3 lorsque Mario est grand ou en mode feu) autour de lui
8. Ennemis proches est dans une portée de 7*7 (ou 8*7 lorsque Mario est grand ou en mode feu) autour de lui.
9. Ennemis lointains : dans une portée de 11*11 (ou 12*11 lorsque Mario est grand ou en mode feu) autour de lui.
10. Si un ennemi a été écrasé : 0 ou 1
11. Si un ennemi vient d'être tué par une boule de feu : 0 ou 1
12. Obstacles : 4 valeurs booléennes qui représentent s'il y a un obstacle (bloc, bord intraversable, pot à fleur, canon, échelle) immédiatement à droite de Mario.

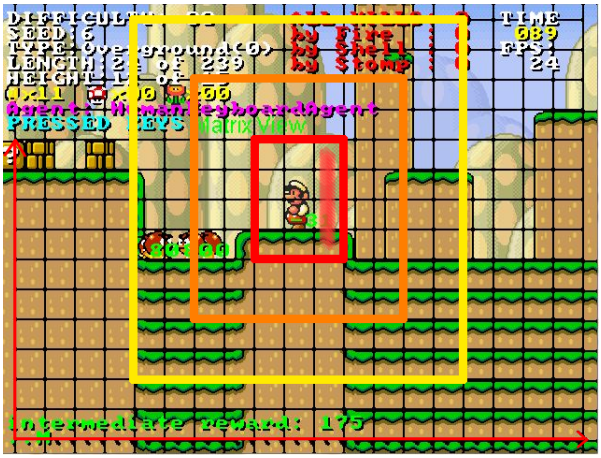


FIGURE 2 – Modélisation de la scène de jeu : représentation des 3 fenêtres d'observation des ennemis et repère.

4.2 Actions

Mario peut effectuer 7 actions différentes, qu'on regroupe en trois ensembles :

- Déplacement horizontal D_h
  1. Aller à droite
  2. Aller à gauche
  3. Ne pas se déplacer horizontalement
- Déplacement vertical D_v
  4. Sauter
  5. Ne pas sauter
- Modification du mouvement et attaque M
  6. Courir ou lancer une boule de feu
  7. Ne pas courir et ne pas lancer de boule de feu

En effet, chaque action (ou inaction) correspond à une touche virtuelle de la plateforme MarioAI. Ainsi, chaque action correspond à un triplet $(h, v, m) \in D_h \times D_v \times M$.

4.3 Récompenses

La valeur de récompense est calculée en fonction des différentes valeurs décrivant l'état et les variations de distance (de gauche à droite) et hauteur (du bas vers le haut) entre l'état précédent et le nouvel état.

D'une manière générale l'agent est récompensé positivement lorsqu'il avance vers la droite et qu'il prend de la hauteur. Tuer un ennemi est aussi récompensé positivement car cela permet d'une part de marquer des points mais diminue aussi les risques pour Mario ! La récompense pour déplacement vers la droite est aussi diminué lorsqu'un un ennemi s'approche.

On doit aussi tenir en compte que, si l'on récompense le mouvement vers la droite, il existe la possibilité de rester bloqué. Ainsi, pour rémédier à ce problème, une récompense négative est attribuée à l'agent lorsqu'il est bloqué trop longtemps.

La méthode de calcul et la valeur des récompenses est complètement arbitraire et fait partie de la modélisation et implémentation de l'environnement et du jeu.

5 Détails d'implémentation

5.1 Principe

Le principe de l'implémentation est assez direct : pour chaque état s, on possède un tableau de Q-valeurs donnant l'utilité de chaque action possible depuis cet état. A chaque état s, l'agent n'a qu'à observer son tableau de Q-valeurs (appelé Q-table) et choisir l'action avec la plus grande Q-valeur (si l'agent ne choisit pas d'explorer).

Cette approche semble assez naturelle mais elle possède un défaut important. En effet, cette approche demande l'utilisation d'une grande quantité de mémoire. Nous avons dans notre représentation de l'environnement plus de 150000 états différents (cf. détail ci-dessous) avec en général $3 \times 2 \times 2 = 12$ actions différentes par état (cf. Modélisation).

En plus, pour l'actualisation de la vitesse de convergence, on a besoin d'une deuxième collection qui tienne compte du nombre de fois qu'une action a été choisie depuis un état vers un autre état.

Le reste de l'implémentation consiste en la modélisation d'un état, l'utilisation de l'interface MarioAI et des méthodes servant à avoir un environnement d'expérimentation adapté (utilitaires d'écriture dans des fichiers, paramètres d'apprentissage, etc).

5.2 Implémentation

5.2.1 Représentation d'un état

Les états sont implémentés par la classe *EtatMario*. On utilise une classe enveloppe abstraite *Champ* pour identifier les champs composant notre modèle de l'environnement (cf. Modélisation). Pour utiliser le moins de ressources possibles, on stocke les valeurs de ces champs dans un objet de type ***Long***. En effet, on peut stocker chaque information sous forme d'octets ou suite d'octets, qu'on *place* sur cet objet. Cette représentation nous permet d'obtenir un nombre correspondant à l'état assez facilement, et donc d'identifier chaque état par un nombre unique.

5.2.2 Calcul de la récompense

Pour le calcul de la récompense (effectué dans la méthode *EtatMario.calculerRecompense()*), on tient compte de plusieurs éléments, mais les valeurs sont arbitraires :

- Si Mario a été **bloqué** pendant plusieurs itérations, on attribue une récompense négative. Le nombre d'itérations avant de donner le statut de bloqué est défini dans la classe utilitaire *ParametresApprentissage*.

- Si Mario s'est déplacé de gauche à droite/du bas vers le haut, on attribue une récompense positive. Cependant, si un ennemi est proche, on utilise un coefficient visant à réduire la récompense de déplacement. En effet, si un ennemi est proche, on voudrait que Mario fasse attention et se déplace en accord à la position de l'ennemi.
- Si Mario a tué ou a évité un ennemi, on attribue une récompense positive. On attribue des récompenses différentes si l'ennemi a été incinéré ou écrasé.

La récompense attribuée après une action est égale à la somme de ces différents termes.

Les valeurs dynamiques utilisées dans cette somme (nombre d'ennemis tués, déplacement, etc) proviennent de méthodes de la classe Environment du paquet ch.idsia.benchmark.mario.environments.

5.2.3 Représentation de la Q-table

Nous avons utilisé comme collection une table de hashage car cela correspond parfaitement à nos besoins. En effet, la table de hashage permet de récuperer la donnée cherchée en $O(1)$ (temps quasi-constant). La clef de la table représente ainsi notre état et la valeur stockée est un tableau de Q-valeurs (nombre à virgule flottante). En fait, chaque état possède son propre tableau de Q-valeurs qui peut d'ailleurs être de taille différente : la taille correspond au nombre d'actions possibles pour cet état. Par exemple, si Mario est en l'air, il ne peut plus sauter, donc il aura un tableau contenant une Q-valeur en moins que les états où Mario est sur le sol.

Nous avons aussi une deuxième table de hashage qui comptabilise le nombre de fois où l'on a pris une certaine action a dans un certain état s. Comme nous l'avons vu précédemment les Q-valeurs sont mis à jour au travers d'une formule qui contient des coefficients qui changent au cours du temps. Ainsi il est nécessaire de savoir combien de fois on a pris chaque action dans chaque état afin de connaitre la valeur de ces coefficients et de bien mettre à jour les Q-valeurs (actualisation du paramètre α, la vitesse d'apprentissage).

Pour implémenter cette représentation nous avons utilisé trois classes : *QTable*, *QTableActions* et *TableTransitions*.

- *TableTransitions* : C'est dans cette classe que l'on recense le nombre de fois où l'on a pris une certaine action a dans un certain état s pour arriver dans un état s'. On utilise une classe interne *DonneesAction*, contenant l'information de l'état d'arrivée et l'action prise, comme valeur de la table de hashage.
- *QTable* : C'est une classe abstraite qui définit les attributs et les méthodes principales que l'on utilisera dans notre Q-table.

- *QTableActions* : C'est la classe qui hérite de Q-table. Elle possède comme attributs et méthodes importantes :
  1. Une instance de *TableTransition* qui contient le nombre de fois où chaque action a a été pris depuis un certain état s.
  2. La méthode actualiserQValeur qui actualise la Q-valeur correspondante suivant la formule d'actualisation.
  3. Et la méthode getActionOptimale qui retourne l'action qui possède la plus grande valeur dans l'état indiqué.

5.2.4 Phases d'apprentissage

L'environnement MarioAI est muni de différentes phases d'exécution. On a choisi d'utiliser une autre division de phases : INIT, APPR et EVAL (cf. énumération interne *Phase* de la classe *AgentQL*).

La phase INIT correspond au tout début de l'épisode d'entraînement. En fait, dans la plupart des niveaux, Mario commence en l'air et ne peut pas bouger. Donc, pendant la phase initiale, on ne fait rien (pas d'actualisation de Q-valeurs, pas de choix d'action ...).

La phase APPR correspond à l'entraînement de l'agent. On associe des paramètres spécifique pour qu'il y ait un apprentissage efficace et une bonne répartition exploration/exploitation dans cette phase. On peut aussi configurer l'environnement simulé de MarioAI pour optimiser le temps d'entraînement (e.g. pas d'affichage dans cette phase ni de calcul de score).

La phase EVAL correspond à l'évaluation de l'agent. De la même manière que dans la phase APPR, on associe des paramètres spécifique. Dans ce cas, on veut tester la Q-table issue de l'entraînement, donc on réduit l'apprentissage et l'exploration. En plus, pour avoir une meilleure idée de l'efficacité de notre agent, on évalue plusieurs épisodes, puis on fait une moyenne arithmétique des résultats. Les itérations lors de cette phase prennent un peu plus de temps à cause du calcul du score final et autres manipulations.

On a aussi implémenté le choix d'un mode d'exécution via la ligne de commandes. Les différents modes sont DEBUG, DEMO et EVAL (cf. énumération interne *Mode* de la classe *Evaluation*). Il suffit de passer un paramètre *-n MODE* (avec MODE un des modes acceptés) pour exécuter le programme dans chaque mode. Brièvement, le mode DEBUG ajoute des messages de débuggage, le mode DEMO entraîne l'agent puis montre la phase d'évaluation (visualisation du jeu), et le mode EVAL entraîne l'agent mais ne montre pas la phase d'évaluation et écrit les résultats d'évaluation pour future exploitation.

6 Expériences et résultats

6.1 Objectifs

Le but de nos expériences était de mettre en évidence :

1. Le bon fonctionnement de notre implémentation à travers l'évaluation des différents scores mis à notre disposition.
2. L'influence des différents paramètres de l'algorithme décrits dans la section 3.2.
3. La convergence de l'algorithme sous les conditions suffisantes décrites dans la section 3.2.

Calcul de la récompense

La formule de calcule de la récompense est l'addition des termes suivants :

- Une récompense négative si l'agent est coincé : -20. Pour cela on multiplie par une valeur indiquant si l'agent est bloqué ou pas (0 ou 1),
- Une récompense positive pour déplacement horizontal (vers la droite) : 2,
- Une récompense positive pour déplacement vertical (vers le haut) : 8,
- Une récompense négative importante en cas de collision avec un ennemi : -800,
- Une récompense positive pour incinérer les ennemis : 60,
- Une récompense positive pour écrases les ennemis : 60

Ces valeurs sont complètement arbitraires. Il faut aussi savoir qu'on a choisi de décrementer la récompense pour déplacement (horizontal) lorsqu'un ennemi approche pour inciter à l'agent à ralentir.

On prend en compte plusieurs paramètres mais non pas tous (e.g. récupération de pièce et de champignons) et on effectue des évaluations correspondantes à chaque paramètre.

6.2 Choix de α

On utilise la formule suivante pour actualiser α de façon à avoir convergence de l'algorithme :

$$\alpha_t = \frac{\alpha_0}{\text{Nombre de fois que le couple } a_t, s_t \text{ a été choisi}}$$

Pour commencer, en nous basant sur les différentes valeurs trouvées lors de nos recherches, on a fait de nombreuses expériences pour déterminer des paramètres qui nous semblaient optimaux. On est arrivé aux paramètres suivants pour l'entraînement :

- $\alpha_0 = 0.8$
- $\gamma = 0.6$
- $\epsilon = 0.3$

En effet, ces paramètres permettent à l'agent d'apprendre assez rapidement tout en utilisant les données du passé, en favorisant les récompenses futures et une exploration restrainte. En plus l'actualisation de α permet à l'agent d'identifier des états récurrents et de préférer d'utiliser l'information existante. En fait, en réduisant α, on réduit le taux d'information qu'apporte l'arrivée à ces états.

Pour l'évaluation, l'objectif est d'évaluer la Q-table produite suite à la phase d'entraînement, donc on a décidé de ne pas actualiser la Q-table pendant la phase d'évaluation. Cependant, on pose $\epsilon = 0.01$, pour réduire la fréquence d'exploration mais garder cette option.

Ainsi, on a effectué quatre expériences (variation des paramètres du Q-learning) plusieurs fois et on a choisi la meilleure exécution pour chaque expérience.

6.2.1 Expériences

L'entraînement a été fait par sous-entraînement de 20 épisodes en chaque mode (i.e. 20 épisodes Mario est petit, 20 épisodes Mario est grand, 20 épisodes Mario est en mode feu). De cette façon, on prépare Mario à des éventuelles transitions de mode, e.g. Mario en mode feu perd son pouvoir de feu mais comme on a entraîné l'agent avec grand Mario, les résultats de la Q-table restent utiles.

Expérience	α	γ
N° 1	0.3 actualisé	0.6
N° 2	0.8 actualisé	0.15
N° 3	0.8 constant	0.6
N° 4	0.8 actualisé	0.6

On a effectué 12 000 itérations d'entraînement pour avoir 200 mesures d'évaluation. Chaque mesure d'évaluation correspond à la moyenne arithmétique des résultats de 100 épisodes d'évaluation.

6.2.2 Résultats

On a obtenu les résultats suivants :

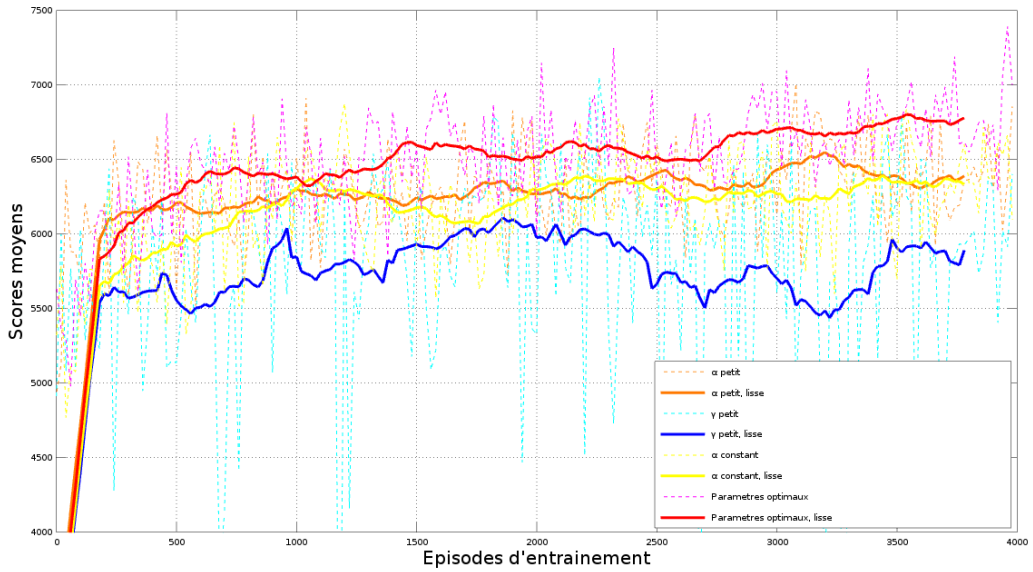


FIGURE 3 – Evaluation du score

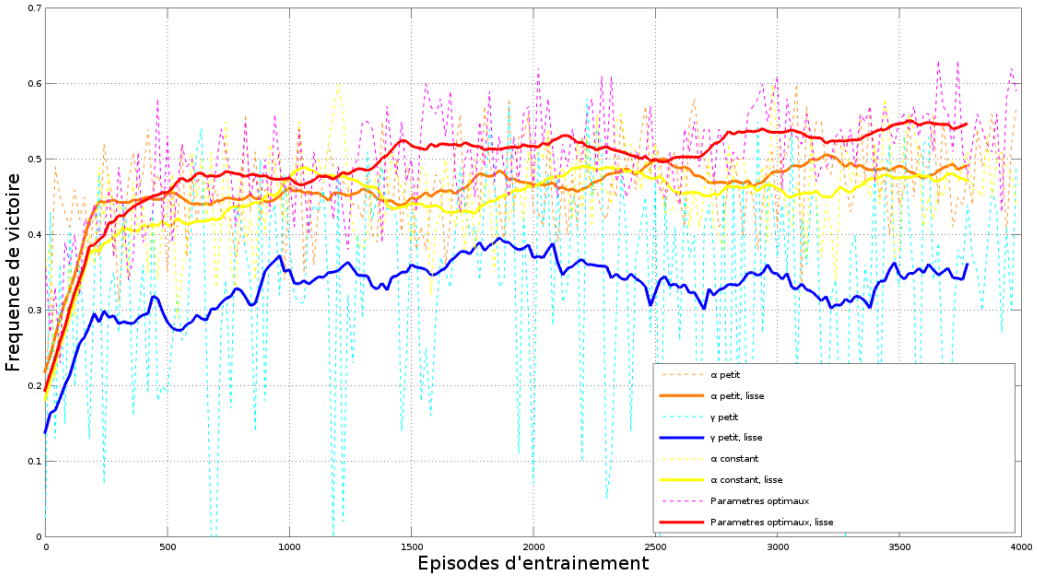


FIGURE 4 – Evaluation de la fréquence de victoire

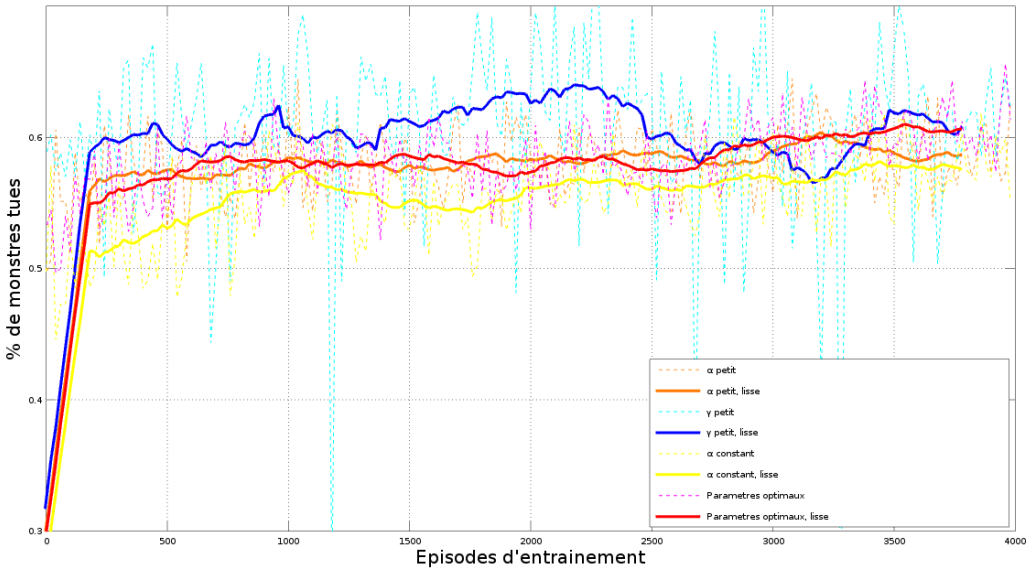


FIGURE 5 – Evaluation du pourcentage d'ennemis tués

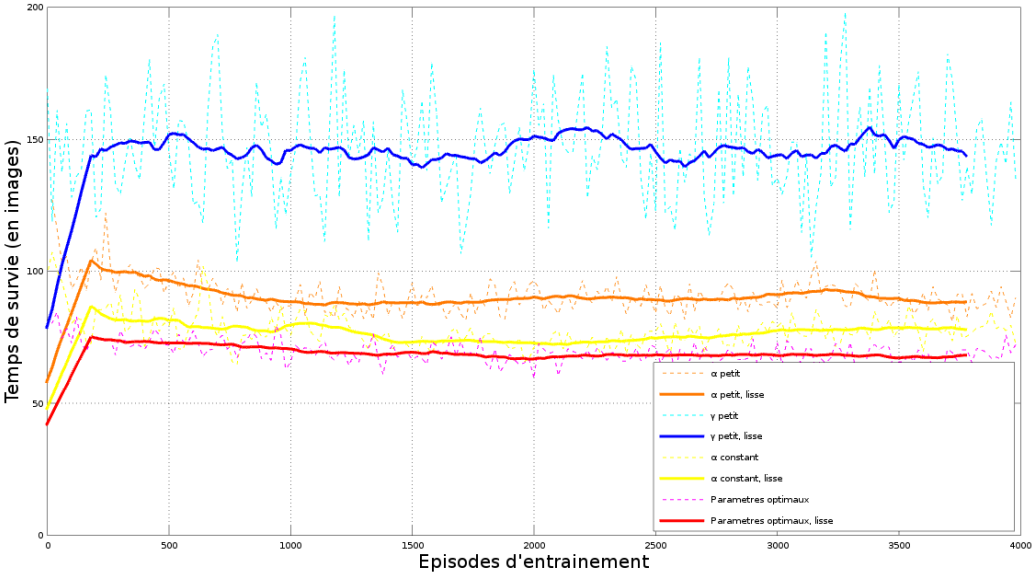


FIGURE 6 – Evaluation du temps (en images) d'un épisode (mort ou victoire)

L'utilisation des 4 métriques d'évaluation permettent d'observer l'influence des différences paramètres dans l'apprentissage de l'agent.

Vitesse d'apprentissage α faible

On peut constater que le paramètre α correspond bien à la vitesse d'apprentissage de l'agent. En effet, on peut voir dans l'évaluation du score moyen

que les scores augmentent de façon similaire.

On constate aussi qu'au début, c'est l'agent avec une vitesse d'apprentissage faible qui semble être meilleur. On pourrait dire qu'il s'agit d'un effet de bord de l'initialisation avec des valeurs aléatoires : vu qu'il apprend moins vite (i.e. les q-valeurs initiales sont privilégiées), il explore plus d'états et donc plus de chances de trouver des états plus utiles.

Facteur d'actualisation γ faible

Le facteur d'actualisation permet de moduler l'importance donnée aux récompenses court et long terme. Ainsi, un facteur d'actualisation faible devrait privilégier le choix d'actions en fonction de leurs récompenses court terme.
Les résultats de nos expériences montrent que le facteur γ a une influence très importante sur notre agent. En effet, l'agent essaie de récuperer des récompenses immédiates, donc il essaie de tuer le plus possible d'ennemis ! Cela lui pose deux problèmes.

Premièrement, il se positionnera plus près des ennemis pour essayer de les écraser et court une chance plus élevée de perdre par collision. On a constaté que même en implémentant une variable de modulation de la récompense attribué en fonction de la proximité des ennemis, l'agent myope choisi de s'approcher des ennemis.

Deuxièmement, on a constaté un bug dans la plateforme. Des ennemis sont placés sur des carreaux inaccessibles par l'agent (e.g. sous le sol). Ainsi, l'agent myope essaie d'écraser l'ennemi, mais il ne peut pas le faire. Cela explique le temps de survie bien plus élevé que pour les autres agents : le temps de survie estime le temps que l'agent passe sur un niveau, il s'agit d'un indicateur non seulement du temps que l'agent met à arriver à la fin de l'épisode mais aussi du temps que l'agent passe sur l'épisode. Ainsi, s'il reste bloqué à essayer de faire quelque chose d'impossible, ce temps augmente davantage.

On peut aussi voir que l'algorithme ne semble pas converger pour un γ si petit. En effet, on constate une forte variation des scores moyens et que la courbe ne semble pas se stabiliser au cours du temps. C'est avec ce choix de paramètre qu'on a eu les pires résultats, qu'on a dû coupé des graphiques pour qu'ils restent lisibles.

Actualisation de α

Lorsqu'on compare les résultats des expériences avec et sans actualisation de α, on peut voir que l'agent est moins efficace. Les valeurs de toutes les métriques sont inférieures pour l'agent sans actualisation.

Cependant, au contraire de ce qu'on attendait, il semblerait que l'algorithme converge vers un police optimale. On voit sur la courbe des scores

moyens que celle-ci se stabilise avec le temps et que les courbes des autres métriques ont une monotonie semblable à celles des courbes pour les paramètres optimaux.

Ainsi, cette expérience nous a permis de conjecturer que les conditions de convergence décrites plus haut ne sont que des **conditions suffisantes de convergence**.

Remarque 1 : Ces expériences ont aussi permis d'évaluer notre modélisation et implémentation. Certes, on n'a pas pris en compte plusieurs critères qui pourraient être critiques pour l'évaluation de l'efficacité de l'agent (critères de récompense du jeu), mais les critères choisis nous on permis d'avoir des résultats correctes et cohérents avec la théorie.

Remarque 2 : on voudrait ajouter, à titre indicatif, des mesures de temps effectués. En fait, au début on ne savait pas combien de temps l'entraînement allait prendre, c'est pourquoi on s'est pressé à commencer les expériences au plus vite possible. Cependant, on s'est vite rendu compte que la police convergeait assez rapidement vers une police correcte (après près de 3000 itérations d'entraînement). De plus, on a fait de notre mieux pour optimiser le temps d'exécution de notre code. Le temps mis est relativement faible :

- 1500 épisodes d'entraînement avec 2500 épisodes d'évaluation : 3 minutes 24 secondes
- 3000 épisodes d'entraînement avec 5000 épisodes d'évaluation : 8 minutes 47 secondes
- 6000 épisodes d'entraînement avec 10000 épisodes d'évaluation : 18 minutes 24 secondes
- 12000 épisodes d'entraînement avec 20000 épisodes d'évaluation : 41 minutes 52 secondes

7 Conclusion

7.1 Conclusion du projet

Ce projet s'est révélé très enrichissant dans la mesure où il a consisté en une approche concrète du métier d'ingénieur, mais aussi de chercheur. En effet, on a passé près d'un tiers du temps à faire des recherches, à nous documenter, puis à faire un modèle, à l'implémenter, le tester et finalement l'exploiter. L'application de cette démarche scientifique, en plus du travail en équipe et la gestion de projet, seront des aspects essentiels de notre futur métier.

Cependant, on a aussi acquis beaucoup de connaissances sur un domaine qui nous était inconnu auparavant. L'apprentissage automatique, et plus particulièrement l'apprentissage par renforcement, le thème de ce projet, est un champ d'étude de l'intelligence artificielle qui se développe très rapidement et qui devient assez populaire de nos jours. Ainsi, la réalisation d'un projet sur cette notion nous a permis non seulement d'enrichir nos connaissances scientifiques et de découvrir un nouveau domaine, mais aussi de gagner une expérience significative dans le monde du travail.

7.2 Perspectives de continuation et améliorations

7.2.1 Autres méthodes d'apprentissage par renforcement

Il serait intéressant de comparer les résultats obtenus par notre modélisation et par l'algorithme du Q-learning avec d'autres représentations et algorithmes.

On pourrait comparer l'algorithme du Q-learning avec un autre algorithme proche du Q-learning tel que SARSA ou d'autre algorithmes basés sur le *Temporal Difference learning*.

On pourrait étudier aussi les différences entre cette méthode d'apprentissage en ligne et des méthodes d'apprentissage hors ligne. Si bien qu'on peut comparer l'efficacité de l'agent, on pourrait aussi évaluer les différences en temps d'exécution et la quantité de mémoire utilisée.

7.2.2 Apport des réseaux de neurones

Lors de notre projet, on a soulevé plusieurs problèmes connus de l'apprentissage par renforcement utilisant des tableaux et des structures de données semblables. Néanmoins, lors de la phase de documentation, on a aussi trouver des solutions, dont la plus intéressante, et peut-être la plus efficace, est l'utilisation d'un réseau de neurones.

En fait, un réseau de neurones artificiel peut servir à approcher une fonction. Donc, on a développé des Q-networks, des réseaux de neurones approchant la police optimale Q. Plusieurs travaux ont été publiés et la popularité de cet outil ne cesse de croître, notamment depuis la publication *Playing Atari*

with Deep Reinforcement Learning de la société *DeepMind* en 2013. Dans ce papier, on explique comment en utilisant un réseau de neurone, entraîné avec une variante de Q-learning (Double Q-learning), un agent a réussi à jouer de façon exceptionnelle plusieurs jeux. On voit bien que l'utilisation d'un réseau de neurones pourrait rémedier plusieurs des problèmes soulevés comme, par exemple, le stockage et recherche des Q-valeurs et l'approximation d'état semblables.

Références

[1] Thomas Degris. *Apprentissage par renforcement dans les Processus de Décision Markoviens Factorisés*, Thèse de doctorat en Informatique, sous la direction Olivier Sigaud, Université Paris IV, 2007, 223 p.
http://people.bordeaux.inria.fr/degris/papers/These_Thomas_Degris.pdf

[2] Abdselam Boularias. *Apprentissage par renforcement*, Support de cours, Université Laval, 2006.
https://pdfs.semanticscholar.org/8516/77fdab6be05bc8b6bf8f3f6ca671422f2ee3.pdf

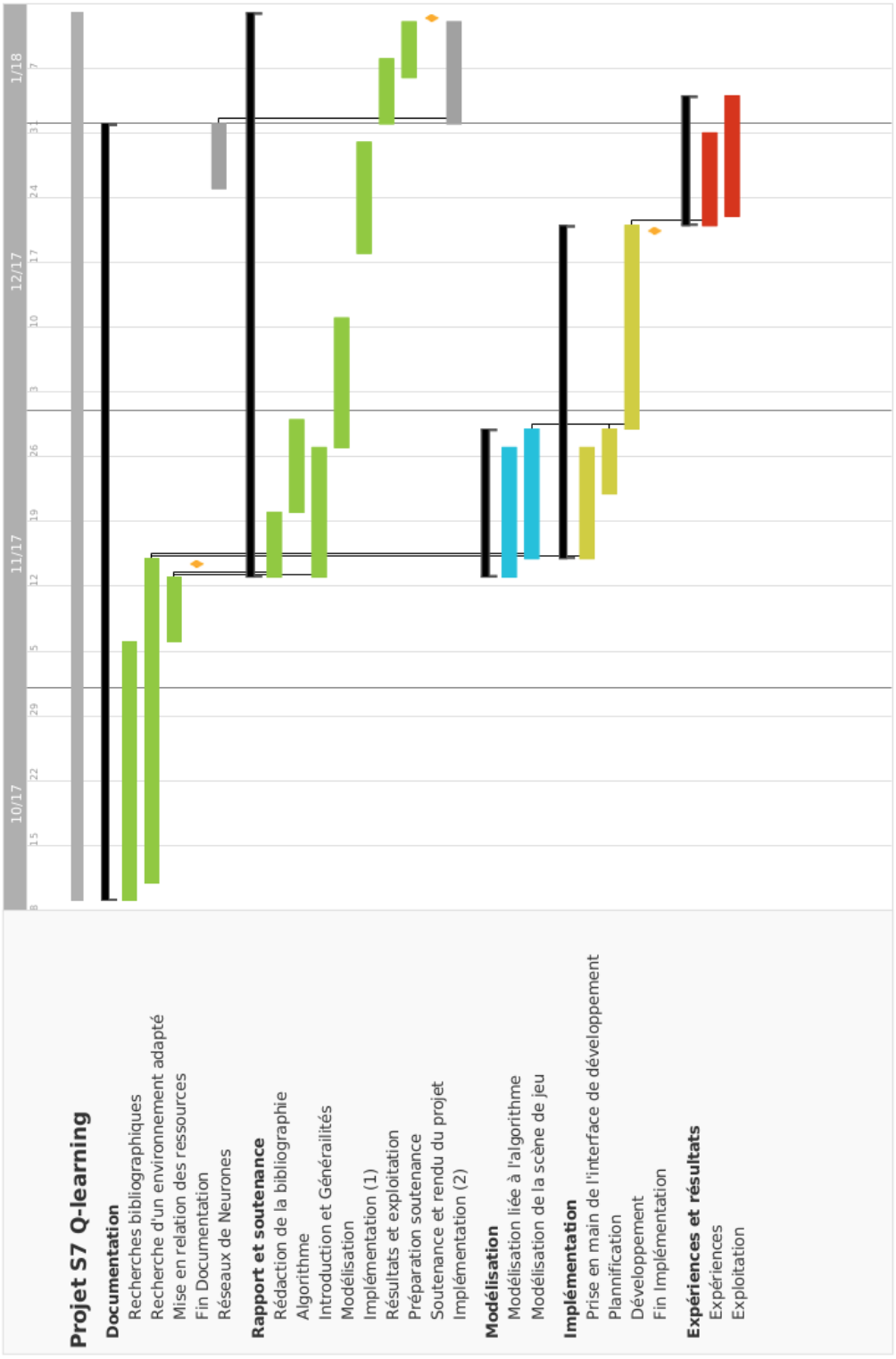
[3] Francisco S. Melo. *Convergence of Q-learning : A simple proof*, Institute for Systems and Robotics, Instituto Superior Técnico de Lisboa
http://users.isr.ist.utl.pt/~mtjspaan/readingGroup/ProofQlearning.pdf

[4] Romaric Charton. *Des agents intelligents dans un environnement de communication multimédia : vers la conception de services adptatifs*, Thèse de Doctorat en Informatique, sous la direction de Anne Boyer et Jean-Paul Haton, Université Henri Poincaré.
http://slideplayer.fr/slide/466397/

[5] *Planning and Learning*, Support de cours, Technische Universität Chemnitz, 2010.
https://www.tu-chemnitz.de/informatik/KI/scripts/ws0910/ml09_9.pdf

[6] *Reinforcement learning*, Support de cours, University of Sydney.
http://www.cse.unsw.edu.au/~cs9417ml/RL1/algorithms.html
http ://mnemstudio.org/path-finding-q-learning-tutorial.htm

Annexes



Annexe 7 – Diagramme de Gantt du le déroulement du projet