UNIVERSITÉ DE SHERBROOKE

Rapport du Séminaire sur l'Apprentissage par renforcement et le Q-Learning

DUSSIER Alexys

Table des matières

Introduction

L'Apprentissage par Renforcement, ou Reinforcement Learning dans la langue de Shakespeare, ou encore RL, est une des branches du Machine Learning permettant à un agent d'apprendre à agir dans un environnement par une suite d'essaies et d'échecs.
Nous verrons ici comment cette technique peut être implémenté et nous décrirons son fonctionnement, notamment dans le cadre d'applications dans les jeux vidéos.

1 Le Principe

Le principe du RL est d'apprendre à un agent les actions optimales qu'il doit prendre dans un environnement donnée.
Ainsi, à chaque instant t, notre agent est face à une multitudes d'actions déterminées possibles. Il va en choisir une, ce qui aura pour effet d'affecter son environnement.
En retour, l'environnement enverra une récompense à notre agent ainsi que le nouvel état dans lequel il se trouvera, permettant ainsi à notre agent de choisir une nouvelle action.
On notera toutefois que les récompenses que reçoit l'agent sont très rares, dans le sens où elle seront très souvent nulles, le but étant de forcer notre agent à découvrir son environnement pour arriver à accomplir ce que l'on veut qu'il fasse par ses propres moyens. On ne veut en aucun cas lui dire comment le faire, juste qu'il le fasse. C'est lui qui déterminera le "comment".

2 Un problème d'introduction

Pour bien comprendre les algorithmes de RL, il est important de bien comprendre le problème dit des "K-bandits machines", ou des "K machines à sous".

Ce problème peut être décrit de la sorte :
Vous êtes dans une pièce, avec devant vous K machines à sous avec chacune un levier.
Vous disposez de h tours pour activer h fois ces machines. Il est possible d'activer plusieurs fois d'affilée la même machine.
A chaque fois que vous activez une machine i, celle-ci à une probabilité Pi de vous donner 1 pièce, ou sinon de ne rien vous donner. Vous ne connaissez aucune des probabilités Pi.
Comment décidez-vous de répartir vos h actions pour maximiser votre quantité d'argent ?
La question revient à demander, quelle est votre stratégie d'activation des leviers ?

2.1 Exploration contre Exploitation

On remarque tout de suite quelque chose :

- d'un coté il semble intuitif de n'activer que le levier dont a vu qu'il donnait le meilleur rendement
- de l'autre coté, comment connaître ce levier sans passer du temps à tous les tester un par un ?
- et de plus, comment décider que le levier que l'on a choisi est vraiment le meilleur ? On pourrait très bien avoir été victime d'un mauvais échantillonnage qui nous aurait fait choisir un levier non-optimal.

Ainsi, à la suite de ces réflexions, on est amené à définir les 2 concepts d'Exploration et d'Exploitation.
L'Exploration consiste à découvrir notre environnement pour acquérir de l'information et ainsi pouvoir faire de meilleurs choix dans le futur.
L'Exploitation consiste à exploiter les informations que l'on a acquises jusqu'à présent pour en tirer le maximum de profit.
La balance entre ces deux concepts est fondamental dans l'approche du Reinforcement Learning.

2.2 Optimiser

Quand on parle de stratégie, on cherche souvent à atteindre un but, ici on cherche à maximiser la quantité d'argent que l'on gagne.
Ce qui se traduit par la maximisation de la formule suivante : $E(\sum_{t=0}^{\infty} r_t)$
Le problème c'est que cette série ne converge pas et ne rentre donc pas dans le cadre de notre modèle mathématique, on utilise donc un petit trick de mathématicien : $E(\sum_{t=0}^{\infty} \gamma^t r_t)$
Le γ^t nous permet de nous assurer que notre série converge. De plus on peut l'interpréter comme le facteur de réduction. Il permet d'indiquer à quel point une récompense future est moins intéressante qu'une récompense instantanée. Vous préférez 1000 euros tout de suite, ou dans 10 ans ?
γ est compris entre 0 et 1, et il dépendra du contexte général, mais on prendra généralement une valeur de 0.9.

2.3 Stratégies formellement justifiées

Pour revenir sur notre problème des K-bandit machines, nous allons décrire succinctement 3 stratégies qui ont été prouvés comme convergentes vers le levier optimal. Seulement aucune de ces techniques n'est applicable en pratique.

2.3.1 Approche Récursive

Si l'on se place au temps h-1, on a déjà fait tous nos choix de levier à activer sauf 1, et l'on sait quel levier activer, c'est celui avec le meilleur rendement, et on saura aussi combien on espère gagner.
Comme cela est vrai quelque soit l'état au temps h-1, alors pour tous les états au temps h-2 on sait tous les états dans lesquels on pourrait aller ainsi que leurs gains espérés, on a donc toutes les informations nécessaires pour faire le meilleur choix.
En remonttant comme cela jusqu'au cas t = 0, on peut définir une stratégie optimale.
Cependant son temps de calcul est juste exponentiel et n'est donc pas applicable en pratique.

2.3.2 Indices d'Allocation de Gittins

Si l'on définit pour chaque états le tuple $(n_1, \omega_1, n_2, \omega_2, ...n_k, \omega_k)$ où n_i est le nombre de fois que l'on a activé la machine i et ω_i est le nombre de fois où l'on a été récompensé pour avoir activer la mahcine i.
Alors on peut utiliser les tables de Gittins qui pour un temps t, un temps maximum h, et le tuple $(n_1, \omega_1, n_2, \omega_2, ...n_k, \omega_k)$ et le facteur de réduction γ nous donne des incides associées à chaque levier i. Cette théorie nous assure que choisir la machine dont l'indice est le plus élevé est le meilleur choix. En prennant en compte la balance optimale entre Exploration et Exploitation.
Seulement cette technique n'est pas généralisable pour le problème d'après, qui est celui qui nous intéresse réellement.

2.3.3 Apprentissage avec automates

Il existe également une stratégie qui consiste à associer une probabilité P_i à chaque machine i d'être choisit.

A chaque fois que l'on choisit une machine, on met à jour les P_i de toutes les machines selon la loi :

- quand l'action i est un succès
  - $P_i = P_i + \alpha(1 - P_i)$
  - $P_j = P_j - \alpha P_j$ pour tous $j! = i$
- quand l'action i est un échec, on ne change rien

Le problème est que cette technique peut converger vers la mauvaise machine.
Heureusement on peut faire que cette possibilité soit aussi faible que possible en diminuant le paramètre α qui est en fait le taux d'apprentissage du modèle. Quand il vaut 1, on apprend aussi vite que possible, quand il vaut 0 on n'apprend pas.

2.4 Stratégies non-formellement justifiées

En pratique, on aura plutôt tendance à utiliser des stratégies qui ne sont pas assurées de converger vers le meilleur résultat mais qui ont au moins le mérite d'être facile à implémenter et de donner des résultats plutôt satisfaisant.

2.4.1 Greedy Stratégie

La Greedy stratégie, ou stratégie gourmande, consiste à chaque temps t de choisir la machine ayant le plus haut rendement.
Bien évidemment ce n'est pas une stratégie optimale à appliquer du début à la fin. Mais elle a beaucoup de sens si l'on veut l'appliquer après une longue phase d'exploration !

2.4.2 Random Stratégie

On choisit une machine au hasard avec une probabilité p, sinon on applique la Greedy Stratégie.
Cette technique est en pratique très performante, il s'agit d'ailleurs de celle que j'ai choisit d'implémenter pour mon exemple.
On choisira en général un paramètre p important au début, disons 0.9, que l'on feras diminuer au fur et à mesure que l'on acquiert de l'information jusqu'à être pratiquement nul.
On fera attention à ne jamais faire passer ce paramètre à 0 car notre modèle arrêterais de s'adapter dans ce cas-là.
Elle a cependant un défaut, dans le cas où l'on choisit une machine au hasard, on choisit avec égale probabilité une machine dont on sait que son espérance est très faible qu'une machine avec un fort potentiel.

2.4.3 Random Stratégie avec probabilités de Boltzmann

Pour pallier au problème de la stratégie précédente, on peut au lieu d'utiliser des probabilités uniformes utiliser des probabilités de Boltzmann qui auront tendance à favoriser les machines avec le plus d'espoir. Voici la formule $P(a) = (e^{ER(a)/T})/(\sum_{a' \in A} e^{ER(a')/T})$ où $ER(a)$ est la récompense espérée en prenant l'action a et T est la température et peut être vu comme à quel point on cherche à explorer les leviers qui ne présentent presque aucun intéret.

3 Généralisation au problème principal

Le problème avec le modèle précédent c'est que l'on a toujours notre récompense instantanément. Hors dans la réalité des jeux vidéos, on déclenche notre coup de poing en appuyant sur A à la frame t, mais l'on ne touche notre adversaire que quelques frames plus tard.
Il nous faut donc un modèle pour simuler cela.
On va utiliser les Markov Decision Processes(MDP).

Un MDP est constitué de :
— un ensemble d'états S
— un ensemble d'actions A
— une fonction de récompense R : S x A -> $\mathbb{R}$
— une fonction de transition T : S x A -> P(S)

On peut voir un MDP comme un ensemble de pièce où à l'intérieur de chacune on a un k-bandit machine différent, et où lorsque l'on active un levier on est téléporté dans une autre pièce suivant la fonction de transition.

3.1 Stratégie optimale connaissant le modèle

On définit la valeur V d'un état comme un gain potentiel que l'on pourrait avoir si l'on était dans celui-ci.
Ainsi, V^* est la gain optimal que l'on obtiendrait à partir de cet état si l'on suivait la meilleur stratégie π possible. D'où $V^*(s) = \underset{\pi}{max} E(\sum_{t=0}^{\infty} r_t)$

Comme la valeur d'un état est la valeur d'un état en prenant la stratégie π revient à la récompense obtenu en prenant le premier coup de la stratégie π plus la valeur de l'état dans lequel on va se trouver, on obtient : $V^*(s) = \underset{\pi}{max}(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^*(s'))$

Ainsi, on constate que le meilleur coup à faire n'est rien d'autre que l'argmax de ce maximum.
On vient donc de définir la meilleur stratégie $\pi^*(s) = \underset{\pi}{argmax}(R(s,a)+\gamma \sum_{s' \in S} T(s,a,s')V^*(s'))$

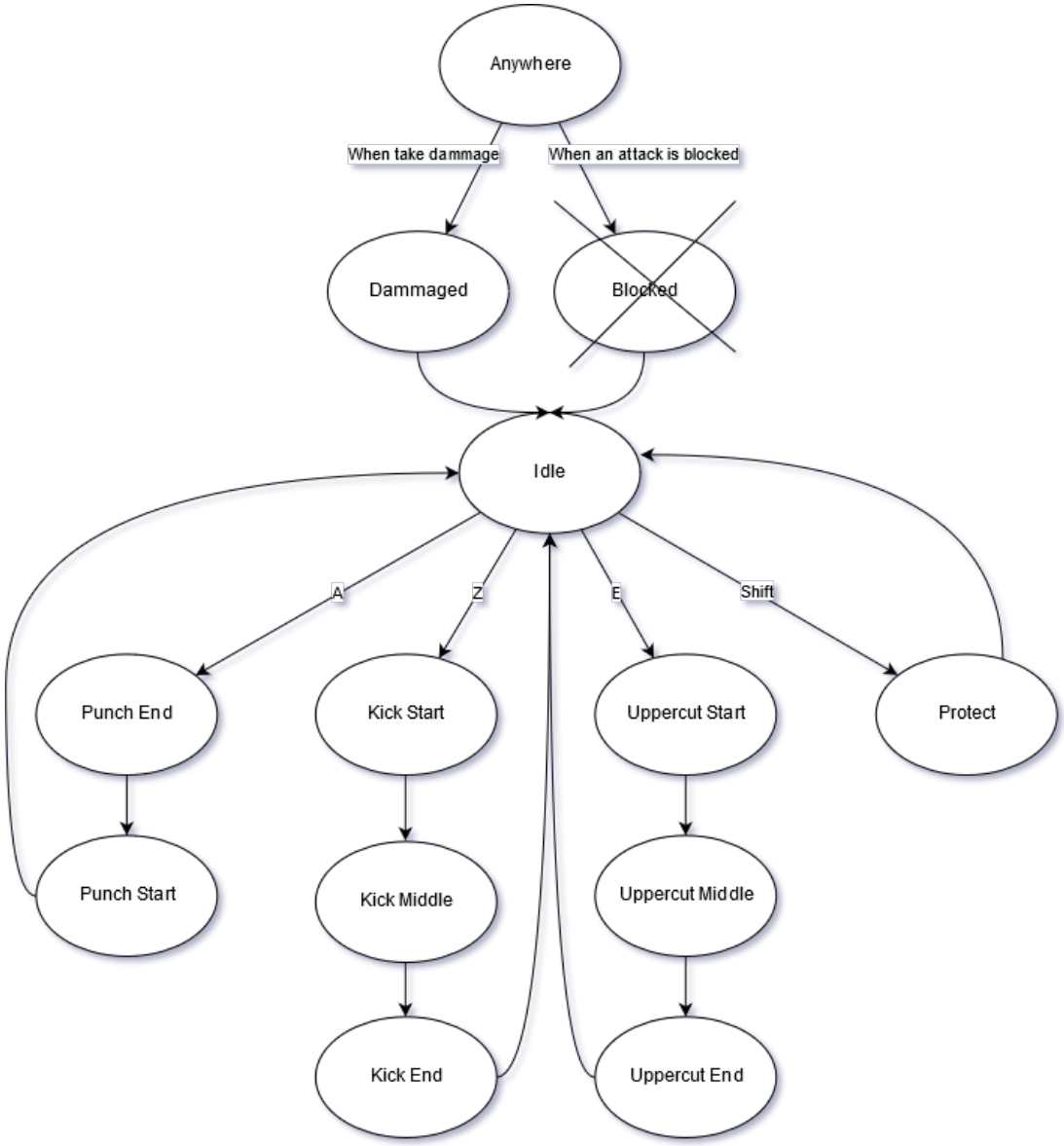
3.2 Stratégie ne connaissant pas le modèle : Q-Learning

On définit Q, la valeur d'une action à partir d'un état.

On a de manière trivial $V^*(s) = \underset{a}{max} Q^*(s,a)$.
Donc en injectant dans une des formules ci-dessus on a $Q^*(s,a) = R(s,a)+\gamma \sum_{s' \in S} T(s,a,s')Q^*(s')$
Ce qui nous donne la formule la plus important, THE Q-Learning rule :
$Q(s,a) = Q(s,a) + \alpha(r + \gamma(\underset{a'}{max} Q(s',a') - Q(s,a))$
où α est le taux d'apprentissage.
On a ainsi une QTable qui stocke toutes les QValeur associées à toutes les combinaisons possibles entre les états s et les actions a. Il s'agit souvent d'une matrice.
Cette formule, si appliqué à chaque instant entre notre état s' et celui précédent s permet de faire converger toutes les valeur de notre QTable vers des valeurs tels prendre le choix $\underset{a}{max} Q(s,a)$ à un état s donné revient à appliquer la stratégie π^* !

4 Un exemple : HadokeMDP

J'ai donc crée un jeu de combat style Street Figher, nommé HadokeMDP, très simple. Le but étant de reduire au maximum le nombre d'états.
Chaque personnage a une position de 0 à 10 et un état courant. Son état caractérise ce qu'il est en train de faire, comme en train de donner un coup de poing, ou un coup de pied etc ...
Voici la logique du jeu :



A chaque frame, on peut rentrer pour chaque joueur un input {gauche, rien, droite} et un input {coup de poing, coup de pied, uppercut, bloquer, rien}, pour un total de 15 combinaisons.
On a de plus un total de 29 000 états du jeu possible, ainsi notre QTable aura une taille de 450 000 ce qui est absolument énorme ! Il faudra trouver d'autres techniques pour généraliser à un jeu plus complet.

J'ai entrainé mes bots avec un $\gamma = 0.9$ et un $\alpha = 0.1$ et une probabilité d'exploration de $p = 0.9$ initiallement et qui diminuait d'un facteur 0.999 après chaque match.

Après 100 matchs, les bots avait déjà atteint un niveau humain voir bien meilleur mais étaient toujours vincibles. Ils avaient entre eux un taux de victoire de 50%.
Puis j'ai fais tourner mon programme d'entrainement une nuit entière, pour un total de 15 000 matchs ! A ce moment là, l'IA de droite à pris le dessus et battait l'IA de gauche 92% du temps ! J'ai compris par la suite que c'était du à une erreur de codage du jeu dans la fonction de déplacement : si les deux personnages étaient au corps à corps et que les deux voulaient aller dans la même direction, je faisais d'abord le test de déplacement pour le personnage de gauche, je l'appliquais, puis faisais de même pour le personnage de droite. Ainsi le personnage de gauche ne pouvait pas fuir le corps à corps si celui de droite voulait le poursuivre, tandis que le personnage de droite le pouvait !
Et c'est cette minuscule faille qui a permis à l'IA de droite de gagner presque 100% de ses parties, on voit bien ici à quel point le RL peut être aussi efficace que surprenant !

5 Avantages et Inconvénients

5.1 Avantages

— On a pas besoin de connaître la logique du jeu
— Pas besoin de données initiales
— Très effice !
— Il est facile d'adapter le niveau de difficulté de l'agent en réglant son paramètre p
— Très rapide en jeu, il suffit juste de consulter une QTable et de prendre le max !
— L'agent continue de s'adapter même une fois déployé

5.2 Inconvénients

— On a des temps de pré-calculs très importants !
— On a besoin d'un espace mémoire complètement astronomique
— Demande un univers discret
— Doit pouvoir être simulé, ce n'est donc pas très adapté à la vie réelle.

Conclusion

On a ainsi un algorithme de Q-Learning qui peut être tout aussi performant que facile à implémenter si les bonnes conditions sont réunis.
On notera d'ailleurs que le RL a été utilisé dans des programmes de renons tel que AlphaGo Zero et l'IA de Dota.