Apprentissage automatique
Algorithme de Q-Learning

Implémentation et analyse d'un algorithme

Damien DOUTEAUX

Avril
24
2017

Table des matières

Liste des figures

Introduction

Ce rapport s'intéresse à l'implémentation d'un algorithme de QLearning en utilisant Matlab. Pour présenter ce travail, nous orienterons ce rapport autours de trois axes.

Dans un premier temps, nous réintroduirons les différents éléments théoriques vus afin de remettre ce travail dans son contexte. Nous redéfinirons en partie tous les termes usuels de l'apprentissage par renforcement pour pouvoir les utiliser librement par la suite. Cette partie sera également l'occasion de présenter l'algorithme implémenté sous un point de vue « théorique ».

Nous nous intéresserons ensuite à l'implémentation de l'algorithme proposé en Matlab. Nous détaillerons ainsi tous les éléments de code proposé ainsi que les moyens de lancer cet algorithme.

Pour terminer, nous regarderons les résultats obtenus en utilisant notre algorithme. En particulier, nous essaierons de retrouver les résultats proposés dans les supports de cours. Nous conclurons ce rapport en regardant l'influence des paramètres de l'algorithme sur les résultats obtenus.

Ce rapport s'accompagne de trois fichiers Matlab :

QLearning_Douteaux.m *train_q_model.m* *optimize_behaviour.m*

En cas de question sur les contenus de ce rapport ou les fichiers sources proposés, n'hésitez pas à me contacter à l'adresse mail suivante :

damien.douteaux@ecl13.ec-lyon.fr

1 • Rappels théoriques

1.1 L'apprentissage par renforcement

Cette partie représentera les idées principales de l'apprentissage par renforcement et du QLearning.

1.1.1 Principaux aspects de l'apprentissage par renforcement

L'apprentissage par renforcement est une méthode de *machine learning* adapté pour l'apprentissage de compétence dans un milieu complexe et incertain. L'idée est alors d'avoir un agent qui va agir en fonction de récompenses fournies par l'environnement dans lequel il évolue pour ses actions.

On retrouve ainsi les principales caractéristiques suivantes :

- *Pas de superviseur* Ainsi, les décisions ne sont pas prises en fonction de labels traités, mais uniquement via des récompenses aux actions menées par l'agent.
- *Différé* L'agent va exécuter son action et aura un retour ultérieur à cette dernière, elle n'est donc pas traitée en instantané. Le processus suivi est également séquentiel.
- *Influence de l'agent* Les données qu'il reçoit dépendent de ses actions, il influe donc sur les données qu'il recevra de son environnement.

1.1.2 Qu'est-ce qu'un agent ?

Avant de définir formellement ce qu'est un agent, nous allons commencer par voir comment ce dernier peut intéragir avec l'environnement.

Interragir avec l'environnement L'agent par intéragir avec son environnement via trois éléments :

- *Observation* O_t Ce que l'agent perçoit de son environnement.
- *Action* A_t Celle qui est prise par l'agent.
- *Récompense* R_t Nous les discuterons en détail juste après.

On définit ainsi une expérience comme une séquence sous la forme :

$$o_1 \;\; r_1 \;\; a_1 \;\; \ldots \;\; a_{t-1} \;\; o_t \;\; r_t \tag{1}$$

Ainsi, une expérience (du point de vue de l'environnement) est caractérisé par une suite d'observations et de récompenses envoyées à l'agent, suivi par une action de l'agent.

Les récompenses Les récompenses seront notées R_t dans la suite de ce rapport. Elles correspondent à un signal de retour de l'environnement sur les actions menées par l'agent dans ce dernier.

Ces dernières vont ainsi permettre à l'agent de savoir si ses actions étaient positives ou négatives dans la modélisation qui est fait de cet environnement. Ainsi, pour optimiser son comportement l'agent va chercher à optimiser ses récompenses futures.

Définir un état L'état s_t dans le processus d'apprentissage est définit sous la forme d'un résumé des expériences comme (1) via (2).

$$s_t = f(o_1, r_1, a_1, \ldots, a_{t-1}, o_t, r_t) \tag{2}$$

On remarque donc que l'état dépend différement de l'expérience menée.

Définir l'agent Suite à cette définition de l'expérience, on peut définir un agent comme un élément qui peut inclure un des éléments suivants :

- *Une politique d'action* On la note π, cette dernière permet de déterminer le comportement de l'agent. Elle permet ainsi à l'agent d'associer un état à une action, ie. dans le cas déterministe $a = \pi(s)$ (l'action a est associée à l'état s pour la politique π).

- *Une fonction de valeur* Cette fonction va permettre d'estimer la récompense future, et ainsi d'estimer la valeur des états. L'idée est alors de savoir estimer les actions à partir d'un certain été donné. La définition d'une valeur est souvent liée à une politique, elle se définit alors comme l'espérence des observations à venir, soit :

$$v_\pi(s) = \mathbb{E}\left[\sum_{i=1}^{\infty} \gamma^i R_{t+i} | S_t = s\right] \tag{3}$$

  Dans cette relation, le cœfficient γ mesure l'actualisation via les différentes récompenses.
- *Un modèle* Ce dernier permet à l'agent de prédire ce que l'environnement fera à partir d'état donné pour des actions données. L'objectif est ainsi pour l'agent de pouvoir prédire l'état qui suivra via une méthode $\mathcal{P}$ mais également la récompense suivante via une méthode $\mathcal{R}$.
  - *Prévision de l'état suivant* L'état va dépendre d'une action à un état donné. Ainsi de manière générale, cela revient à connaître la probabilité d'obtenir l'état s' sachant que l'on est en état s et que l'on va appliquer l'action a, soit :

$$\mathcal{P}_{ss'}^a = \mathcal{P}\left[S_{t+1} = s' | S_t = s, A_t = a\right] \tag{4}$$

  - *Prévision de la récompense suivante* Pour cette prévision, il s'agit de connaître l'espérence de la récompense en $t+1$ sachant que l'on est dans l'état s_t et que l'on va utiliser l'action a, soit :

$$\mathcal{R}_s^a = \mathcal{E}\left[R_{t+1} | S_t = s, A_t = a\right] \tag{5}$$

  Ce modèle sera ainsi appris par l'expérience et servira à planifier ultérieurement les actions.

Ainsi, un agent peut comprendre tout ou partie de ces éléments pour être défini.

1.1.3 Apprentissage d'un modèle

L'apprentissage d'un modèle pour un agent s'oriente autours de deux axes principaux, l'exploration et l'exploitation.

Dans un premier temps, une phase exploratoire va nous permettre de trouver des informations sur l'environnement dans lequel on évolue.

Dans un second temps, l'exploitation revient à exploiter les données obtenues en exploration pour maximiser la récompense et ainsi optimiser l'action.

De manière optimale, nous verrons que la meilleure stratégie est d'utiliser conjointement ces deux étapes.

1.1.4 Le Q-Learning

Fonction d'action-valeur Ces fonction notées Q, permettent de représenter la récompense future totale qui est attendue à partir d'un état s et d'une action a. Elles permettent ainsi de généraliser les fonctions de valeurs v_π évoquées avec (3), en cherchant la récompense totale future, soit (6).

$$Q^\pi(s,a) = \mathbb{E}\left[\sum_{i=1}^{\infty} \gamma^i r_{t+i} | s, a\right] \tag{6}$$

De la fonction d'action-valeur au Q-Learning L'objectif du Q-Learning est de pouvoir approcher un optimum de la fonction Q. L'optimum d'une telle fonction est définie par (7).

$$Q^*(s,a) = \max_\pi Q^\pi(s,a) = Q^{\pi^*}(s,a) \tag{7}$$

On remarque ainsi que la fonction d'action-valeur optimale correspond également à une politique optimale π^*. Une fois cette politique définit, on peut alors directement générer une action de manière optimale via (8) en utilisant cette politique optimale π^*.

$$\pi^*(s) = \text{argmax}_a Q^*(s, a) \quad (8)$$

Il est à remarquer cependant que le Q-Learning ne va pas apprendre *exactement* la fonction Q, mais une *approximation* de cette dernière. De plus, on notera également qu'à terme la décision ce fera donc en maximisant *toutes* les décisions.

1.2 Un algorithme de QLearning

1.2.1 *Principe générique*

Le principe générique de l'algorithme revient à mettre à jour la fonction Q de manière itérative. Les quatres étapes principales étant alors :

- *1.* L'agent commence par recevoir l'observation o_t de l'environnement, l'objectif de la suite est d'utiliser cette observation pour afficher la fonction d'action-valeur Q.
- *2.* Notre agent va choisir une action a_t. Ce choix peut être fait de manière aléatoire ou alors en suivant une politique particulière.
- *3.* L'environnement réagit à l'action a_t en renvoyant une nouvelle observation o_{t+1} et en renvoyant en même temps la récompense R_t pour l'action proposée.
- *4.* L'agent dispose alors de la récompense à son action entre les deux états s_t et s_{t+1}, une mise à jour proposée est alors définir en (9).

$$Q_{t+1}(s_t, a_t) = \underbrace{Q_t(s_t, a_t)}_{\text{Valeur actuelle}} + \alpha_t(s_t, a_t) \cdot \left[\overbrace{R_{t+1} + \gamma \cdot \underbrace{\max_a Q_t(s_{t+1}, a)}_{\text{Valeur optimale estimée}}}^{\text{Valeur apprise}} - Q_t(s_t, a_t) \right] \quad (9)$$

Dans la relation (9), on observe la présence de deux paramètres que nous détaillerons en Section 1.2.2. Concernant les autres termes de la relation (9), on remarque que cette dernière revient à appliquer un correctif à la valeur actuelle $Q_t(s_t, a_t)$.

Pour cette mise à jour, comme nous l'avons dit, l'objectif est de prendre la décision en fonction de toutes les valeurs futures, on chercher donc le maximum optimal pour la suite. On calcule ainsi une nouvelle valeur apprise en combinant la récompense et l'optimal estimé auquel on retire la valeur actuelle (pour savoir si on fait mieux ou moins bien que ce que l'on avait avant).

1.2.2 *Les paramètres de l'algorithme*

Le principe fait intervenir deux paramètres α et γ, que nous allons présenter ici.

Paramètre α Ce paramètre permet de représenter la vitesse d'apprentissage, et permet de savoir à quel point la nouvelle valeur va « effacer » celle précédemment calculer.

Ainsi, ce paramètre est compris entre 0 (pas de mise à jour) et 1 (on oublie totalement la valeur précédente).

Le cours proposait des valeurs aux alentours de $0,1$, nous regarderons ce qu'il en est avec notre implémentation.

Paramètre γ Ce facteur permet de représenter l'actualisation, ie. l'importance des récompenses futures sur la nouvelle valeur. L'idée est que dans l'actualisation, la récompense reçue R_{t+1} est augmentée d'une pondération des récompenses futures que l'agent pense obtenir en fonction de la fonction Q qu'il connaît déjà.

Comme pour α, le paramètre prend ses valeurs entre 0 (on ne considère que la récompense courante R_t) et 1 (égalité entre les récompense courante et celles à venir).

1.2.3 *L'algorithme implémenté*

In fine, l'algorithme que nous implémenterons ici reprend les quatre points vus à la Section 1.2.1.

L'algorithme qui en découle est alors directement le suivant :

```
Initialiser gamma ; alpha et les récompenses R
Q <- 0

Pour chaque épisode
    | Sélection aléatoire de l'état initial (s_0).
    | Tant que l'état s_t n'est pas final
    |  | Sélection aléatoire d'une action possible a_t depuis s_t.
    |  | Considérer s_(t+1) = a_t(s_t).
    |  | Déterminer max(Q) pour s_(t+1) en considérant sur toutes les
    |  |       actions a possibles depuis s_(t+1).
    |  | Màj de Q selon la relation vue.
    |  | Màj état courant (s(t+1) <- s_t).
Renvoyer Q
```

Cet algorithme reprend ainsi la plupart des éléments déjà vus. Nous verrons également par la suite qu'il est possible d'y greffer une solution pour que l'algorithme s'arrête une fois la convergence atteinte.

2 • Présentation du code

2.1 Structure du code proposé

Avant de nous intéresser explicitement au code proposé, nous allons en reprendre sa structure. Nous proposons pour cela trois fichiers :

- *Fichier* `QLearning_Douteaux.m` Ce fichier contient le script générique pour exécuter les différents codes réalisés. Il propose en particulier la définition des différents paramètres que nous verrons à la Section 2.2.
- *Fichier* `train_q_model.m` Ce fichier contient la fonction permettant d'entraîner la fonction Q. L'algorithme utilisé sera détaillé à la Section 2.3.
- *Fichier* `optimize_behaviour.m` Ce fichier contient la fonction permettrant d'utiliser la fonction Q entraînée pour obtenir un comportement optimal. L'algorithme utilisé sera détaillé à la Section 2.4.

La suite de cette section reviendra ainsi à détailler les contenus et l'utilisation de ces trois fichiers.

2.2 Paramètres de l'algorithme

Comme nous l'avons évoqué, le fichier `QLearning_Douteaux.m` contient tous ces différents paramètres.

2.2.1 *Paramètres déjà présentés*

Ce fichier redéfinit en premier lieu les cinq paramètres déjà retenus pour le modèles, à savoir :

- `R` Cette matrice représentera les récompenses d'une action à partir d'un état.
- `Q` Il s'agit de l'initialisation de la matrice d'action-valeur qui représent la fonction d'action-valeur.
- `alpha` Il s'agit de la vitesse d'apprentissage.
- `gamma` Il s'agit du facteur d'actualisation pour la prise en compte des récompenses futures.
- `nb_iter` Il s'agit du nombre maximal d'itérations à réaliser dans l'apprentissage de `Q`.

Les valeurs retenues sont alors celles proposées dans le sujet. Le code se déduit alors immédiatement :

```
%% Matrice de récompense
%  Cette matrice représente les récompenses que recevra l'agent
%  en fonction des décisions de mouvement qu'il prendra.
%  Les valeurs sont celles proposées dans le support de cours.
R = [[-1 -1 -1 -1  0 -1];
     [-1 -1 -1  0 -1 100];
     [-1 -1 -1  0 -1 -1];
     [-1  0  0 -1  0 -1];
     [ 0 -1 -1  0 -1 100];
     [-1  0 -1 -1  0 100]];

%% Matrice d'action-valeur
%  Représente la valeur des actions calculées par l'algorithme.
%  Cette matrice sera mise à jour par essais successifs.
%  Par défaut on ne saît rien, donc tous les coefficients à 0.
Q = zeros(size(R));

%% Paramètres de l'algorithme
%  Alpha : la vitesse d'apprentissage.
%  Gamma : facteur d'actualisation.
%  Nb_iter : nombre d'épisodes à réaliser (au plus).
alpha = 1;
```

```
gamma = .8;
nb_iter = 100000;
```

2.2.2 *Paramètres pour gérer la convergence*

En plus de l'algorithme proposé dans le cours, nous avons ajouté une partie de code pour que l'exécution de ce dernier s'arrête à convergence de la matrice *Q*.

Pour cela, nous avons ajouté deux paramètres différents pour vérifier si on aboutit à une convergence de l'algorithme :

- *epsilon* Après chaque épisode, on obtient une nouvelle matrice d'action-valeur Q_{t+1}. L'idée est alors de vérifier si $\|Q_{t+1} - Q_t\|_2 < \varepsilon$, plus ce comportement sera présent, plus cela signifiera que la convergence est presque atteinte.
- *nb_convergence* La condition pour tester la convergence précédente pose un problème. En effet, nous choisissons notre état de départ au hasard. Ainsi, il est possible de faire deux fois les mêmes actions, et donc la condition proposée peut être vérifiée à une étape sans qu'il n'y ait convergence (nous le verrons dans les applications). Pour parer à ceci, nous avons donc mis en place un nombre d'itérations successives où il faut que la condition soit vérifiée pour considérer qu'il y ait convergence.

Une valeur possible à retenir pour ces paramètres est proposée dans le code suivant :

```
%% Paramètres de l'algorithme
%  Nb_convergence : nombre d'étapes où il faut que ||prev_Q-Q||_2 < epsilon
%                   pour considérer qu'il y eu convergence de l'algorithme.
%  Epsilon : seuil pour ||prev_Q - Q||_2 pour déterminer la convergence.
nb_convergence = 100;
epsilon = 0.001;
```

Dans la mesure où tous les paramètres utiles ont été présentées, nous pouvons désormais nous intéresser à l'apprentissage du modèle.

2.3 Algorithme de Q-Learning pour l'apprentissage

Nous allons ici détailler les différentes étapes de l'implémentation de l'algorithme d'apprentissage de Q-Learning implémenté. L'objectif n'est pas ici de faire un détail très important des aspects liés au langage, mais plus dans la structure de ce dernier.

2.3.1 *Appeler l'algorithme*

En réutilisant les paramètres vus à la Section 2.2, l'appel à l'algorithme se fait alors simplement de la manière suivante :

```
function [ Q ] = train_q_model( R, Q_init, alpha, gamma, nb_iter, nb_convergence,
    epsilon )
```

Pour pouvoir exécuter ce code, il faudra uniquement faire attention à ce que toutes les grandeurs aient été définies et que le fichier *train_q_model.m* soit dans le même répertoire que le script appelant cette méthode.

2.3.2 *Structure générale du code*

Le code se découpe en quatre parties, qui reprennent celles déjà observées dans la partie théorique :

- *Initialisation* Une partie de ces dernières ont été déportées, mais certaines valeurs sont initialisées dans la fonction.
- *Une boucle principale* Cette dernière applique la mise à jour de *Q* épisode par épisode.

- *Déroulé d'un épisode* Au sein de la boucle principale, une boucle qui permet de faire de « petites » mises à jour de Q action par action. Cette boucle s'arrête quand un état final est atteint.
- *Court-circuit et normalisation* Le reste de code permet d'arrêter ce dernier à la convergence et de normaliser le résultat obtenu.

Nous allons détailler ces blocs un à un dans la suite. Pour avoir une vision globale, nous vous proposons de regarder directement le code (beaucoup commenté pour guider le lecteur).

2.3.3 *Initialisation*

La plupart de ces dernières sont réalisées dès l'appel de la fonction. On retrouve ainsi l'initialisation de Q avec une valeur passée en paramètres.

Un point qui n'était pas prévu dans l'algorithme initial était l'initialisation du nombre d'essais successifs pour lesquels $\|Q_{t+1} - Q_t\|_2 < \varepsilon$. Ce nombre d'essais est stocké dans la variable *conv_count* qui est donc initialisée à *0*.

On remarquera enfin une ligne pour optimiser l'affichage pour l'utilisateur de manière plus « propre ». Tous ces éléments de codes se retrouvent dans le code suivant :

```
%% Initialisation des grandeurs génériques de l'algorihthme.
% Initialisation de la matrice d'action-valeur (Q).
Q = Q_init;

% Nombre d'épisodes successif où ||prov_Q - Q||_2 < epsilon.
conv_count = 0;

% Un peu d'affichage pour l'utilisateur
disp('--------------------------------');
```

2.3.4 *Boucle principale*

Cette boucle principale va correspondre aux mises à jour de Q épisode par épisode. Cette dernière va ainsi être exécutée au plus un nombre *nb_iter* de fois.

Le début de cette boucle principale est marqué par la définition des deux grandeurs globales pour le déroulé de l'épisode, à savoir :

- *final_state* Ce booléen permet de savoir si l'épisode est terminé, ie. si l'état courant est un état final.
- *init_state* Cet entier va représenter l'état initial qui est tiré au hasard parmis tous les états possibles.

En plus de ces paramètres, on remarquera que l'on stocke la valeur de Q avant le déroulé de l'épisode pour réaliser les test $\|Q_{t+1} - Q_t\|_2 < \varepsilon$. Le code qui en découle est alors immédiat :

```
%% Boucle principale de l'algorithme.
%  Les étapes de l'algorithme sont spécifiées directement dans
%  le code de ce dernier.
for i=1:nb_iter
    % Booléen pour savoir si l'épisode a abouti à un été final.
    final_state = false;

    % Tirage au sort de l'état de départ.
    init_state = randi(size(R,1));

    % Sauvegarde de la matrice Q pour regarder || prev_Q - Q ||_2.
    previous_Q = Q;

    << Déroulé d'un épisode >>

    << Court-circuit si convergence >>
end
```

On pourra également remarquer la partie relative à la convergence en fin de boucle que nous détaillerons un peu plus tard.

2.3.5 *Déroulé d'un épisode*

Le cœur de l'implémentation est en fait le déroulé de l'épisode. Ce dernier est réalisé à l'aide d'une boucle `while` qui va faire son travail tant que l'on n'est pas situé dans un état final.

Dans le détails, le déroulé de cette boucle est le suivant :

- *Recherche de l'état suivant* Pour l'état courant dans lequel on est, on recherche quels sont les états potentiels pour l'étape suivante. Le choix définitif est fait par tirage au sort parmis les possibilités.

  Dans le code, on remarquera le `find(R(init_state,:)+1)` qui part du principe que les transitions impossibles sont notées `-1` dans la matrice de récompense. Dans la mesure où `find` cherche par défaut les valeurs non nulles, cette astuce nous permet de trouver les états possibles à moindre coût.
- *Mise à jour de* `Q` Pour mettre à jour cette matrice, on utilise la relation (9). Le code proposé va ainsi juste calculé ses différents termes un à un. On met ensuite à jour le terme de `Q` correspondant via la ligne de code suivante :

```
Q(init_state, future_state) = q_actuel + alpha*(valeur_apprise - q_actuel);
```

- *Vérifier si l'état est final* Pour vérifier cela, il suffit de regarder si la récompense de l'action testée est de `100` (valeur proposée pour les actions vers les états finaux dans notre exemple).
- *Mettre à jour l'état courant* On finit en stockant dans `init_state` la valeur du nouvel état `future_state`.

Tous ces éléments se retrouvent alors dans le code suivant pour le déroulé de l'épisode :

```
while(~final_state)
    % Recherche des potentiels états suivants.
    recompense_states = find(R(init_state,:)+1);

    % Sélection d'un état suivant au hasard.
    future_state = recompense_states(randi([1 size(recompense_states,2)]));

    % Mise à jour de la matrice action-valeur.
    q_actuel = Q(init_state, future_state);                    % La valeur actuelle
    recompense = R(init_state, future_state);                  % La récompense de la
        transition
    val_opt_estimee = max(Q(future_state,:));                  % Valeur optimale estimée
    valeur_apprise = recompense + gamma*val_opt_estimee;       % Valeur apprise
    Q(init_state, future_state) = q_actuel + alpha*(valeur_apprise - q_actuel);

    % On regarde si l'on est dans un état final.
    final_state = (R(init_state, future_state) == 100);

    % On repard du nouvel état pour la suite.
    init_state = future_state;
end
```

Dans l'absolu, ce code peut suffir à lui seul à appliquer l'algorithme de Q-Learning. Nous avons cependant pour des raisons d'optimisation ajouter la partie pour arrêter ce dernier en cas de convergence que nous allons désormais vous présenter.

2.3.6 *Court-circuit si convergence*

Pour la mise en place de la convergence, l'idée est donc de vérifier à chaque étape si la différence entre l'ancienne et la nouvelle matrice `Q` est inférieure à une valeur `epsilon`. Comme nous l'avons déjà expliqué, l'idée est d'avoir cette condition qui soit vérifiée un nombre donnée de fois de manière successive. Ainsi, le code va comptabiliser les essais successifs positifs dans la variable *conv_count*. En cas d'échec au test cependant, ce compteur est remis à *0*.

Ces éléments vous sont proposés ci-dessous :

```
% On regarde s'il y a convergence (ie. ||prev_Q - Q||_2 < epsilon).
if(norm(previous_Q-Q) < epsilon)
    conv_count = conv_count + 1;
else
    % Condition non vérifiée, on repasse à 0 épisodes successifs.
    conv_count = 0;
end
```

Ensuite, il nous suffit de regarder si ce compteur dépasse une valeur fixée. Le cas échéant, on arrête de dérouler les épisodes et on considère que nous avons la matrice *Q* finale. Nous informons également l'utilisateur que l'algorithme a été arrêté avant d'avoir réalisé tous les épisode proposés.

Le code associé est alors le suivant :

```
% Si au moins nb_convergence épisodes sucessifs où
%                    ||prev_Q - Q||_2 < epsilon
% alors on conclue qu'il y a eu convergence et on arrête.
if(conv_count > nb_convergence)
    disp('Convergence de Q avant le nombre total d''étapes');
    disp(['Nombre d''étapes pour convergence : ' num2str(i)]);
    break;
end
```

2.3.7 *Normalisation*

Pour terminer, nous avons après toutes les étapes proposées obtenu une matrice *Q* qui a été entraînée. La fin de l'algorithme revient à normaliser cette matrice par sa valeur maximale. On obtient alors des cœfficients standardisés entre *0* et *100*.

Le code se conclut par un affichage du résultat à l'utilisateur.

```
% Normalisation de Q en fonction de sa valeur maximale
Q = 100*Q/max(max(Q));

% Affichage de la matrice obtenue
disp([char(10) 'Q final : ' char(10)]);
disp(Q)
```

2.4 Prédiction en utilisant *Q*

2.4.1 *Appeler l'algorithme*

Comme pour l'entraînement, il suffit d'appeler la fonction développer et que le fichier définissant cette dernière soit dans le même répertoire que le script appelant. Cette fonction prend comme paramètre la matrice d'action-valeur apprise *Q* ainsi qu'un état initial pour définir les actions *init_state*.

L'appel se fait donc simplement de la manière suivante :

```
optimize_behaviour(Q, 1)
```

2.4.2 *Structure du code*

Ce code se structure simplement en deux axes :

- *Initialisation* On initialise les valeurs générales de l'algorithme, ces dernières ressemblent fortement à celles observées en apprentissage.

- *Lecture de* `Q` L'idée étant de trouver un ensemble d'actions successives pour arriver à un état final de manière optimale.

Nous allons détailler ces deux parties dans la suite de cette section.

2.4.3 *Initialisation des grandeurs*

L'initialisation revient à considérer les deux variables globales de l'algorithme, à savoir ici :

- `seq` Ce tableau représente la suite des états à parcourir depuis notre état initial jusqu'à un état final. Il s'agira du retour de cette fonction sous la forme d'une séquence optimale.
- `final_state` Ce booléen est le même que pour l'apprentissage, ie. un booléen qui faudra `false` tant que l'état courant ne sera pas un état final.

Nous obtenons alors le code suivant :

```
%% Variables globales pour l'algorithme.
seq = [init_state];      % Pour stocker la séquence finale.
final_state = false;     % Savoir que l'on est dans un état final.

% Un petit peut d'affichage pour l'utilisateur.
disp('-------------------------------');
disp(['Départ depuis l''état ' num2str(init_state)]);
```

On remarquera aussi un affichage pour l'utilisateur afin de résumer la demande de ce dernier.

2.4.4 *Parcours de* `Q`

L'idée pour le parcours de `Q` afin d'obtenir une séquence optimale à partir d'un état initial est de parcourir cette matrice en sélectionnant à chaque étape l'action avec la plus grande valeur apprise.

L'algorithme revient ainsi juste en une boucle `while` qui pour chaque état sélectionne la meilleure action, met à jour son état et continue ainsi de suite jusqu'à arriver à un état final. Le seul point particulier ici est qu'il existe des situations ou plusieurs solutions optimales sont possibles. Nous avons choisi ici de choisir une solution au hasard et d'indiquer à l'utilisateur qu'une choix aléatoire a été réalisé.

Enfin, pour déterminer que nous sommes arrivés à un état final, nous regardons simplement si l'action de plus grande valeur nous ramène dans l'état ou nous sommes déjà.

Avec tous ces éléments, on en déduit alors le code suivant :

```
%% Boucle principale de lecture de la matrice Q.
while(~final_state)
    % Ensemble des prochains états optimaux à atteindre.
    optim_next_state_value = max(Q(init_state, :));
    optim_next_states = find(Q(init_state, :) == optim_next_state_value);

    % Si plusieurs états optimaux possibles, on prévient l'utilisateur.
    if(size(optim_next_states,2) > 1)
        disp(['--> Plusieurs chemins optimaux possibles à l''indice ' num2str(size(
            init_state,2)+1)]);
    end

    % On définit le prochain état optimal (s'il y a le choix).
    init_state = optim_next_states(randi([1 size(optim_next_states,2)]));

    % On regarde si on est à un état final. Pour cela, on regarde si
    % l'algo boucle sur lui même (la meilleure transition est vers
    % l'état courant.
    final_state = (init_state == seq(size(seq,2)));

    if(~final_state)
        seq = [seq init_state];
    end
end
```

On remarquera également qu'à chaque étape, on met à jour `seq` avec le nouvel état (si on ne le détecte pas comme final). Cette dernière remarque permet de voir que lorsque l'on arrive dans un état final, nous allons en fait faire une itération supplémentaire pour que ce dernier soit repéré comme tel (puisque détecter cette propriété se base sur la valeur précédente dans `seq`).

Une dernière ligne de code non présentée ici affiche à l'utilisateur le résultat de l'optimisation.

3 • Résultats obtenus

3.1 La situation test utilisée

3.1.1 *Aspects matériels du problème*

Nous avons choisi de reprendre la situation proposée dans le cours. Cette situation correspond en une maison de cinq pièces dans laquelle l'agent va vouloir optimiser sa sortie vers l'extérieur. Le plan proposé pour cette maison est celui de la Figure 1

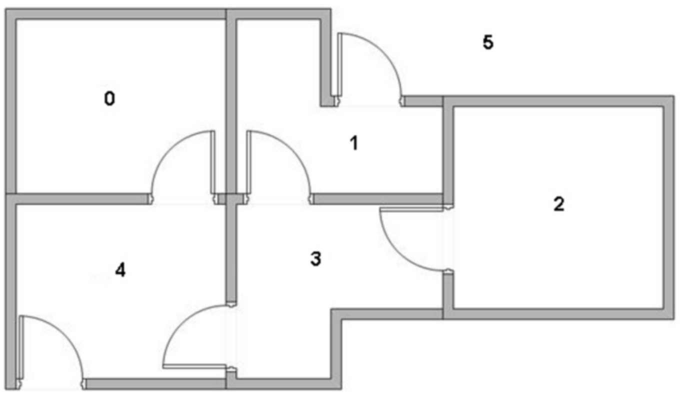
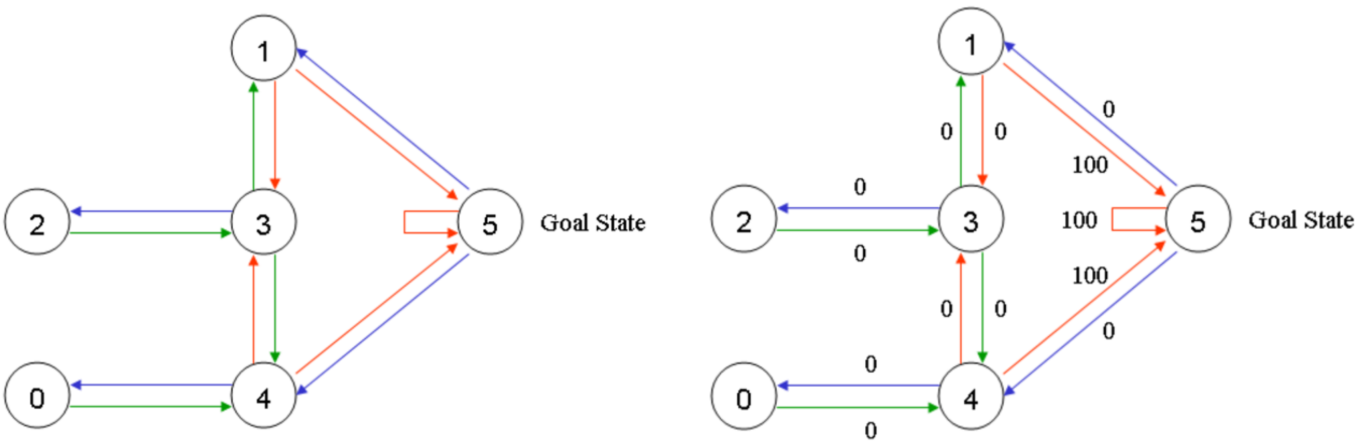


FIGURE 1 • Situation pour notre exemple (d'après J. MC CULLOCK)

Nous pouvons transformer cette maison en graphe, où les flèches représentent les transitions autorisées dans cette maison. L'état final (notre objectif) est alors indiqué par *Goal State* dans la Figure 2a. Nous pouvons de plus associer des récompenses aux différentes transitions autorisées, comme cela est présenté à la Figure 2b.



(a) Transformation de notre maison en forme de graphe **(b)** Ajout de récompenses sur les actions autorisées

FIGURE 2 • Modélisation de notre problème sous forme de graphe

L'objectif particulier proposé initialement était de sortir de la maison depuis l'état « 2 ».

Un point important est que sur ces figures, la numérotation commence à *0*, or les indices Matlab commencent eux *1*. Ainsi, les séquences qui seront produites par notre modèles seront toutes *shiftées de 1*. Ce point est à avoir en tête quand nous mentionnerons des indices plus tard dans ce rapport.

3.1.2 *Modélisation et matrice de récompense*

Pour terminer dans la définition de notre modèle, il s'agit de transformer le graphe de la Figure 2a sous forme d'une matrice R. Dans cette dernière, nous représenterons les transitions de la manière suivante :

- *Valeur* -1 Ceci signifie que la transition entre l'état (en ligne dans la matrice) vers un autre (ie. une action, en vertical dans la matrice) n'est pas autorisée.
- *Valeur* 0 Ceci signifie que cette transition est autorisée mais qu'elle ne conduit pas à un état final.
- *Valeur* 100 Ceci signifie que cette transition est autorisée et qu'elle conduit a un état final.

En utilisant cette modélisation, nous obtenons alors la matrice de récompenses R de la relation (10).

$$\mathrm{R} = \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \tag{10}$$

On remarque qu'il existe donc un seul état final (dernière colonne) et uniquement trois manières d'y parvenir.

3.2 Résultats de l'apprentissage

3.2.1 *Résultats bruts*

En utilisant l'algorithme présenté avec les valeurs proposées pour ses paramètres, nous obtenons alors les résultats ci-dessous :

```
Convergence de Q avant le nombre total d'étapes
Nombre d'étapes pour convergence : 1169

Q final :

         0         0         0         0   80.0000         0
         0         0         0   64.0000         0  100.0000
         0         0         0   64.0000         0         0
         0   80.0000   51.2000         0   80.0000         0
   64.0000         0         0   64.0000         0  100.0000
         0   79.9999         0         0   80.0000   99.9999
```

Nous retrouvons donc la matrice Q qui était proposée dans les supports de cours, si ce n'est le terme de valeur $51,20$ au lieu de 51 (problème d'arondi dans le cours ?). Cependant, toutes les autres valeurs sont conformes. On remarque également qu'avec les valeurs fournies pour *epsilon* et *nb_convergence* l'algorithme a uniquement eu besoin de 1169 itérations.

Ainsi, la convergence vers la valeur attendue est assez rapide (le temps de calcul est presque indolore sur la machine) pour ce cas jouet.

3.2.2 *Évolution de la valeur de* Q

Pour regarder le processus de mise à jour de Q au fil des épisodes, il est possible d'ajouter quelques éléments aux codes précédemment vus pour afficher les états intermédiaire de cette matrice, ainsi que les actions testées.

Initialisation À l'issue de l'initialisation, la matrice Q est simplement la matrice vide.

```
     0       0       0       0       0       0
     0       0       0       0       0       0
     0       0       0       0       0       0
     0       0       0       0       0       0
     0       0       0       0       0       0
     0       0       0       0       0       0
```

Première itération Pour cette première itération, le tirage aléatoire a fait que nous sommes parti de l'état *1*. À partir de cet état, seule la transition vers l'état *5* était possible. Puis, depui l'état *5*, trois transitions étaient possibles, vers *1*; *4* ou encore *6*. Le choix au hasard nous a ici renvoyé en *1*. On remarque que ce « yo-yo » a été exécuté plusieurs fois avant de partir vers *4* puis vers *6*.

On remarque ainsi que le caractère aléatoire du choix fait que rien n'empêche l'algorithme de revenir en arrière dans la boucle `while` de l'itération.

Le fait qu'un seul cœfficient (celui final) ait été mis à jour vient du fait qu'initiallement tous les cœfficient de *Q* sont nuls, ainsi dans la relation (9), seul le terme R_t peut donner une valeur non nulle.

```
Liste des transitions :
(1,5) (5,1) (1,5) (5,1) (1,5) (5,4) (4,5) (5,6)

-------------------------------
     0       0       0       0       0       0
     0       0       0       0       0       0
     0       0       0       0       0       0
     0       0       0       0       0       0
     0       0       0       0       0     100
     0       0       0       0       0       0
```

Deuxième itération Pour cette itération, l'état initial choisi aléatoire a été *3*. Le déroulement a été le même, on remarquera cependant un terme interne de la matrice qui a été mis à *80*, ce dernier correspond à la transition *(4,5)*. En effet, en mettant à jour ce cœfficient, nous avions déjà une valeur non nulle dans la cinquième ligne de *Q*, d'où le fait que la recherche du maximum à postériori n'ait pas donné un résultat nul (en fait nous avons ici $80 = 0,8 \times 100 = \gamma \times \mathrm{Q}(5,6)$).

```
Liste des transitions :
(3,4) (4,3) (3,4) (4,2) (2,4) (4,2) (2,4) (4,2) (2,4) (4,5) (5,6)
-------------------------------
     0       0       0       0       0       0
     0       0       0       0       0       0
     0       0       0       0       0       0
     0       0       0       0      80       0
     0       0       0       0       0     100
     0       0       0       0       0       0
```

Troisième itération Pour cette troisième itération qui part ici de l'état *2*, le tirage aléatoire nous emmène immédiatement à l'état final, il n'y a donc pas de commentaire particulier à faire. En effet, la mise à jour de *Q* est alors immédiate.

```
Liste des transitions :
(2,6)
-------------------------------
     0       0       0       0       0       0
     0       0       0       0       0     100
     0       0       0       0       0       0
     0       0       0       0      80       0
     0       0       0       0       0     100
     0       0       0       0       0       0
```

Quatrième itération Pour le dernier coup de cet exemple pas à pas, le départ choisi aléatoirement est ici de *3*. On remarquera qu'ici aussi un certain nombre d'aller-retour ont été fait dû au choix aléatoire des états suivants. On remarquera également que la matrice Q a subi un nombre important de mises à jour. En cause, le fait que plus de cœfficients étaient connus et que de moins en moins de lignes étaient toutes nulles. Ainsi, le terme de maximum à postériori dans (9) était peu souvent nul, d'où des mises à jour fréquentes.

```
Liste des transitions :
(3,4) (4,5) (5,4) (4,2) (2,4) (4,3) (3,4) (4,2) (2,4) (4,2) (2,4) (4,3) (3,4)
(4,3) (3,4) (4,5) (5,4) (4,3) (3,4) (4,3) (3,4) (4,2) (2,4) (4,5) (5,6)
Q final :
         0         0         0         0         0         0
         0         0         0   64.0000         0  100.0000
         0         0         0   64.0000         0         0
         0   80.0000   51.2000         0   80.0000         0
         0         0         0   64.0000         0  100.0000
         0         0         0         0         0         0
```

L'algorithme se déroulera ainsi jusqu'à ce que les différences ne soient plus notables sur un certain nombre d'étapes successives comme nous l'avons expliqué.

3.2.3 *Valeur de l'erreur avant convergence*

Le dernier point intéressant à regarder sur cet apprentissage est l'évolution de Q_2 ou encore de $\|Q_{t+1} - Q_t\|_2$. Nous avons ainsi ajouté quelques lignes de code pour enregistrer les valeurs au fur et à mesure et afficher les graphiques d'évolution in fine.

À noter que ces modifications qui n'avaient pas été montrées dans la présentation du code ont été conservées dans le code qui vous a été rendu.

Nous obtenons alors avec les paramètres proposés un peu plus tôt les résultats de la Figure 3.

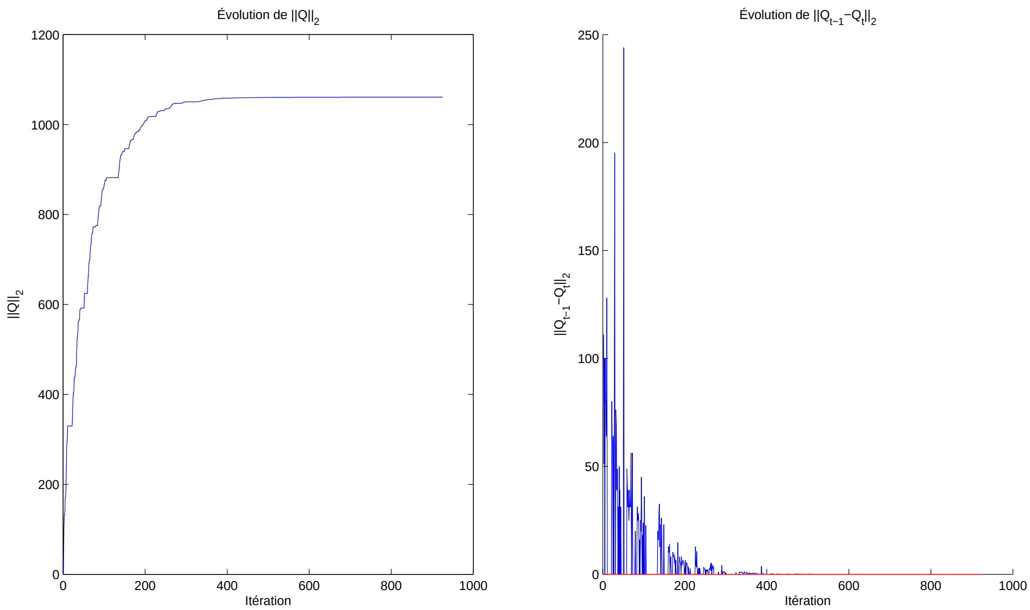


FIGURE 3 • Évolution de la norme de la différence de Q entre deux étapes et de la norme de Q

Sur la Figure 3, la ligne rouge représente la valeur de ε retenue. On remarque que le système nous permet bien de détecter un plateau de $\|Q\|_2$ (ce qui n'est pas suffisant pour dire qu'il y a convergence !), mais aussi le moment où les différences entre les Q d'itérations différentes deviennent presque nulles. On remarquera que le plateau semble en fait être atteint aux alentours de 300-400 itérations pour les deux graphes, les choix de nos paramètres sont donc peut-être un peu excessifs. Ceci sera discuté dans la section sur l'influence des paramètres.

Enfin, le graphe montre que s'il y a bien une enveloppe globale pour le graphe de $\|Q_{t+1} - Q_t\|_2$, des variations très fortes peuvent apparaître. Ceci provient du caractère aléatoire de nos choix qui fait que l'on peut relancer plusieurs fois d'affilées des tirages très proches qui ne changeront donc pas beaucoup Q avant d'en faire un autre radicalement différent qui changera significativement cette matrice.

3.3 Résultats pour la prédiction

Avec la matrice Q que nous avons appris, il est alors possible de chercher les séquences optimales pour sortir de notre maison. Nous avons alors obtenu les résultats énoncés ci-dessous en partant de chacune des pièces possibles :

```
Départ depuis l'état 1
Séquence optimale proposée :
     1     5     6

-------------------------------
Départ depuis l'état 2
Séquence optimale proposée :
     2     6

-------------------------------
Départ depuis l'état 3
--> Plusieurs chemins optimaux possibles à l'indice 2
Séquence optimale proposée :
     3     4     2     6

-------------------------------
Départ depuis l'état 4
--> Plusieurs chemins optimaux possibles à l'indice 2
Séquence optimale proposée :
     4     5     6

-------------------------------
Départ depuis l'état 5
Séquence optimale proposée :
     5     6

-------------------------------
Départ depuis l'état 6
Séquence optimale proposée :
     6
```

Si on compare ces parcours avec le graphe proposé à la Figure 2b, on observe bien que notre algorithme a d'une part trouvé tous les bons trajets optimaux et d'autre part précisé les deux trajets pour lesquels il y avait deux possibilités.

3.4 Influence des paramètres

Nous discuterons ici rapidement de l'influence des paramètres sur la qualité d'apprentissage. Les valeurs prises par défaut sont rappelées ci-dessous :

```
alpha = 1;
gamma = .8;
nb_iter = 2000;
nb_convergence = 75;
epsilon = 0.001;
```

Ainsi, les courbes de la Figure 3 pourront être prises en références par rapport à celles proposées ici.

3.4.1 *Influence de la vitesse d'apprentissage* α

Si nous prenons $\alpha = 0$, nous avons déjà expliqué que le modèle n'apprendrait rien et que nous nous retrouverions en sortie avec la matrice Q que nous avions en entrée (ie. une matrice nulle). Nous allons donc plutôt chercher les valeurs intermédiaires.

Pour $\alpha = 0,75$, nous obtenons à nouveau la bonne matrice, mais au prix d'un peu plus d'itération, puisque 1185 épisodes ont été nécessaires. On remarque également que le plateau mentionné sur la courbe donnant $\|Q\|_2$ est atteint plus tard, pour environ 600 itérations, comme le témoigne la Figure 4.

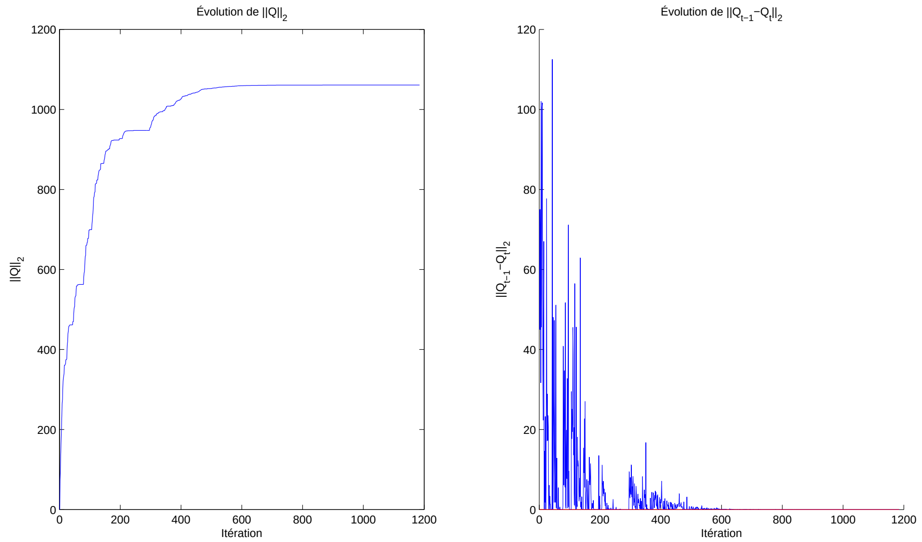


FIGURE 4 • Évolution des métriques pour le cas où $\alpha = 0,75$

En allant un peu plus loin à $\alpha = 0,5$, le résultat est quasiment parfait :

```
Q final :

         0         0         0         0   80.0000         0
         0         0         0   64.0000         0  100.0000
         0         0         0   64.0000         0         0
         0   80.0000   51.2000         0   80.0000         0
   64.0000         0         0   64.0000         0  100.0000
         0   79.9999         0         0   79.9999   99.9999
```

Cependant, ce dernier n'a pas été obtenu avant la fin des 2000 épisodes, l'algorithme ayant été coupé par le nombre maximum d'épisodes à réaliser. On peut cependant remarquer à la Figure 5 que nous nous situons bien sur le plateau attendu, signe que le non arrêt de l'algorithme est imputable à de faibles variations dans Q.

Nous voyons donc que diminuer α revient à rendre le modèle plus difficile à mettre à jour, ce qui dans notre cas recule la convergence du modèle. Nous pouvons (juste pour voir), tenter le cas où $\alpha = 0,1$. Pour cela, nous avons ré-haussé le nombre maximum d'épisodes pour voir à quel moment intervient la convergence. Cette dernière intervient alors au bout de 8680 itérations, cependant le résultat est beaucoup plus bruité que celui proposé dans le cours :

```
Convergence de Q avant le nombre total d'étapes
Nombre d'étapes pour convergence : 8680
Valeur de la ||Q_(t+1)-Q(t)||_2 à la convergence : 8.6896e-05

Q final :

         0         0         0         0   79.9999         0
         0         0         0   63.9997         0  100.0000
         0         0         0   63.9998         0         0
```

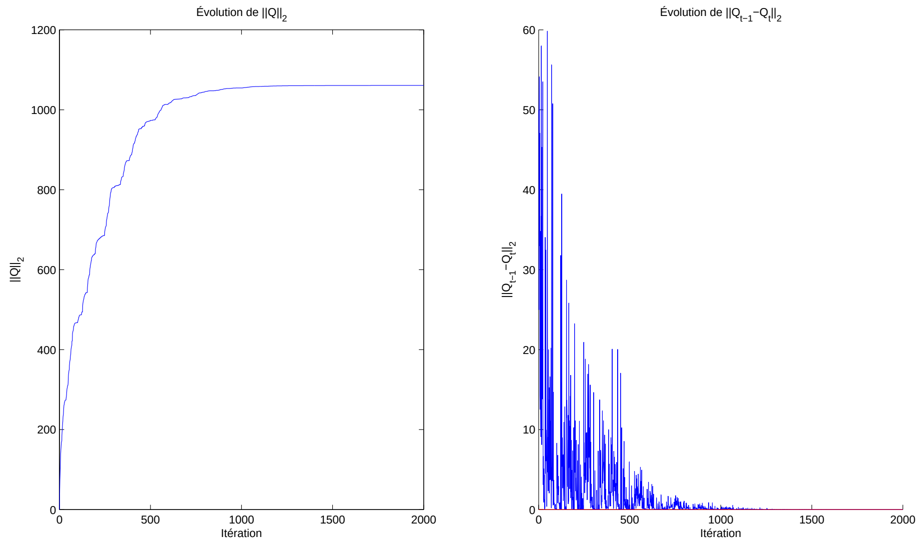


FIGURE 5 • Évolution des métriques pour le cas où $\alpha = 0,5$

```
        0    79.9999    51.1998          0    79.9998          0
  63.9998          0          0    63.9998          0   100.0000
        0    79.9983          0          0    79.9980    99.9982
```

On remarque alors à la Figure 6 que le plateau mentionné apparaît seulement au bout de plus de 6000 épisodes.

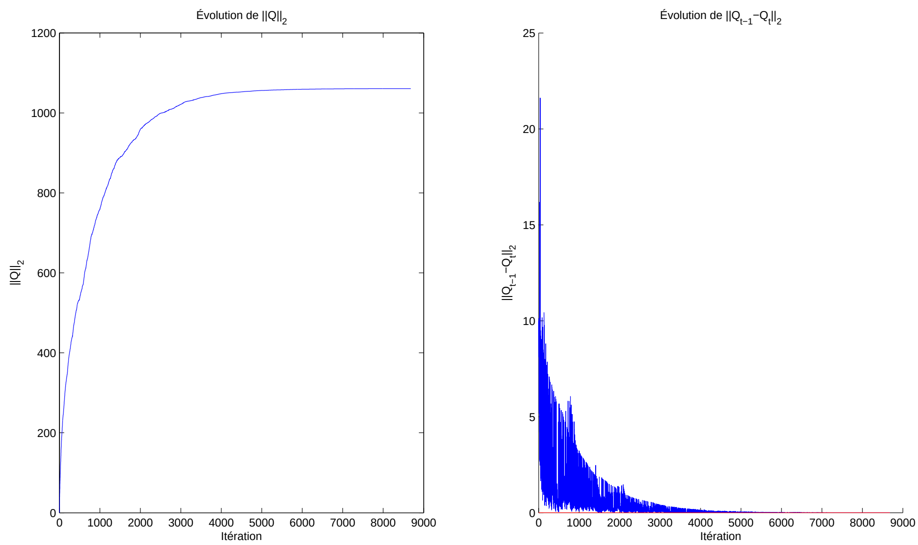


FIGURE 6 • Évolution des métriques pour le cas où $\alpha = 0,1$

En conclusion sur ce paramètre, pour le problème traité, il semble que les valeurs les plus réalistes soient situées aux alentours de 1. On remarque que plus la valeur de ce paramètre est faible, et plus le modèle met de temps à converger. De même, plus cette valeur est faible et plus le résultat final est bruité, au sens où les cœfficients ont des petits écarts aux valeurs proposées dans le cours. Cependant, prendre une valeur basse peut s'avérer utile dans des situations où le sur-apprentissage serait un danger.

3.4.2 *Influence du facteur d'actualisation* γ

Pour comprendre l'influence de ce paramètre, nous allons nous intéresser à ses cas extrêmes.

Si $\text{gamma} = 1$ Dans ce cas, cela signifie que nous donnons une importance maximale à la probabilité maximale de la relation (9). Ainsi, pour choisir une action cette probabilité jouera à jeu égal avec la récompense réelle de l'action. En conséquence, on risque de sur-valoriser les étapes intermédiaires pour générer des solutions finales. Ceci peut se vérifier en observant la matrice Q obtenue (50000 épisodes, convergence non terminée) :

```
Q final :

         0          0          0          0   100.0000          0
         0          0          0   100.0000          0   100.0000
         0          0          0   100.0000          0          0
         0   100.0000   100.0000          0   100.0000          0
  100.0000          0          0   100.0000          0   100.0000
         0    99.9328          0          0    99.9866    99.9597
```

Nous remarquons donc bien ce qui avait été annoncé, à savoir que tous les cœfficients correspondant à des transitions réelles (et qui sont donc utilisés pour aller vers un état final) convergent tous vers *100*. Ainsi, dans notre cas prendre un facteur d'actualisation de 100 ne va rien nous apprendre sur le modèle et fera que les choix réalisés grâce à ce modèle risquent de ne pas être optimaux.

Si nous regardons l'évolution des métriques à la Figure 7, on remarque que la norme de la matrice Q ne fait qu'augmenter (on rappelle que l'affichage numérique proposé est normalisé) et que l'évolution de la norme de la différence ne montre pas de tendance. Ceci signifie qu'à chaque étape les cœfficients sont mis à jour de manière importance, et ainsi la matrice ne pourra pas converger. Dans ce cas, nous aurons $\|Q\|_2 \to +\infty$.

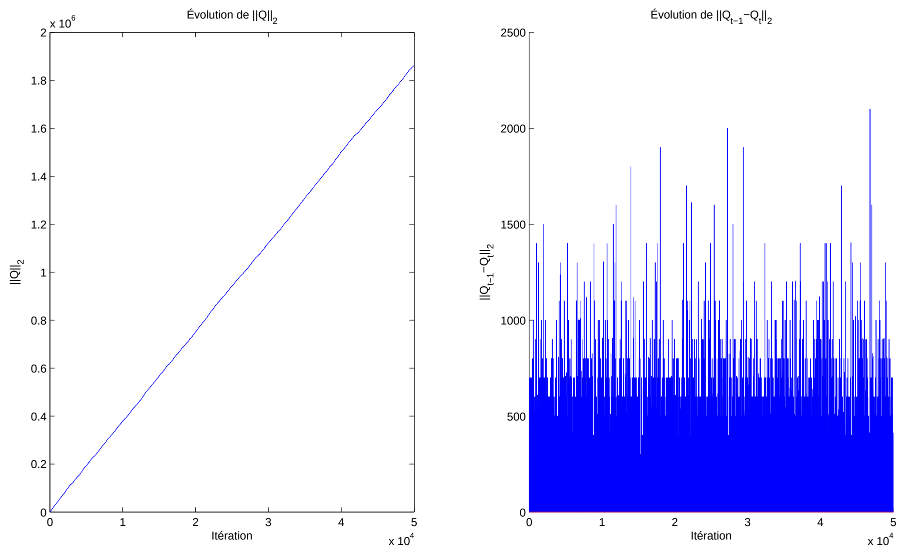


FIGURE 7 • Évolution des métriques pour le cas où $\gamma = 1$

Si $\gamma = 0$ Dans ce cas, la mise à jour se réalise uniquement sur la matrice de récompense. Ainsi, avec le choix par défaut qui est prix ($\alpha = 1$), la relation (9) revient à écrire :

$$Q_{t+1}(s_t, a_t) = R_{t+1}$$

Ainsi, l'algorithme va se contente de recopier le contenu de R et s'arrêter que ceci sera fait. Nous pouvons le vérifier avec le retour textuel de l'algorithme :

```
Convergence de Q avant le nombre total d'étapes
Nombre d'étapes pour convergence : 87
Valeur de la ||Q_(t+1)-Q(t)||_2 à la convergence : 0

Q final :
```

```

     0      0      0      0      0      0
     0      0      0      0      0    100
     0      0      0      0      0      0
     0      0      0      0      0      0
     0      0      0      0      0    100
     0      0      0      0      0    100
```

Nous voyons que ce dernier a effectivement juste recopié les récompenses pour les transitions finales (les autres étaient nulles). Avec cette valeur, l'algorithme aprend donc juste les « dernières » transitions avant les états finaux.

Si $\gamma = 0,25$ Pour comprendre un peu mieux l'influence de γ, nous avons également testé le cas intermédiaire où $\gamma = 0,25$. Nous obtenons alors le résultats écrit suivant :

```
Convergence de Q avant le nombre total d'étapes
Nombre d'étapes pour convergence : 267
Valeur de la ||Q_(t+1)-Q(t)||_2 à la convergence : 7.4579e-08

Q final :

         0         0         0         0   25.0000         0
         0         0         0    6.2500         0  100.0000
         0         0         0    6.2500         0         0
         0   25.0000    1.5625         0   25.0000         0
    6.2500         0         0    6.2500         0  100.0000
         0   25.0000         0         0   25.0000  100.0000
```

On remarque donc le même constat que précédemment, à savoir que pour une valeur faible de γ, les dernières transitions avant les états finaux sont bien apprises. Au contraire, celles les plus éloignées se retrouvent beaucoup moins bien détectées (par exemple le $1,5625$ correspond à un $51,2$ dans la solution « réelle »). Ainsi, la valeur de ce paramètre permet de contrôler la « profondeur de mémoire » permise par l'algorithme pour estimer son maximum espéré.

Il est également intéressant de noter que le résultat ici obtenu est considéré comme optimal par le système et qu'il ne s'agit donc pas d'un état intermédiaire tronqué !

3.4.3 *Influence de* `nb_convergence`

Sans surprise, si la valeur de ce paramètre est trop faible, un « mauvais » tirage aléatoire pourra faire stopper l'algorithme avant qu'il ne soit réellement arriver à convergence. Au contraire, une valeur élevée assurera que nous sommes arrivé à une vraie convergence, cependant les calculs supplémentaires pourraient s'avérer inutiles.

À titre illustratif, nous avons considéré la situation où *nb_convergence=5*. Dans ce cas, nous obtenons les résultats suivants pour la matrice Q :

```
Convergence de Q avant le nombre total d'étapes
Nombre d'étapes pour convergence : 13
Valeur de la ||Q_(t+1)-Q(t)||_2 à la convergence : 0

Q final :

         0         0         0         0   80.0000         0
         0         0         0   64.0000         0  100.0000
         0         0         0   64.0000         0         0
         0   51.2000   51.2000         0   80.0000         0
   64.0000         0         0   64.0000         0  100.0000
         0         0         0         0         0         0
```

Nous observons alors que la matrice est « à trous », il manque en effet des valeurs ! Ceci signifie que les tirages aléatoires ne sont pas tombés sur ces cœfficients manquants, mais sont tombés assez régulièrement sur les autres au moins cinq fois d'affilée, ce qui a arrêté l'algorithme un peu trop tôt. On remarque en particulier qu'une transition vers un été final n'a pas été tenté (cœfficient *6,6*). Cependant, un bon nombre de cœfficient déjà calculé est déjà à sa valeur finale, à cette itération.

3.4.4 Influence de `epsilon`

Comme pour le cœfficient précédent, nous pouvons avancer sans surprise que :

- *Si `epsilon` est trop faible* Le modèle sera certe très précis, mais le temps de convergence (et le nombre d'épisodes nécessaire) sera lui aussi beaucoup plus important !
- *Si `epsilon` est trop grand* Dans ce cas, on risque d'arrêter l'apprentissage avant une stabilisation complète de tous les cœfficients, ie. avant d'être réellement sur le plateau de la courbe donnant $\|Q\|_2$.

La question dans notre exemple peut être de savoir jusqu'où aller pour les valeurs élevées de `epsilon`. En prenant `epsilon=1`, nous obtenons la matrice Q suivante :

```
Convergence de Q avant le nombre total d'étapes
Nombre d'étapes pour convergence : 530
Valeur de la ||Q_(t+1)-Q(t)||_2 à la convergence : 0

Q final :

         0         0         0         0   80.0000         0
         0         0         0   64.0000         0  100.0000
         0         0         0   64.0000         0         0
         0   80.0000   51.2000         0   80.0000         0
   64.0000         0         0   64.0000         0  100.0000
         0   79.9643         0         0   79.9643   99.9553
```

Cette dernière est proche de la réalité, on se rend compte en fait que nous sommes arrivé en tout début de plateau comme en témoigne la Figure 8.

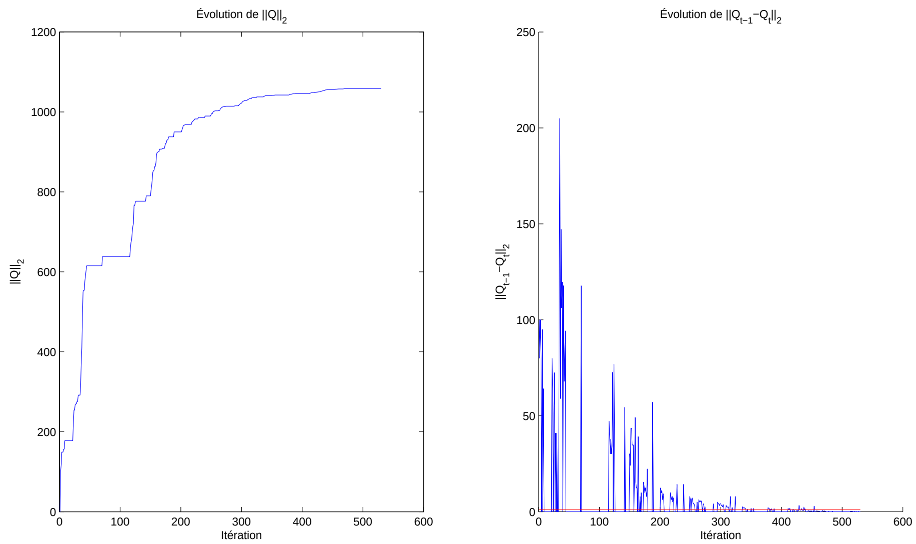


FIGURE 8 • Évolution des métriques pour $\varepsilon = 1$

Pour le cas où `epsilon=10`, les résultats sont encore plus bruités, comme en témoigne la valeur de Q :

```
Convergence de Q avant le nombre total d'étapes
Nombre d'étapes pour convergence : 251
Valeur de la ||Q_(t+1)-Q(t)||_2 à la convergence : 4.8216

Q final :

```

```
         0         0         0         0   78.9868         0
         0         0         0   63.1895         0   98.7335
         0         0         0   63.1895         0         0
         0   78.9868   50.5516         0   80.0000         0
   63.1895         0         0   63.1895         0  100.0000
         0   78.9868         0         0   78.9868   98.7335
```

Ceci est d'autant plus important que le problème fixé est ici simple. On remarque sur la Figure 9 que nous arrivons en fait à une situation encore plus limite sur le plateau.

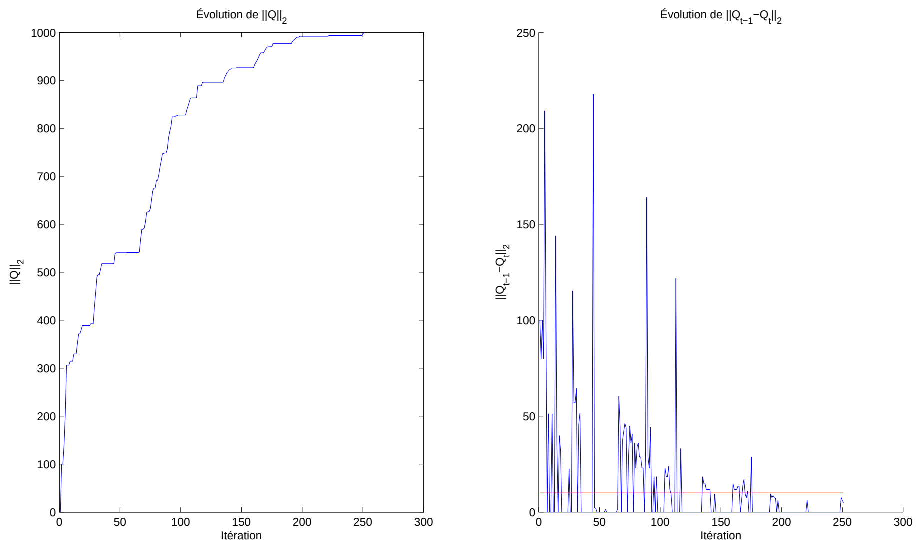


FIGURE 9 • Évolution des métriques pour $\varepsilon = 10$

Ainsi, il est recommandé de prendre des valeurs inférieures à 1 pour avoir des résultats réalistes par rapport aux cœfficients à trouver. Cependant, dans des phases exploratoires, il peut être utilse de prendre des valeurs un peu plus élevé pour dégager les principales tendances avant d'abaisser la valeur de ce cœfficient.

Conclusion

Ce travail a été pour nous l'occasion de nous initier au Q-Learning en implémentant nous-même un algorithme sur un exemple jouet. Les résultats obtenus sont conformes à ceux présentés en cours.

Nous avons également dans ce travail ajouté une partie de contrôle sur l'algorithme afin de vérifier la convergence et d'éviter des calculs inutiles. De même, cette partie nous permet de stoper l'algorithme si jamais la convergence est trop lente à arriver.

Les enseignements de ce travail se situent également dans la manière de paramétrer l'algorithme pour arriver à des résultats intéressant. Nous avons ainsi établi les règles suivantes :

- *Vitesse d'apprentissage* Plus cette dernière est faible, plus le modèle aura de mal à assimiler les nouvelles récompenses et sa convergence sera donc plus lente. Il semble donc intéressant de prendre une valeur basse si des problèmes de sur-apprentissage sont susceptibles de ce produire. Au contraire, dans les autres cas, prendre une valeur proche de 1 permet de se focaliser sur les récompenses (ce qui a été proposé ici).
- *Facteur d'actualisation* Nous avons vu que pour ce dernier, plus la valeur était faible, plus l'algorithme allait juste « copier » la matrice de récompense, ie. ne pas s'intéresser réellement à « optimiser la récompense future ». Ce paramètre permet donc de contrôler l'importance de l'avenir dans le calcul des cœfficients, c'est-à-dire l'importance des récompenses des états accessibles par les actions à mener depuis un état.
- *Facteur pour la convergence* Nous avons discuter des méthodes pour régler ces derniers afin d'avoir des valeurs fiables et peu bruiter. Les recommandations sont ainsi d'utiliser un `nb_convergence` élevée (supérieur à 50) et un `epsilon` faible. L'objectif de ces derniers étant de s'assurer d'arrêter l'algorithme sur le « plateau » de la courbe donnant l'évolution de $\|Q\|_2$ en fonction de l'épisode.

Si ce travail nous a permis de nous initier à cette méthode, des éléments supplémentaires pourraient être intéressants pour le poursuivre, comme :

- *Nouveaux exemples* Nous nous sommes ici concentré sur notre exemple simple, il aurait pu être intéressant de confronter notre algorithme à ces situations plus complexes.
- *Optimisation du code* Nous avons vu que le code proposé pouvait pendant son apprentissage faire des retour en arrière parfois nombreux, il pourraît donc être intéressant de vérouiller en partie ces derniers. De même, le code actuel peut entraîner (ce n'est pas le cas avec le graphe utilisé cependant) des boucles dans la génération de la séquence optimale, ce qu'il serait intéressant d'éviter.
- *Utilisation de réseaux neuronaux* Une autre méthode pour réaliser l'apprentissage est de passer par des réseaux de neurones, ce que nous n'avons malheureusement pas implémenté dans ce travail.