Polycopié de cours

Université Toulouse III

M1 - UPSSITECH

Introduction à l'apprentissage par renforcement

Author:
Emmanuelle Claeys

Date: February 6, 2024

February 6, 2024

Contents

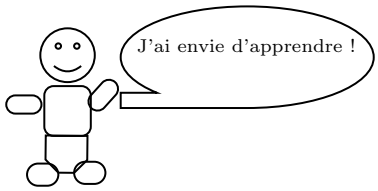
1 Introduction

Bienvenue à bord de votre voyage passionnant dans le monde du de l'apprentissage par renforcement, chers étudiant.e.s ! Attachez vos ceintures, car nous allons explorer les mystères fascinants de l'espace d'état, dévoiler les trésors cachés de la valeur d'état, et plonger dans les abysses du Q-learning.

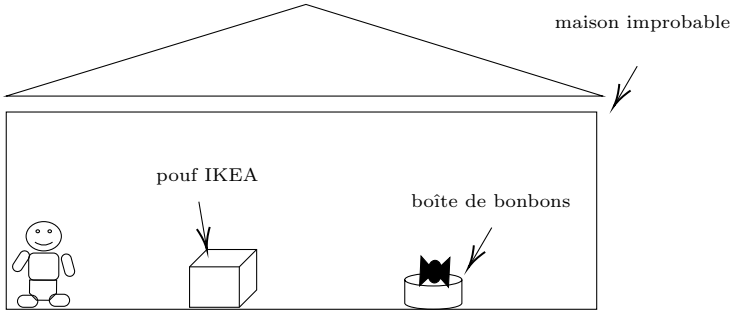
Commençons déjà par expliquer ce qu'est le Reinforcement Learning (qu'on appellera RL car sinon c'est un peu long...)

Imaginons le RL comme le processus d'apprentissage d'un bébé explorant le monde qui l'entoure. Dans cette analogie, le bébé est l'agent cherchant à comprendre son environnement, et sa maison joue le rôle de l'environnement, fournissant des retours sous forme de récompenses ou de punitions. Nous pouvons donc commencer à formaliser tous nos acteurs :

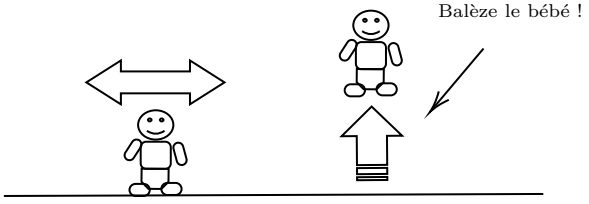
- L'Agent : Le bébé représente l'agent qui prend des actions pour interagir avec son environnement. Ces actions pourraient être aussi simples que toucher des objets ou se déplacer.



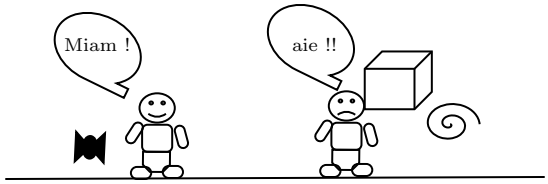
- L'Environnement : Les parents ou sa maison forment l'environnement dans lequel le bébé évolue. C'est là que se déroulent les expériences et les interactions.



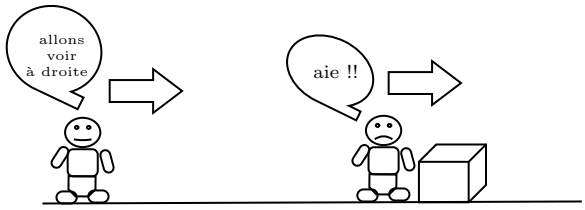
- Les Actions : Les actions du bébé, telles que se déplacer à gauche ou droite ou encore sauter d'un mètre (oui le bébé est très fort !), sont les équivalents des actions prises par un agent dans le RL. Chaque action est une tentative d'interaction avec l'environnement.



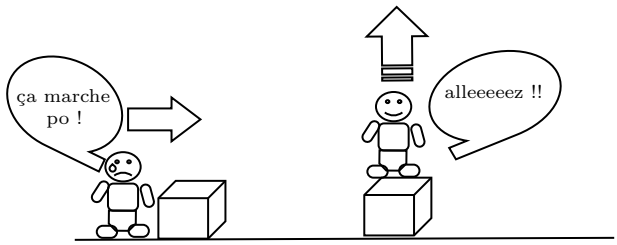
- Les Récompenses : Les réponses que peut obtenir le bébé, qu'elles soient positives (trouver des bonbons) ou négatives (se prendre un pouf Ikea dans la figure), sont les récompenses ou les punitions. Le bébé ajuste ses comportements en fonction de ces retours pour maximiser les expériences positives.



- L'Apprentissage: À travers ces interactions répétées, le bébé apprend quels comportements entraînent des récompenses et lesquels mènent à des résultats moins favorables. C'est un processus d'exploration et d'apprentissage constant.



- L'Exploration et l'Exploitation : Le bébé alterne entre reproduire des actions qui ont été précédemment récompensées (exploitation) et explorer de nouvelles actions (exploration). Cela ressemble à la manière dont un agent dans le RL doit équilibrer l'exploration pour découvrir de nouvelles stratégies et l'exploitation pour maximiser les récompenses connues.



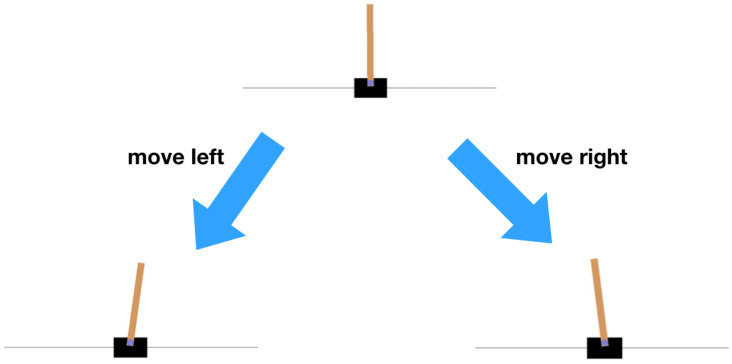
En utilisant cette image, le RL devient une métaphore intuitive, où l'agent (bébé) apprend à interagir avec son environnement (sa maison et ses meubles Ikea) pour maximiser les expériences positives (manger des bonbons). Les principes fondamentaux de l'apprentissage par renforcement, tels que l'exploration, l'exploitation et l'ajustement des actions en fonction des retours, trouvent un écho dans le développement naturel d'un enfant.

2 Modélisation de l'environnement

Bien entendu, notre cerveau est (encore) bien supérieur à une IA. Nous savons interagir dans notre environnement et nous apprenons très vite, soit par notre propre expérience, soit par l'expérience transmise par un tiers (nos parents, notre environnement social et culturel, ...). Pour une IA c'est plus compliqué ! L'environnement doit être complètement défini dans un programme informatique, un peu comme dans un jeu vidéo. D'ailleurs les premiers usages de l'intelligence artificielle étaient appliqués à des jeux de logiques justement parce que dans un jeu **les règles définissaient tous les scénarios possibles !** Chaque action dans un jeu donne un résultat attendu, faisant intervenir de l'aléatoire (si l'on doit, par exemple, lancer un dé) ou non (comme aux échecs). Si l'on souhaite utiliser un algo de RL pour résoudre un problème il faut commencer par **modéliser son environnement**. Commençons avec un exemple applicatif avec le problème **Care Pole** de la librairie Python `Gym`.

2.1 Les environnements benchmark de Gym

La librairie Gym, est une librairie Python développée par OpenAI, pour commencer à se familiariser avec la programmation RL. Gym fournit un ensemble d'environnements d'apprentissage standardisés, permettant aux chercheurs et aux développeurs de tester, de développer et de comparer efficacement différents algorithmes de RL. Présentons l'un des plus connu des environnements de Gym : **Cart Pole** !



Cet environnement correspond à la version du problème du chariot et de la perche décrite par Barto, Sutton et Anderson dans "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problem" (Éléments adaptatifs utilisant un réseau de neurones qui peuvent résoudre des problèmes difficiles de contrôle de l'apprentissage). Une perche est attachée par une articulation non actionnée à un chariot, qui se déplace sur une piste sans frottement. Le pendule est placé à la verticale sur le chariot et l'objectif est d'équilibrer le bâton en appliquant des forces dans les directions gauche et droite sur le chariot.

Espace d'action : L'action est représenté par une valeur qui peut prendre la valeur 0, ou 1, indiquant la direction de la force vers lequel le chariot est poussé. Plus précisément :

- 0 : on pousse le chariot vers la gauche
- 1 : on pousse le chariot vers la droite

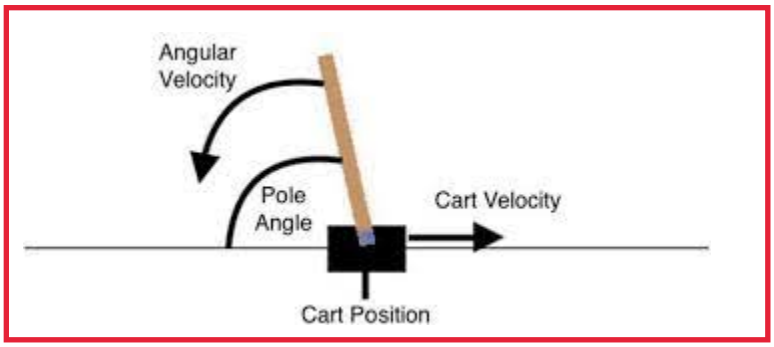
L'agent n'a pas la possibilité de contrôler la force avec laquelle on effectue un déplacement dans cet environnement (nous allons expliquer pourquoi plus tard).

Les récompenses : L'objectif est de maintenir la perche debout le plus longtemps possible, **une récompense de +1** est attribuée pour chaque pas de temps considéré, y compris le derniers pas de temps. Le seuil des récompenses est de 500 pour v1 ce qui veut dire qu'il y a 500 pas de temps (millisecondes) considérés. Plus le bâton est incliné, plus il est nécessaire d'enchaîner les actions dans le sens opposés. Pour que l'agent puisse connaître l'inclinaison du bâton, **il va falloir lui envoyer des infomations sur son environnement**.

L'espace d'observation : L'agent, avant chaque action, va pouvoir observer la position du bâton. Si l'humain par son oeil bionique arrive à identifier rapidement plusieurs observations sans avoir à les détailler, pour un programme informatique c'est un peu plus contraignant. Voici les 4 informations dont dispose l'agent **avant et après chaque action** :

- Cart Position : position du chariot sur l'axe horizontal. Le chariot peut se déplacer d'une position allant de -4.8 à 4.8 (sinon il tombe dans les limbes). En pratique la simulation s'arrête quand le chariot est dans [-2.4, 2.4] (histoire de sauver ce pauvre chariot).
- Pole Angle : L'angle de la perche peut être observé entre (-.418, .418) radians (ou ±24°), mais la simulation se termine si l'angle de la perche n'est pas compris dans l'intervalle (-.2095, .2095) (ou ±12°).
- Angular Velocity : représente la vitesse angulaire de la perche autour de son axe (calculée à partir de "Pole Angle"). La perche peut osciller vers la gauche ou la droite, et la vitesse angulaire mesure la rapidité avec laquelle elle se déplace dans l'une de ces directions.

- Cart Velocity : représente la vitesse du chariot, calculé à partir de "Angular Velocity" et "Pole Angle" Cette valeur est comprise dans $]\infty;\infty[$ bien que la perche tombe par terre bien avant d'atteindre de grandes valeurs. Plus la valeur absolue de Cart Velocity augmente, plus l'agent vas être déplacé dans le sens opposé.



En pratique il est possible de calculer mathématiquement la stratégie adoptée (pour les plus féru.e.s de math et de physique c'est ici) mais comme nous ne sommes ni mathématicien.ne ni physicien.ne nous allons utiliser un algorithme de RL (et tant pi pour les calculs !).

Pour ne pas faire tourner la simulation quand la perche est à terre nous allons considérer que la simulation est terminée lorsque l'un de ces évènements arrive :

- L'angle du bâton est supérieur à $\pm 12°$ (la perche est donc tombé du chariot). On est en **échec**.
- La position du chariot est supérieure à $\pm 2{,}4$ (le centre du chariot atteint le bord de l'écran). On est en **échec**.
- Le nombre de pas de temps est supérieure à 500 (200 pour v0). On est **succès**.

La notion de succès et d'échec est ici implicie pour nous. En réalité on va chercher **à maximiser le temps ou la perche tiens en équilibre** (rappelons qu'une perche en équilibre vaut un +1 pour chaque pas de temps considéré). Nous avons terminé de modéliser notre environnement, passons maintenant à la **résolution du problème par l'algorithme**.

.

En apprentissage par renforcement (RL), la modélisation de l'environnement revêt une importance cruciale. Elle détermine la manière dont l'agent perçoit et interagit avec son contexte. Une modélisation précise et représentative de l'état de l'environnement, des actions possibles et des récompenses permet non seulement une compréhension approfondie du problème, mais aussi une convergence plus efficace des algorithmes RL. En effet, une modélisation bien ajustée offre à l'agent la capacité de prendre des décisions informées, contribuant ainsi à la réussite des tâches d'optimisation et d'apprentissage.

Exercice : Modéliser l'environnement associé au célèbre jeu du Morpion. Lister les états (configurations possibles avant de faire une action) et les actions possibles d'un jeu (on considérera que vous jouez contre une IA). Vous choisirez une valeur de récompenses en fonction des états du jeu et listerez les conditions d'arrêt de la partie.

Les états pour la case 1 : $S_{11}=X$, $S_{12}=0$, $S_{13}=\emptyset$

Actions

Récompences : si on arrive à aligner 3 x 0 on gagne
si on essaie de placer x 0 sur un rond déjà rempli : -10
// // // de sortir de cadre -10

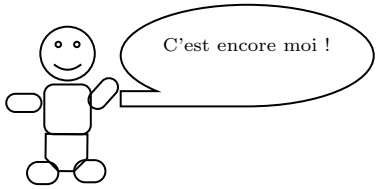
condit d'arrêt : si une personne gagne : aligner 3 x 0
si il n'y a plus d'état vide

3 Résolution par l'algorithme

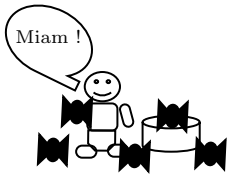
Avant de rentrer dans le détail de la résolution du problème, parlons un peu des récompenses à long terme...

3.1 Temporisation des récompenses

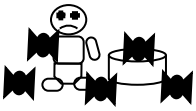
Rappelez vous notre cher bébé de tout à l'heure.



Notre bébé a trouvé le paquet de bonbon et commence à tout manger sans vergogne...



Toute personne ayant passé les fêtes avec une énorme boîte de chocolat sait qu'il n'est généralement pas bon pour l'estomac de manger la boîte en entier, mais le bébé ne le sait pas !!! Il continue donc à réaliser l'action "manger des bonbons" en mode infinie. Malheureusement pour notre bébé, au bout d'un moment il s'étouffe et meurt.



Le bébé aurait dû se douter que se **concentrer sur les récompenses à court terme n'est pas toujours une bonne stratégie à long terme** (dommage pour lui) ! Et bien c'est la même chose avec notre agent ! Nous devons **pondérer la valeur d'une action par rapport à leur gain à long terme** ! Et ça tombe bien car un certain Richard Ernest Bellman à trouvé comment faire.

3.2 L'équation de Bellman

Richard Ernest Bellman (né en 1920) réalise sa thèse sur les équations différentielles à Princeton après avoir été affecté au projet Manhattan entre 1944 et 1946 (cf l'excellent film Oppenheimer pour plus de détail sur ce projet). Richard se rend compte que l'utilisation d'équations différentielles permet de contrôler une trajectoire lorsqu'on on cherche à minimiser un certain coût (par exemple la consommation de carburant). Sans rentrer dans les détails, il modélise certaines équations qui seront aujourd'hui utilisées en RL et co-écrit plus de 600 articles de recherche et plus de 40 livres avant de mourir d'une crise cardiaque le 19 mars 1984 (plutôt prolifique donc).

Nous allons ici utiliser trois fonctions proposées par Bellman :

- Les Values Functions.
- Les Optimal Value Functions.
- L'équation de Bellman (bien qu'il en ai fait plus d'une !).

3.3 Notation

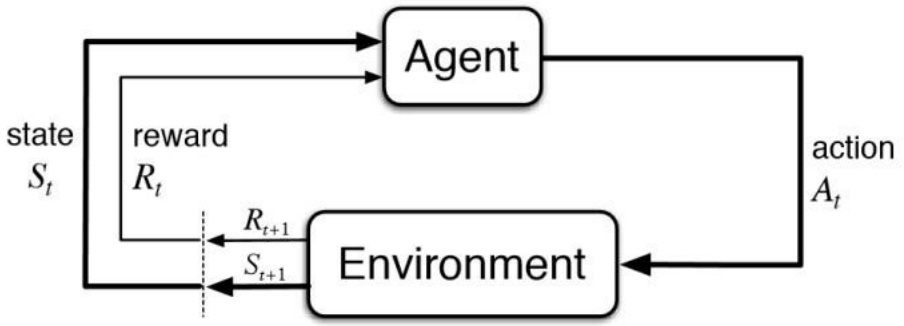
Pour nous plonger dans les équations de Bellman il va falloir utiliser des notations mathématiques. La **stratégie de choix** d'un agent est appelée **Policy** (notée symboliquement par π). Lorsqu'il est confronté à un état particulier (noté s) issu d'un ensemble d'états possibles (noté $\mathbb{S}$) il va choisir une action particulière (notée a) parmi un ensemble d'actions possibles (notés $\mathbb{A}$). Pour représenter ce choix, on va affecter des probabilités à chaque action. Par exemple si notre bébé peut faire 3 actions : [droite, saut, manger un bonbon], qu'il ne sais pas quelle action choisir entre droite et saut mais qu'il ne peut pas manger de bonbon (parce que pas de paquet présent) on pourrait obtenir un vecteur [0.5;0.5;0] ce qui veut dire que notre bébé à une chance sur deux de faire l'action "droite" et "saut" mais aucune chance de faire l'action "manger des bonbons". Ainsi, on peut formaliser une fonction représentant la **policy** comme une fonction qui **prend en entrée un état s et une action a** et qui **renvoi une probabilité de réaliser cette action a sachant l'état s**.

$$\pi : \begin{cases} \mathbb{S} \times \mathbb{A} & \rightarrow & [0,1] \\ (s,a) & \rightarrow & \pi(a|s) \end{cases}$$

Si vous n'avez pas encore oublié vos cours de statistique, vous vous rappellerez que la somme des probabilités d'évènements possibles doivent toujours faire 1 (même si certaines actions ont une probabilité d'être réalisées nulle). Ainsi les fonctions de policy d'un état s vérifient :

$$\forall s \in \mathbb{S}, \qquad \sum_{a \in \mathbb{A}} \pi(a|s) = 1$$

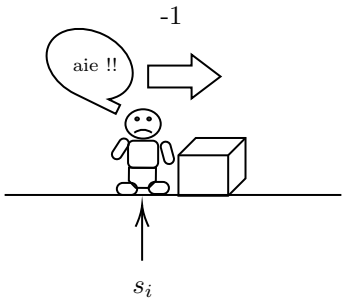
ce qui peut se lire comme "la somme des probabilités associées aux actions possibles a sachant qu'on se trouve dans un état spécifique s et égale à 1, et ce quelque soit s". Nous avons formalisé notre fonction de Policy, passons maintenant aux **états/récompenses dans un environnement non stationnaire**. La particularité du RL est que nous évoluons dans un environnement dynamique et temporisé. En effet, la récompense reçue au premier pas de temps n'est pas la même qu'après 10 actions (et donc 10 pas de temps). Il va falloir distinguer les états/actions/récompenses en fonction du pas de temps t considéré. Une autre particularité du RL est que les états/actions/récompenses vont dépendre des états/actions/récompenses précédents



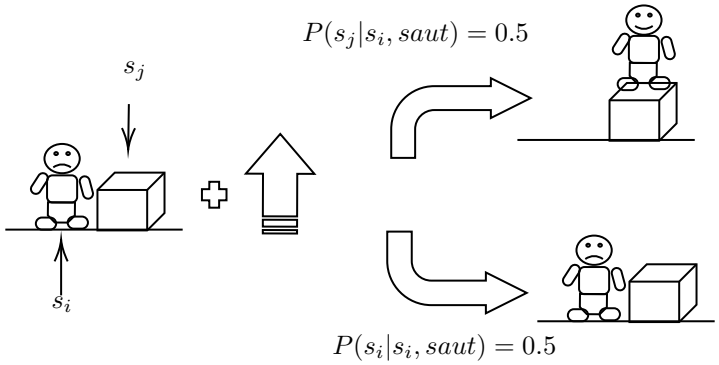
Le schéma ci-dessus est **Processus de Décision Markoviens (MDP)**. Un MDP est un modèle mathématique utilisé pour décrire des situations de prise de décision séquentielle où un agent interagit avec un environnement. Il est caractérisé par des états, des actions, des récompenses, et la probabilité de transition entre les états en fonction des actions prises. Les MDP fournissent un cadre formel pour modéliser et résoudre des problèmes de prise de décision dans des environnements incertains. L'équation de Bellman est étroitement liée aux MDP, car elle exprime la relation fondamentale entre la valeur d'un état ou d'une action et les valeurs des états ou actions futurs dans le contexte d'un processus de décision séquentiel. Ajoutons enfin deux notations supplémentaires

- **La fonction de récompense** $R(s,a)$. Elle renvoi une récompense pour un couple d'état/action donné. Par exemple réaliser l'action 'droite' depuis l'état 's_i' donne une récompense de -1 (le bébé se

prend le pouf Ikea dans la figure). Il est possible de bruiter les récompenses pour rajouter de l'aléatoire à l'environnement, mais retenez que **la fonction de récompense est codée par l'utilisateur lorsqu'il défini son environnement**.



- **La fonction de probabilités de transition** $P(s'|s,a)$) renvoi la probabilité d'accéder à l'état s' sachant que l'agent a réalisé une action a depuis un état s. Par exemple notre bébé, après avoir sauté, peut se trouver sur le pouf Ikea (il a réussi son saut) ou bien se retrouver à sa position initiale s_i (en ayant raté son saut comme un gros nul). Ainsi $P(s_j|s_i, saut) = 0.5$ (une chance sur deux de réussir son saut et de se retrouver sur le pouf), et $P(s_i|s_i, saut) = 0.5$ (le bébé rate son saut et se retrouve à l'état s_i). Pour les états différents de i et j, la probabilité d'arriver sur ces états est nulle :$P(s_{\neq i,j}|s_i, saut) = 0$ (sauf si le bébé peut se téléporter). **Il est donc possible pour un même couple d'état/actions d'arriver sur un état différent**. Si l'on souhaite rendre l'état s' déterministe (le bébé réussi toujours ses sauts), nous n'aurons qu'à fixer à $P(s_j|s_i, saut) = 1$. Comme pour la fonction de récompense, **la fonction de probabilités de transition est codée par l'utilisateur lorsqu'il défini son environnement**.



Passons justement à la **première équation de Bellman pour la fonction optimale de valeur d'un état (V)** !

3.4 Équation de Bellman dans le cas déterministe pour la fonction de valeur optimale d'un état (V)

Nous allons nous placer dans le cas stationnaire c'est-à-dire que **la récompense observée pour un couple état/action est systématiquement la même**. Les équations de Bellman ont en commun le fait de coupler la récompense immédiate d'un état avec les valeurs des états suivants :

$$V^{\pi}(s) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^{\pi}(s')$$

La valeur d'un état s pour une politique π est en fait la récompense obtenue immédiatement additionnée avec la valeur de l'état suivant (pondéré par un facteur de discount γ, qui donne plus ou moins d'importance aux récompenses futures). Ici V^{π} va dépendre de la policy choisie (par exemple toujours prendre la première action possible, ou bien la dernière). Cependant, nous allons chercher à trouver **la valeur optimale d'un**

état c'est à dire que **pour chaque état, la politique optimale correspond simplement à l'action ayant la plus grande valeur**. On pose alors l'équation d'optimalité de Bellman pour la fonction de valeur d'un état (appelée plus sobrement $V^*(s)$) :

$$V^*(s) = \max_a\{[(R(s,a) + \gamma \sum_{s'} P(s'|s,a)V(s')]\}$$

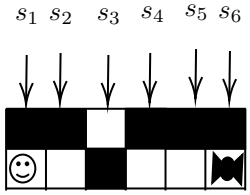
On retrouve ici :

- $R(s,a)$ qui renvoi une récompense lorsqu'on accède à un état s après une action a.
- $P(s'|s,a)$ qui renvoi la probabilité d'accéder à un état s' depuis un état s et une action a.
- La value fonction qui fait introduire la récursivité (c'est à dire que la valeur d'un état vaut sa récompense instantanée + ce qu'on peut espérer gagner **au maximum** dans les états futurs s' qu'on pourra explorer ensuite).

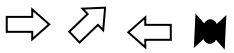
On notera également la présence d'une constante γ qui est un **hyperparamètre** (fixé par l'utilisateur avant de lancer le programme) et qui représente le **facteur de dévaluation** (à quel point on accorde de l'importance au futur). Nous y reviendrons mais considérer pour le moment qu'il est fixé 'au doigt mouillé' par l'utilisateur.

3.4.1 Mise en pratique de la fonction valeur (V)

Passons maintenant à la pratique. On reprend notre exemple du bébé en simplifiant un peu le problème :



Il y a 6 états possibles c'est-à-dire les cases qui peuvent être visitées par le bébé. Concernant les actions, le bébé peut se déplacer à droite, à gauche, sauter sur la diagonale droite et manger un bonbon (uniquement s'il y a un bonbon présent). Lorsque le bébé est en hauteur il redescend en allant à gauche ou a droite. Si le bébé rencontre un bonbon il n'a d'autre action possible que celle de le manger.



Si jamais le bébé cherche à se déplacer sur une zone noircie, il reçoit un malus de -1 et reste sur sa case. Si le bébé sort de la zone de jeu il reçoit un malus de -10 (la partie s'arrête) et si le bébé mange un bonbon il reçoit un bonus de 10 points (et la partie s'arrête). On peut commencer à modéliser la table d'espace d'état/action/reward ainsi

Cette table n'est pas encore complète. Il faut également renseigner la **fonction de policy** et la **fonction de transition**. Nous allons commencer par considérer que notre bébé **choisi aléatoirement** une action parmi celles qui possibles, et que les **transitions sont déterministes**

Cette table va nous donner toutes les informations nécessaires pour calculer la fonction valeur d'un état. Rappelons la formule associée à la valeur optimale d'état :

$$V^*(s) = \max_a\{[R(s,a) + \gamma \sum_{s'} P(s'|s,a)V^*(s')]\}$$

Voici le détail des calcule pour l'état s_1 (on va fixer $\gamma = 0.2$):

- On commence par lister les $R(a|s)$
  - $R(s_1, gauche) = -10$

État	Action	État résultat	Reward
s1	gauche	FIN	-10
	droit	s2	0
	saut	s1	-1
s2	gauche	s1	0
	droit	s2	-1
	saut	s3	0
s3	gauche	s2	0
	droit	s4	0
	saut	FIN	-10
s4	gauche	s4	-1
	droit	s5	0
	saut	s4	-1
s5	gauche	s5	0
	droit	s6	0
	saut	s5	-1
s6	manger	FIN	10

État s	Action a	$\pi(s,a)$	État résultat s'	$P(s'\|s,a)$	Reward $R(s\|a)$
s1	gauche	1/3	FIN	1	-10
	droit	1/3	s2	1	0
	saut	1/3	s1	1	-1
	manger	0			
s2	gauche	1/3	s1	1	0
	droit	1/3	s2	1	-1
	saut	1/3	s3	1	0
	manger	0			
s3	gauche	1/3	s2	1	0
	droit	1/3	s4	1	0
	saut	1/3	FIN	1	-10
	manger	0			
s4	gauche	1/3	s4	1	-1
	droit	1/3	s5	1	0
	saut	1/3	s4	1	-1
	manger	0			
s5	gauche	1/3	s4	1	0
	droit	1/3	s6	1	0
	saut	1/3	s5	1	-1
	manger	0			
s6	gauche	0			
	droit	0			
	saut	0			
	manger	1	FIN	1	10

- $R(s_1, droite) = 0$
- $R(s_1, saut) = -1$
- $R(s_1, manger) = 0$

Pour la deuxième partie de l'équation c'est plus subtile : on doit appeler récursivement[1] la fonction $V()$ sur les états suivants possibles (c'est à dire s_1 et s_2). Or nous n'avons pas encore calculé ces $V(s_1)$ et $V(s_2)$.

[1] une fonction récursive est une fonction qui s'appelle elle-même

Nous allons donc utiliser un tableau ou chaque valeur d'état va être initialisé à 0. **Au fur et à mesure que nous avançons dans les états, nous allons mettre à jour ce tableau.** Voyez plutôt :

$V^{\pi,*}(s_1)$	$V^{\pi,*}(s_2)$	$V^{\pi,*}(s_3)$	$V^{\pi,*}(s_4)$	$V^{\pi,*}(s_5)$	$V^{\pi,*}(s_6)$
0	0	0	0	0	0

Table 1: Valeur d'état à l'itération t=1

- On peut maintenant calculer $\gamma P(s'|s,a)V(s')$ pour les actions s_1 et s_2 :
  - $\gamma \times P(s_1|s_1, gauche)V(s_1) = 0.2 \times 1 \times 0 = 0$. Vous noterez que nous sommes obligés de spécifier un état résultat $V(s')$ lorsque le jeu se termine (nous choisirons de considérer la case de d'origine).
  - $\gamma \times P(s_1|s_1, saut)V(s_1) = 0.2 \times 1 \times 0 = 0$.
  - $\gamma \times P(s_2|s_1, droite)V(s_2) = 0.2 \times 1 \times 0 = 0$.
- On peut maintenant chercher le max entre :
  - $-10 + 0 = -10$ (gauche depuis s_1)
  - $0 + 0 = 0$ (droite depuis s_1)
  - $-1 + 0 = -1$ (saut depuis s_1)
- On choisi de mettre à jour $V(s_1)$ avec le max des trois valeurs ci-dessus (dans notre cas c'est 0 il n'y a pas de changement)

$V^{\pi,*}(s_1)$	$V^{\pi,*}(s_2)$	$V^{\pi,*}(s_3)$	$V^{\pi,*}(s_4)$	$V^{\pi,*}(s_5)$	$V^{\pi,*}(s_6)$
0	0	0	0	0	0

Table 2: Valeur d'état à l'itération 1 (état s_1 traités)

En fait la mise à jour de la valeur d'état est basée sur la plus grande valeur que peut espérer gagner l'agent. Tant que l'agent n'a pas **trouvé l'état but** les valeurs sont stabilisées à 0. Imaginons que nous ayons traité l'état s_2, s_3, s_4 et s_5. Intéressons-nous maintenant à l'état s_6 :

- $R(s_6, manger) = 10$

Ici il n'y a qu'une seule action possible : manger, comme le tour s'arrête on considère que le bébé est resté à son état s_6 et on met à jour :

$$V(s_6) = 10 + 0.2 \times 0$$

On obtient le tableau suivant

$V^{\pi,*}(s_1)$	$V^{\pi,*}(s_2)$	$V^{\pi,*}(s_3)$	$V^{\pi,*}(s_4)$	$V^{\pi,*}(s_5)$	$V^{\pi,*}(s_6)$
0	0	0	0	0	10

Table 3: Valeur d'état à l'itération 1 (tous les états traités)

Avons-nous terminé ? Et bien non ! Rappelez-vous que la value function **est une fonction récursive** ! Nous venons de changer les valeurs il faut donc **faire une nouvelle itération et recalculer toutes les valeurs d'état**. Pour l'état s_2, s_3, s_4 rien ne va changer mais regardons maintenant l'état s_5 :

- $R(s_5, gauche) = 0$
- $R(s_5, droite) = 0$
- $R(s_5, saut) = -1$

La partie droite de l'équation de Belleman va maintenant **utiliser la nouvelle valeur de** $V(s_6)$:

- $\gamma \times P(s_4|s_5, gauche)V(s_4) = 0.2 \times 1 \times 0 = 0.$
- $\gamma \times P(s_5|s_5, saut)V(s_5) = 0.2 \times 1 \times 0 = 0.$
- $\gamma \times P(s_6|s_5, droite)V(s_6) = 0.2 \times 1 \times 10 = 2.$

On choisi le max parmi

- $+0 = 0$ (gauche depuis s_5)
- $0 + 2 = 2$ (droite depuis s_5)
- $-1 + 0 = -1$ (saut depuis s_5)

On obtient le tableau suivant

$V^{\pi,*}(s_1)$	$V^{\pi,*}(s_2)$	$V^{\pi,*}(s_3)$	$V^{\pi,*}(s_4)$	$V^{\pi,*}(s_5)$	$V^{\pi,*}(s_6)$
0	0	0	0	2	10

Table 4: Valeur d'état à l'itération 2 (états s_1 à s_5 traités)

Passons maintenant au calcule de $V(s_6)$ pour la seconde itération :

$$V^{\pi,*}(s_6) = 1 \times [10 + 0.2 \times 1 \times 10] = 12$$

On obtient le tableau suivant

$V^{\pi,*}(s_1)$	$V^{\pi,*}(s_2)$	$V^{\pi,*}(s_3)$	$V^{\pi,*}(s_4)$	$V^{\pi,*}(s_5)$	$V^{\pi,*}(s_6)$
0	0	0	0	2	12

Table 5: Valeur d'état à l'itération 2 (tous les états traités)

Que c'est-il passé ? En fait **la valeur maximale associée à l'état but va ruisseler, au fur et à mesure des itérations, vers les états intermédiaires menant à cet étant but**. C'est la clé de l'équation de Belleman.

Exercice : Calculer la valeurs optimal de tous les états pour l'itération 3 et 4

Itération 3 : pour $s_1, s_2, s_3 = 0$
pour $s_4 = 0,4$
pour $s_5 = 2,4$
pour $s_6 = 12,4$

Itération 4 : pour $s_3 = 0,08$
$s_4 = 0,48$
$s_5 = 2,48$
$s_6 = 12,48$

L'algorithme **Value iteration** consiste à faire tourner cette procédure sur **plusieurs itérations jusqu'à stabilisation des valeurs**. Voici ce qu'on trouve comme valeurs après 50 itérations

et après 100 itérations

Vous constaterez que la différence est très faible entre les deux tableaux. En pratique **on arrête les itérations lorsque les valeurs évoluent toutes en dessous d'un seuil** δ. Ainsi, δ et un **hyper pramètre fixé par l'utilisateur** (généralement $\delta = 0.001$). Vous pouvez retrouver le code associé à cette

$V^{\pi,*}(s_1)$	$V^{\pi,*}(s_2)$	$V^{\pi,*}(s_3)$	$V^{\pi,*}(s_4)$	$V^{\pi,*}(s_5)$	$V^{\pi,*}(s_6)$
0.0039936	0.0199936	0.0999936	0.49999360	2.4999936	12.4999936

Table 6: Valeur d'état à l'itération 50 (tous les états traités)

$V^{\pi,*}(s_1)$	$V^{\pi,*}(s_2)$	$V^{\pi,*}(s_3)$	$V^{\pi,*}(s_4)$	$V^{\pi,*}(s_5)$	$V^{\pi,*}(s_6)$
0.003999	0.019999	0.09999	0.49999	2.49999	12.4999

Table 7: Valeur d'état à l'itération 100 (tous les états traités)

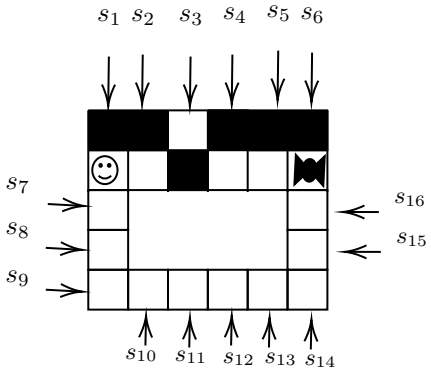
exercice ici.
L'algorithme associé au calcul des **fonctions de valeurs optimale** est appelée **Value iteration** et est formalisé ci-dessous :

Algorithme Value Iteration :

1. Initialiser les valeurs de tous les états à des valeurs arbitraires (par exemple, zéro).
2. Pour tous les états :
   (a) Considérer toutes les actions possibles et calculez la récompense attendue pour chaque action depuis l'état considéré.
   (b) Sélectionner l'action qui maximise la fonction de valeur et mettez à jour la valeur de l'état pour qu'elle corresponde à la valeur Max.
3. Répétez le processus jusqu'à ce que les valeurs des états convergent.

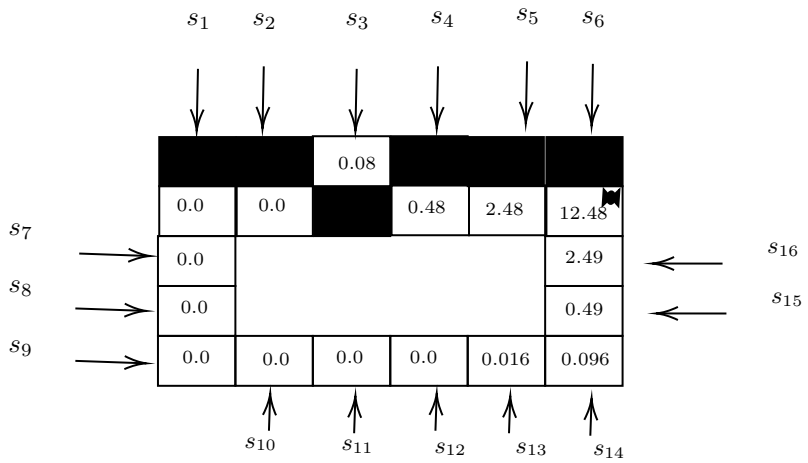
Une fois que les valeurs des états ont convergé, vous pouvez retrouver **le plus cours chemin pour retrouver l'état but**. En effet, on cherchera depuis l'état initial les état intermédiaires **à sélectionner suivant la plus grande valeur**.

Pour comprendre cette idée, changeons un peu notre problème... Nous donnons à notre bébé la possibilité de passer par le sous-sol (pourquoi pas ?)



On donne la possibilité à notre bébé de descendre et de remonter **uniquement quand c'est possible** :

Nous allons considérer que lorsque le bébé est dans le sous-sol, il ne peut que suivre le chemin (pas de chute mortelle dans les limbes pour lui). Par exemple, lorsque le bébé est dans l'état s_8 il ne peut que remonter ou descendre. Mettons notre code à jour ici et observons le résultat de l'algorithme Value Iteration après 50 itérations :



On remarque peut remarquer plusieurs choses importantes :

- Premièrement, si l'on part de l'état s_1, deux chemins sont possibles : le chemin $\{s_2, \dots s_6\}$ ou bien le chemin $\{s_7, \dots, s_{16}, s_6\}$. Pour trouver le chemin optimal on va partir de l'état s_1, et comparer pour l'état suivant **quel est le chemin qui maximise la valeur d'état**. S'il y a égalité, on comparera avec l'état d'après. Par exemple, pour le premier coup, on va arriver respectivement sur l'état s_2 et l'état s_7, ici il y a égalité. Pour le deuxième coup par contre, la valeur de s_3 **est supérieur à la valeur de** s_8. **L'agent partant de l'état** s_1 **va donc choisir le premier chemin qui est effectivement plus court**.

- Si en revanche on fait partir notre bébé depuis l'état s_{12}, il va avoir la possibilité d'arriver à l'état but depuis l'état $s_{11} \dots s_7, s_1 \dots s_6$ ou bien depuis l'état $s_{13} \dots s_{16}, s_6$. Pour choisir, l'agent va **comparer la valeur** des états voisins de s_{12} (ici s_{11} et s_{13}) et **choisir le second chemin**. **Cette stratégie basée sur la comparaison des valeurs d'état** va permettre à l'agent **se s'adapter quelque soit son point de départ**.

3.4.2 Policy iteration

Une autre méthode est l'**itération de la politique**, qui consiste à améliorer itérativement la politique en alternant les étapes d'évaluation et d'amélioration de la politique. Dans l'évaluation de la politique, la valeur de chaque état sous la politique actuelle est calculée en utilisant l'équation de Bellman. Dans l'amélioration de la politique, la politique actuelle est améliorée en sélectionnant l'action qui maximise la valeur de chaque état. Ce processus est répété jusqu'à ce que la politique converge vers la politique optimale. **L'algorithme** associé à **l'itération de la politique** est appelée **Policy iteration** et est formalisé ci-dessous :

Algorithme Policy Iteration :

1. Initialiser la politique à une politique arbitraire (par exemple, en sélectionnant une action aléatoire dans chaque état).
2. Répétez les étapes suivantes jusqu'à ce que la politique converge :
   (a) Effectuer l'évaluation de la politique :
      i. Initialiser les valeurs de tous les états à des valeurs arbitraires (par exemple, zéro).
      ii. Itérer sur tous les états du MDP :
         A. Pour chaque état, calculez la valeur de l'état sous la politique actuelle en utilisant l'équation de Bellman, qui exprime la valeur d'un état comme la récompense attendue pour l'action actuelle plus la valeur actualisée de l'état suivant.
   (b) Améliorer la politique :
      i. Pour chaque état, sélectionnez l'action qui maximise la valeur de l'état.

3. Une fois que la politique a convergé, la politique optimale a été trouvée.

Mettons cet algorithme en application sur notre premier exemple :

Nous allons maintenant utiliser deux tableaux. Ces tableaux vont **combiner les états et les actions** (Policy). Pour notre premier tableau, nous allons considérer la **valeur d'un état selon une action spécifique**. Pour cela, **introduisons la Q-value d'un couple état/action**:

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a)V(s')$$

Cette équation permet de donner la valeur d'un état sachant une action spécifique. Comme pour l'algorithme de Value Iteration, la **valeur de l'état** s' **sera** $\max_a Q(s',a)$.

Au début, notre agent ne connaît rien, les valeurs d'état sont initialisées à 0.

state	droite	saut	gauche	manger
s_1	0	0	0	X
s_2	0	0	0	X
s_3	0	0	0	X
s_4	0	0	0	X
s_5	0	0	0	X
s_6	X	X	X	0

Table 8: Q-value à l'initialisation

Pour le second tableau, nous mettrons **les probabilités** de choisir chacune des actions, qui vont **varier pour orienter l'agent vers les meilleurs actions**. **Le tableau de policy** sera donc **conditionné au tableau précédent**. Les probabilités $\pi(s)$ vont être conditionnées telle que $\pi(s) = \arg\max_a Q(s,a)$. Au départ, comme notre agent n'a **aucune connaissance**, **il choisira une action aléatoirement entre toutes les actions possibles** (s'il y a trois actions possibles, alors chacune aura une probabilités de $\frac{1}{3}$).

state	droite	saut	gauche	manger
s_1	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	X
s_2	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	X
s_3	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	X
s_4	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	X
s_5	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	X
s_6	X	X	X	1

Table 9: Policy à l'initialisation

Commençons l'itération avec l'état s_1. L'agent va **tirer aléatoirement une action avec les probabilités qui lui sont associées pour l'état** s_1. Tirons un action avec un dé imaginaire et obtenons "gauche". **Il faudra faire un tirage pour tous les états possibles.**. Nous affichons ici les actions obtenues pour tous les état :

$$\{s_1 : \text{gauche}; s_2 : \text{droite}; s_3 : \text{saut}; s_4 : \text{droite}; s_5 : \text{gauche}, s_6 : \text{manger}\}$$

Maintenant que les actions ont été tirées pour tous les états, on va **mettre à jour la valeur des états par rapport aux actions tirées**

Nous détaillons ici le calcule pour s_1 et s_2.

- Pour s_1 : on a réalisé l'action 'gauche', on obtient donc la récompense -10, on sort du terrain et il n'y a pas d'état résultat. Ainsi $Q(s_1, \text{gauche}) = -10 + 0.2 \times (0)$.
- Pour l'état s_2, on a réalisé l'action 'droite', on se prend un mur (aïe!) c'est à dire une récompenses de -1 et l'état résultat est l'état s_2 (valant initialement à 0) : $Q(s_2, \text{droite}) = -1 + 0.2 \times 0$.
- Pour l'état s_3, on sort du terrain en ayant une récompense de -10 et pas d'état suivant, donc $Q(s_3, \text{saut}) = -10$.

Exercice : Calculer les $Q(s, a)$ pour les couples d'état/action s_4, droite,s_5, gauche et s_6, manger

$Q(s_4, \text{droite}) = 0 + 0,2 \times 0 = 0$

$Q(s_5, \text{gauche}) = 0 + 0,2 \times 0 = 0$

$Q(s_6, \text{manger}) = 10 + 0,2 \times 1 \times 0 = 10$

On procède ainsi pour les états/actions suivants et obtenons

state	droite	saut	gauche	manger
s_1	0	0	-10	X
s_2	-1	0	0	X
s_3	0	-10	0	X
s_4	0	0	0	X
s_5	0	0	0	X
s_6	X	X	X	10

Table 10: Q-table à t=1

Nous allons maintenant **modifier les probabilité pour exclure les actions sous optimales pour chaque état**. On cas d'égalité, on aura des probabilités uniformes entre les meilleures actions. On obtient à la première itération :

state	droite	saut	gauche	manger
s_1	$\frac{1}{2}$	$\frac{1}{2}$	0	X
s_2	0	$\frac{1}{2}$	$\frac{1}{2}$	X
s_3	$\frac{1}{2}$	0	$\frac{1}{2}$	X
s_4	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	X
s_5	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	X
s_6	X	X	X	1

Table 11: Q-Policy à t=1

À partir des nouvelles probabilités, on procède à un nouveau tirage aléatoire d'action pour chaque état et obtenons ces résultats

$$\{s_1 : \text{droite}; s_2 : \text{saut}; s_3 : \text{gauche}; s_4 : \text{saut}; s_5 : \text{droite}, s_6 : \text{manger}\}$$

Les $Q(s, a)$ sont recalculées en conséquence :

- $Q(s_1, \text{droite}) = 0 + 0.2 \times 0$
- $Q(s_2, \text{saut}) = 0 + 0.2 \times 0$

- $Q(s_3, \text{gauche}) = 0 + 0.2 \times 0$ (on rappelle que depuis une hauteur, un mouvement droite/gauche fait descendre)
- $Q(s_4, \text{saut}) = -1 + 0.2 \times 0$
- $Q(s_5, \text{droit}) = 0 + 0.2 \times 10 = 2$
- $Q(s_6, \text{manger}) = 10 + 0.2 \times 10 = 12$

Mettons maintenant à jours notre Q-table et notre Q-value.

state	droite	saut	gauche	manger
s_1	0	0	-10	X
s_2	-1	0	0	X
s_3	0	-10	0	X
s_4	0	-1	0	X
s_5	2	0	0	X
s_6	X	X	X	12

Table 12: Q-table à t=2

state	droite	saut	gauche	manger
s_1	$\frac{1}{2}$	$\frac{1}{2}$	0	X
s_2	0	$\frac{1}{2}$	$\frac{1}{2}$	X
s_3	$\frac{1}{2}$	0	$\frac{1}{2}$	X
s_4	$\frac{1}{2}$	0	$\frac{1}{2}$	X
s_5	1	0	0	X
s_6	X	X	X	12

Table 13: Q-policy à t=2

Le critère d'arrêt est lorsque le **tableau de Q-policy n'évolue plus entre deux itarations** ET **que la liste des actions est strictement identiques**. En effet, en début d'apprentissage, la Q-policy peut être identique entre deux itérations lorsque beaucoup d'actions donne le même résultat (comme par exemple avancer d'une case sans obstacle et avoir une récompense de 0). Il faut donc obtenir entre deux itérations **la même liste d'actions tirées**.

On décide de faire tourner plusieurs itération notre algorithme de Policy Iteration. Vous retrouverez le code ici pour reproduire l'expérience. Au bout d'une dizaine d'itérations (cela varie légèrement à cause de l'aléatoire dans les tirages) on trouve les résultats suivants

state	droite	saut	gauche	manger
s_1	0.003	-1	-10	X
s_2	-1	0.019	0.0	X
s_3	0.099	-10	0	X
s_4	0.49	-1	0	X
s_5	2.499	-1	0.0	X
s_6	X	X	X	12.499

Table 14: Q-table à $t \approx 10$

state	droite	saut	gauche	manger
s_1	1	0	0	X
s_2	0	1	0	X
s_3	1	0	0	X
s_4	1	0	0	X
s_5	1	0	0	X
s_6	X	X	X	1

Table 15: Q-policy à $t \approx 10$

On prenant $\arg\max_a Q(s,a)$ **on retrouve les valeurs d'état optimale** $V^{\pi,*}(s)$!

3.5 Quelques remarques concernant Value et Policy Iteration

3.5.1 Différences entre Value et Policy Iteration

• Value Iteration (VI) et Policy Iteration (PI) sont deux algorithmes pour trouver la politique optimale, c'est à dire la séquence d'action à entreprendre dans chaque état pour maximiser la récompense cumulative attendue. Toutefois, **ils accèdent à tous les états sans considérer le chemin pour y accéder**. Cela pose des problèmes en pratiques mais nous y reviendrons...

• **La principale différence entre l'itération de valeur et l'itération de politique est la façon dont ils abordent le problème de la recherche de la politique optimale.**

- L'algorithme VI met à jour la valeur de chaque état en fonction de la récompense attendue pour chaque action possible. Ce processus est répété jusqu'à ce que les valeurs des états convergent, **et à la fin de la convergence**, la politique optimale peut être déduite.

- L'algorithme PI est également un algorithme itératif mais il alterne entre les étapes d'évaluations et d'améliorations de la politique.

VI va généralement converger plus rapide que l'itération des politiques, mais elle va aller chercher en mémoire le max entre toutes les actions possibles pour chaque état futurs s'. L'algorithme PI met plus de temps à converger (car il ne teste pas toutes les actions à chaque itération) mais est plus intéressant en terme de mémoire car il n'explore que selon la politique actuelle. Les deux algorithmes sont utilisés dans différentes situations, en fonction des caractéristiques de votre problème et de vos choix d'amélioration.

3.5.2 Conseil pratique l'implémentation

- Pour VI, à considérer comme critère d'arrêt un seuil de différence assez faible entre vos deux valeurs d'un même état.
- Pour PI, n'oublier pas de considérer une stationnarité vos séquences d'actions en plus de votre table de Q-policy
- Pour PI, il est généralement contraignant d'exclure des l'exploration de certains état dans un tableau (comme l'action 'manger' lorsqu'il n'y a pas de bonbon). Pour vous faciliter l'implémentation, n'hésiter pas à renseigner une **probabilité à 0 et une valeurs négative très importantes (par exemple -1000) dès l'initialisation de vos tableaux** . L'algorithme conservera une probabilité de 0 par la suite pour cette action.

Exercice : Coder et observer le résultat de l'algorithme Policy Iteration (PI) sur le problème du bébé et du sous sol référencé en page 13. Donner les Q-table et Q-policy obtenue.

3.6 Équation de Bellman dans le cas non déterministe

3.7 Aléatoire pour les récompenses

Corsons un peu le problème et imaginons que lorsque le bébé tombe sur un bonbon, il a une chance sur deux que l'emballage du bonbon soit vide (tristesse...). Dans ce cas là, la récompense obtenue pour l'action "manger un bonbon" de l'état s_6 va varier aléatoirement. Comment faire pour considérer cette **variabilité dans l'équation de Belleman** ? Nous n'allons plus utiliser la récompenses obtenue mais plutôt **la récompense moyenne** (ou plus formellement l'**espérance**, si vous vous souvenez de vos cours de statistiques). Pour la deuxième partie de l'équation de Belleman, puisque la valeur d'un état dépend maintenant de l'espérance de sa récompense, on obtient cette forme finale de l'espérance

$$V^{\pi,*}(s) = \max_a \{\mathbb{E}[R(s,a) + \gamma \sum_{s'} P(s'|s,a)V^{\pi,*}(s')]\}$$

et

$$Q(s,a) = \mathbb{E}[R(s,a) + \gamma \sum_{s'} P(s'|s,a)V(s')]$$

Le terme $\mathbb{E}$ veut dire "espérance" c'est à dire valeur moyenne attendue. Prenons un exemple avec notre bonbon qui a une chance sur deux d'être vide. L'espérance de la récompense, c'est à dire

$$\mathbb{E}[R(s_6, \text{manger})] = \frac{1}{2} \times 0 + \frac{1}{2} \times 10 = 5$$

Lorsqu'on mange un bonbon, en moyenne on obtient une récompense de 5. Si l'on commence à calculer $Q(s_6, \text{manger})$ à $t = 1$ ($V(s_6) = 0$) alors

$$Q(s_6, \text{manger}) = \frac{1}{2} \times 0 + \frac{1}{2} \times 5 + 0.2 \times 0 = 5$$

Pour $t = 2$:

$$Q(s_6, \text{manger}) = \frac{1}{2} \times 0 + \frac{1}{2} \times 5 + 0.2 \times 5 = 6$$

etc... Voici les Q-table et Q-value obtenue (le code est disponible **ici**)

state	droite	saut	gauche	manger
s_1	0.002	-1	-10	X
s_2	-1	0.010	0.0	X
s_3	0.05	-10	0	X
s_4	0.25	-1	0	X
s_5	1.25	-1	0.0	X
s_6	X	X	X	6.25

Table 16: Q-table à $t \approx 10$

state	droite	saut	gauche	manger
s_1	1	0	0	X
s_2	0	1	0	X
s_3	1	0	0	X
s_4	1	0	0	X
s_5	1	0	0	X
s_6	X	X	X	1

Table 17: Q-policy à $t \approx 10$

Si vous êtes allez voir le code vous aurez remarqué que j'ai fixé cette fois **un nombre d'itération minimum**. Cela s'explique par le fait que la **valeur moyenne de la récompense finale a diminuée** (de 10 à 5) et qu'il faut donc **plus d'itérations du fait que la récompense de l'état final va se 'propager' moins fortement sur les états postérieurs**.

Exercice : À partir du problème initial (bébé sans sous sol) faite tourner Value et Policy iteration tel que lorsque le bébé se cogne la tête, il a un malus de -1 avec une probabilité de $\frac{2}{3}$ et un malus de -2 avec une probabilité de $\frac{1}{3}$.

3.8 Aléatoire pour les transitions

Rappelez vous l'équation de Belleman dans le cas déterministe

$$V^{\pi,*}(s) = \max_a \{[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^{\pi,*}(s')]\}$$

lorsqu'une action pour un état s peut vous amener à deux états possibles (par exemple le bébé souhaite aller à droite mais il a une chance sur deux de ne pas réussir à se lever) alors vous devez calculer la **valeur d'état moyenne à partir des probabilités de transition**. Imaginons que nous soyons à l'itération $t = 2$ (après la première itération les valeurs d'état de 1 à 5 valent 0 et l'état s_6 vaut 10), on s'intéresse à s_6

- Pour Value Iteration :

$V^{\pi,*}(s_1)$	$V^{\pi,*}(s_2)$	$V^{\pi,*}(s_3)$	$V^{\pi,*}(s_4)$	$V^{\pi,*}(s_5)$	$V^{\pi,*}(s_6)$
0	0	0	0	0	10

Table 18: Valeur d'état à l'itération 2 (états s_1 à s_4 traités)

On va calculer $V^*(s_5)$ en choisissant le max parmi les récompenses et valeurs d'état suivant (ici c'est le couple "droite"/s_6), mais il faudra prendre en compte la probabilité de rester sur l'état s_5 après l'action "droite" donc

$$V^{\pi,*}(s_5) = R(s_5, droite) + \gamma(\frac{1}{2} \times V^*(s_5) + \frac{1}{2}V^*(s_6)) = 0 + 0.2(\frac{1}{2} \times 0 + \frac{1}{2} \times 10) = 2$$

La encore, il faudra plus d'itérations en raison d'une diminution des probabilités d'accéder à l'état récompenses en prenant l'action optimale.

Exercice : À partir du problème initial (bébé sans sous sol) faite tourner Value et Policy iteration tel que lorsque le bébé se déplace sur la droite, il se déplace avec une probabilité de $\frac{1}{2}$ et reste sur place avec une probabilité de $\frac{1}{2}$.

3.9 Conclusion sur Value et Policy iterations

Les deux algorithmes que nous avons présenté permettre de trouver la politique optimale à l'aide d'une programmation dynamique. Pour être plus technique, on dira qu'ils **résolvent le Makov Decision Process** (MDP) qui est la représentation du problème sous forme de graph orienté.

Nous reparlerons de cette représentation en MDP plus tard. Mais il y a quelques problèmes pratiques avec VI et PI :

- Ils supposent que nous avons la connaissance de toutes les récompenses associées aux état actions : $R(s, a)$ (pour les cas déterministe) ou $\mathbb{E}[R(s, a)]$ pour les cas non déterministes.
- Ils supposent que nous avons la connaissance des probabilités de transition d'un état s à un état s'.
- Ils supposent que nous pouvons accéder à n'importe quel état sans avoir testé la séquences d'action pour s'y rendre.

En fait, ces algorithmes trouvent toujours la politique optimale, mais ne sont pas représentatif d'une situation réelle ou seule une séquence d'action peut nous permettre d'accèder à certains état (à moins d'avoir inventé la téléportation). Ils sont en revanche utiles pour donner **l'objectif à atteindre**.

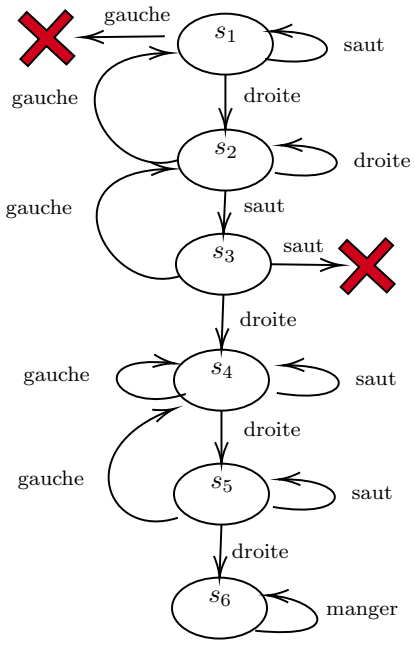
Nous allons maintenant présenter des **algorithmes d'apprentissage par renforcement** qui permettent de résoudre des problèmes **avec des contraintes d'accès et d'informations inconnues initiallement par l'agent**. Ces algorithmes vont eux aussi **utiliser l'équation de Belleman, les valeurs d'état et les Q-table/policy** mais avec **beaucoup plus de contraintes sur l'exploration des états**. Commençons avec un aglorithme général : le Q-learning.

4 Q-learning

Les deux algorithmes que nous avons vu précédemment ont l'avantage de **toujours trouver la politique optimale dans un nombre limités d'observations**. Cela s'explique notamment par le fait qu'**ils testent de façon méthodique tous les couples d'état/action possibles**. Cependant, ces algorithmes **considèrent que l'environnement et la dynamique du système sont connus**, c'est à dire qu'ils vont **évaluer tous les états sans se préocuper du chemin d'accès**. Cela est une contrainte forte car dans la réalité, on ne peut pas se déplacer à un endroit sans avoir pris le chemin pour y accéder. Par ailleurs, le fait de **tester tous les couples possibles** rend difficile leur application lorsque les actions de l'environnement sont continues (comme dans le problème de *car pole*. Nous allons donc utiliser un autre algorithme **qui ne nécessite aucun modèle initial : le Q-learning**.

4.1 Principe du Q-learning

Dans le Q-learning, l'agent exécute une action a en fonction de l'état s et d'une Q-table (comme d'habitude...). La différence avec les algorithmes précédents, est que lorsque l'agent perçoit le nouvel état s' et sa récompense r, et qu'il met à jour la fonction Q, **le nouvel état s' devient alors l'état courant**. L'agent continu son apprentissage **jusqu'à l'état terminal**. On relancera un certain nombre de fois l'algorithme **pour être sûr de converger vers la politique optimale**. Une autre différence importante est que pour choisir des actions, Q-learning aura **besoin d'une stratégie d'exploration-exploitation**. Pour comprendre le fonctionnement de Q-learning nous allons représenter notre **problème du bébé avec un graphe d'état**.



L'agent n'a cette fois **aucune connaissance en amont** (à part son état de départ et les actions qu'il peut réaliser pour chaque état). Comme pour policy iteration, nous allons utiliser une Q-table initialisée à partir de nos connaissances :

state	droite	saut	gauche	manger
s_1	0	0	0	X

Table 19: Q-table à $t = 0$

Comme l'agent va explorer des états en fonction d'une policy π la Q-value d'un couple état/action dépendra maintenant de π. Lorsqu'on considère un facteur d'apprentissage constant (nous y reviendrons),

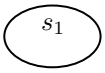
l'apprentissage de la Q-value vaut : :

$$\hat{Q}^{\pi}(s,a) = Q[s,a] + \alpha[R(s,a) + \gamma \max_{a} Q(s',a) - Q(s,a)]$$

Si **notre politique est efficace, alors $Q^{\pi}(s,a)$ devra converger vers la valeur maximale qu'on puisse obtenir, pour un couple état/action donné**, c'est-à-dire :

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

soit **la valeur qu'on aurait observé après avoir fait tourner policy ou value iteration**. Sauf que cette fois **l'agent ne peut accéder à un état que s'il a fait le chemin pour y arriver**. Reprenons notre apprentissage... $Q^{\pi}(s,a)$ mesure à quel point une action est favorable dans un état donné en tenant compte des récompenses immédiates et des récompenses futures potentielles. Comme nous l'avons vu dans l'algorithme **policy iteration**, cette fonction va dépendre **du choix de l'agent sur les actions suivantes**. Notre objectif est **que l'agent approche les valeurs $Q^{\pi}(s,a)$ vers les valeurs optimales** $Q^{*}(s,a)$ c'est-à-dire **ce que l'on peut espérer gagner au maximum pour un couple état/action** : $Q^{*}(s,a) = \max_{\pi} Q^{\pi}(s,a)$ Comme nous l'avons dit, **notre agent n'a aucune connaissance au démarrage de l'apprentissage :**



L'agent à la possibilité de faire 3 actions : droite, saut, gauche. Le choix d'une action dans le cas du Q-learning suit la procédure suivante :

1. Choisir un hyperparametre $\epsilon, \alpha, \delta \in [0;1]$ avant de lancer Q-learning.

Pour chaque action à réaliser, sachant un instant t et un état s_t donné :

1. Tirer aléatoirement une valeur $X_t \in [0;1]$
   - Si $X_t < \epsilon$ alors **choisir n'importe quelle action aléatoirement parmis les actions possibles**
   - Si $X_t \geq \epsilon$ alors consulter la Q-table, et **choisir l'action a_t qui maximise la Q-value pour le couple** $Q^{\pi}(s_{t+1}, a_{t+1})$. En cas d'égalité, on choisira aléatoirement parmis les actions maximales.
2. Jusqu'à : l'état but

En fait, la particularité de Q-learning est qu'il y a une stratégie **d'exploration/exploitation** à l'intérieur de l'algorithme. La stratégie que nous avons présenté ici (faisant intervenir de l'aléatoire et la comparaison par rapport à un hyperparamètre ϵ) est appelé **Espilon Greedy**. Il s'agit d'un algorithme à part, appartenant à la famille des **algorithmes de bandits** (mais nous ne présenterons pas cela dans ce cours). Revenons à notre agent. Voici les **étapes clefs** de l'algorithme Q-learning

- Dans l'état s_t, on effectue une action a_t telle que a_t est aléatoire avec une probabilité ε et $a_t \in \arg\max_a \hat{Q}_t(s_t, a)$ avec une probabilité $1 - \varepsilon$;
- Observer s_{t+1} et la récompense r_t ;
- Calculer $\delta_t = r_t + \gamma \max_a \hat{Q}_t(s_{t+1}, a) - \hat{Q}_t(s_t, a_t)$;
- Mettre à jour $\hat{Q}_{t+1}(s,a) = \hat{Q}_t(s,a) + \alpha_t(s,a)\delta_t \mathbb{1}\{s = s_t, a = a_t\}$.

$\alpha_t(s,a)$ est le facteur d'apprentissage, qui détermine à quel point la nouvelle information calculée surpassera l'ancienne. Si $\alpha_t(s,a) = 0$, l'agent n'apprend rien. A contrario, si $\alpha_t(s,a) = 1$, l'agent ignore toujours tout ce qu'il a appris jusqu'à présent et ne considérera que la dernière information.

Dans un environnement déterministe, la vitesse d'apprentissage $\alpha_t(s,a) = 1$ est optimale. Lorsque qu'on fait intervenir de l'aléatoire dans les récompenses, l'algorithme converge sous certaines conditions dépendant de la vitesse d'apprentissage. En pratique, souvent cette vitesse correspond à $\alpha_t(s,a) = 0.1$ pour toute la durée du processus

On décide de choisir $\alpha = 1$, $\gamma = 0.2$ et $\epsilon = 0.3$ ce qui veut dire que **30% de nos choix seront dédiés à l'exploration**.

- Au premier pas de temps $t = 1$ on tire $X_t = 0.5$, on doit donc **exploiter** l'action qui maximise la Q-value pour l'état s_1

Consultons la Q-table

state	droite	saut	gauche	manger
s_1	0	0	0	X

Table 20: Q-table à $t = 0$

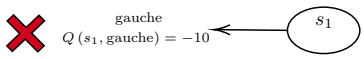
- Ici toutes les actions se valent (sauf l'action 'manger' qui est interdite). On tire donc aléatoirement l'une des trois actions possibles.
- On obtient l'action "gauche".
- On obtient une récompense de -10.
- Il n'y a pas d'état suivant donc $max_a\hat{Q}_t(s_{t+1}, a) = 0$
- On met à jour la valeur de $Q^\pi(s_1, \text{gauche}) = 0 + 1 \times [-10 + 0.2 \times 0 \text{ - } 0\ ]$.

state	droite	saut	gauche	manger
s_1	0	0	-10	X

Table 21: Q-table à $t = 1$

- On **recommence depuis l'état** s_1

Continuons un peu le développement de notre algorithme.... **Notre agent est "mort" et on repart du début**. On est toujours coincé sur l'état s_1 mais nous avons accumulé un peu de connaissance (sur l'action "gauche"), car nous concevront la Q-table intacte:

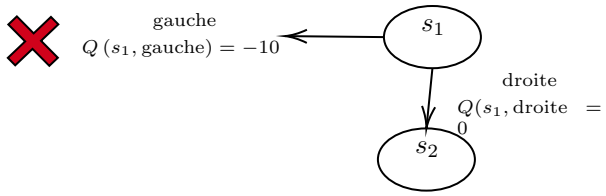


À nouveau

- À $t = 2$ on tire $X_t = 0.8$, **exploitation encore**, on tire aléatoirement entre "droite" et "saut" qui maximisent la Q-value pour l'état s_1
- On obtient l'action "droite"
- On obtient une récompense de 0
- On ne connaît pas l'état suivant donc $max_a\hat{Q}_t(s_{t+1}, a) = 0$
- On met à jour $Q(s_1, \text{droite}) = 0 + 1 \times [0 + 0.2 \times 0 - 0]$
- Cette fois notre agent est **toujours vivant (!)** et on développe l'état s_2

state	droite	saut	gauche	manger
s_1	0	0	-10	X
s_2	0	0	0	X

Table 22: Q-table à $t = 2$

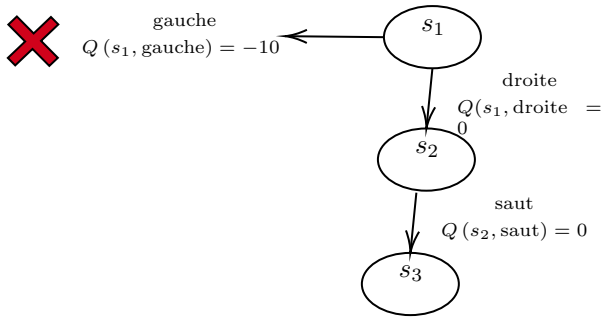


Pour l'état s_2 nous allons faire **exactement comme pour l'état** s_1.

- À $t = 3$ on tire $X_t = 0.1$, cette fois-ci on **explore**, c'est à dire qu'on **tire aléatoirement** entre toutes les actions possibles (dans notre cas cela ne change pas beaucoup, les Q-values des actions pour s_2 se valant toutes)
- On obtient l'action "saut"
- On obtient une récompense de 0
- On ne connaît pas l'état suivant donc $max_a \hat{Q}_t(s_{t+1}, a) = 0$
- On met à jour $Q(s_2, \text{saut}) = 0 + 1 \times [0 + 0.2 \times 0 - 0]$
- On développe l'état s_3

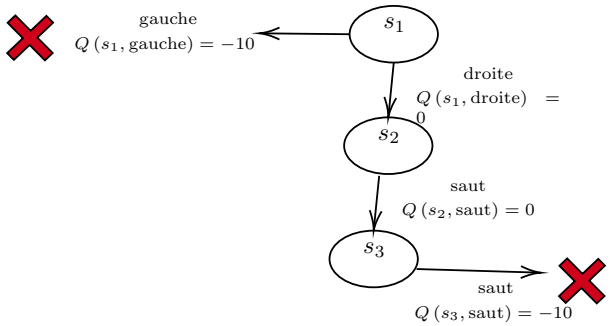
state	droite	saut	gauche	manger
s_1	0	0	-10	X
s_2	0	0	0	X
s_3	0	0	0	X

Table 23: Q-table à $t = 4$



Maintenant :

- À $t = 4$ on tire $X_t = 0.4$, on exploite.
- On obtient l'action "saut"
- On obtient une récompense de -10 et **notre agent meurt**
- L'agent meurt donc $max_a \hat{Q}_t(s_{t+1}, a) = 0$
- On met à jour $Q(s_3, \text{saut}) = -10 + 0.2 \times 0$
- **Puisque notre agent est mort, on repart de l'état initial et on recommence en développant l'état** s_1



Contrairement aux algorithmes précédent **l'agent ne peut développer que l'état qu'il a accédé depuis le pas de temps précédent**. On retourne donc à l'état s_1. Rassurez-vous, **on garde notre Q-table intacte** ! Elle va en effet nous permettre d'écarter les mauvaises décisions pour les prochains choix.

state	droite	saut	gauche	manger
s_1	0	0	-10	X
s_2	0	0	0	X
s_3	0	-10	0	X

Table 24: Q-table à $t = 5$

Exercice : À partir du problème initial (bébé sans sous sol) faite tourner l'algorithme Q-learning tel que $\epsilon = 0.3$ et $\gamma = 0.2$. Arrêter l'apprentissage au bout de 10,30 et 100 pas de temps. Comparer les résultats avec Policy et Value iterations.

5 Conclusion

Dans ce cours, nous avons exploré trois approches fondamentales en apprentissage par renforcement : l'itération de politique (policy iteration), l'itération de valeur (value iteration) et le Q-learning. Ces méthodes offrent des perspectives variées pour résoudre des problèmes complexes en autonomie, en prenant des décisions séquentielles dans des environnements dynamiques. L'itération de politique se concentre sur l'amélioration itérative de la politique, l'itération de valeur cherche à mettre à jour de manière itérative les valeurs associées à chaque état. Ces deux méthodes permettent de trouver la valeur optimale $Q^*(s, a)$. Le Q-learning se base sur l'apprentissage d'une fonction Q qui évalue les actions en développant les états au fur et à mesure. Chacune de ces approches présente ses propres avantages et limitations, offrant ainsi une diversité d'outils pour aborder les défis de l'apprentissage par renforcement. En comprenant ces concepts clés, nous sommes mieux équipés pour concevoir des agents intelligents capables de prendre des décisions optimales dans des environnements complexes et dynamiques.