INTRODUCTION À L'APPRENTISSAGE PAR RENFORCEMENT, SANS CODE ET PRESQUE SANS MATHS

Édouard KLEIN

L'apprentissage par renforcement est un domaine de l'apprentissage statistique qui cherche à résoudre le problème de la prise de décision séquentielle dans l'incertitude.

L'apprentissage par renforcement est l'héritier du domaine du contrôle optimal, remontant aux années 50, cherchant à donner à des systèmes dynamiques la consigne qui permet d'optimiser un critère donné.

Tout cela est très abstrait, nous allons nous intéresser ici à des exemples concrets, qui présentent pour le domaine une difficulté croissante.

NOTE

Les lecteurs curieux des notions mathématiques sous-jacentes, parfois un peu pointues, pourront utilement se référer aux ouvrages de référence cités en fin d'article, tandis que les lecteurs souhaitant simplement une explication succincte du domaine pourront se contenter de mes approximations, dont je m'excuse par avance auprès des puristes.

Comme la plupart des méthodes de ML, les algorithmes de RL sont agnostiques au sens des données qu'ils traitent. Ils ne demandent :

- qu'une description numérique de l'état du système à contrôler ;
- la liste des actions possibles ;
- et une indication de la qualité du contrôle (c.f. section 1.1).

On peut utilement comprendre cette indication de qualité comme les variations du score d'un jeu vidéo, ou des entrées-sorties d'argent.

Si l'on est capable d'abstraire le problème réel auquel on est confronté en ces trois composantes numériques, alors les algorithmes de RL permettent de trouver l'action optimale à chaque instant, quel que soit le problème. Les subtilités qui viennent assombrir cette ambitieuse déclaration sont nombreuses, et nous allons les explorer au travers de l'optimisation de plans de production (section 1), de l'étude de règles dynamiques de sécurité dans un environnement virtuel (section 2), du A/B Testing (section 3), de l'analyse de logs (section 4). Nous verrons aussi comment il est possible de gérer un modèle adversarial où l'incertitude n'est pas seulement liée à la stochasticité du monde (par exemple le jet de dé dans un jeu de plateau), mais aux choix délibérés d'un autre agent qui cherche à minimiser notre critère de réussite (section 5).

ALLOCATION OPTIMALE DES RESSOURCES

États, actions, critère

Nous sommes en 1937 et Leonid Kantorovitch vient de recevoir une lettre d'une usine de contreplaqué qui lui demande comment allouer les matières premières à ses 8 machines pour produire les 5 différentes sortes de contreplaqué, dans les quantités demandées par le plan quinquennal.

C'est ainsi que commence la passionnante et tragique histoire de la programmation linéaire **[Spufford2012]**. La solution de Kantorovitch, qui lui vaudra le prix Nobel d'économie, ne sera jamais appliquée à l'échelle de l'Union. La résolution du problème implique des quantités qui ressemblent trop à des prix...

Le RL, ou plus précisément ici la programmation dynamique (*Dynamic programming*, D.P.) permet également de résoudre ce problème d'allocation des ressources. Il faut commencer par modéliser l'état du monde par un vecteur s_t indiquant pour la semaine t la quantité q_p de matière première disponible, le nombre d de semaines avant la fin du plan quinquennal et les quantités q_1 à q_5 des différentes sortes de contreplaqué déjà fabriquées :

$s_t = (q_p , d, q_1, q_2, q_3, q_4, q_5)$

Le camarade contremaître peut alors décider quelle machine $a_t=\{1,...,8\}$ doit être mise en route pour la semaine. En fonction de la machine qu'il a choisi de mettre en route, la quantité de matière première q_p va diminuer, et certaines quantités $q_1,...,q_5$ vont augmenter. Le temps disponible d va diminuer d'une semaine, amenant ainsi le système dans l'état s_{t+1}.

Cette évolution, à chaque pas de temps, d'un état s_t vers un autre s_{t+1} en fonction de l'action a_t choisie est appelée la *dynamique* du système.

Ce que l'on cherche, c'est une fonction π, appelée la politique, qui à chaque état possible s associe une action $a=\pi(s)$. Si la politique du contremaître est fixe, on se trouve alors dans ce que l'on appelle en mathématiques une chaîne de Markov (*Markov Chain*), dont la dynamique est celle du système, contrainte par la politique du contremaître.

Le contremaître étant libre de choisir sa politique de contrôle de l'usine, on va se placer dans le cadre plus général des processus décisionnels de Markov (*Markov Decision Process*, M.D.P). Ce cadre va l'aider à choisir la politique qui satisfait un critère d'optimalité. Ici, ce critère d'optimalité se mesure grâce à la différence, à la fin du Plan, entre les quantités produites par l'usine $(q_1,...,q_5)$ et les quantités demandées par le Госплáн $(q^*_1,...,q^*_5)$:

$d \neq 0 \Rightarrow R(s_t) = 0$

$d=0 \Rightarrow R(s_t) = -|q^*_1-q_1| - ... - |q^*_5-q_5|$

Avec ce critère, la sanction n'arrive qu'à la fin du Plan. Le contremaître peut optimiser localement : fixer fausse-

ment d à 0 et évaluer pour chaque semaine t la quantité d'ennuis $R(s_t)$ qu'il aurait si le plan finissait aujourd'hui. Il n'a plus qu'à choisir la machine a_t qui fera en sorte que la quantité d'ennuis $R(s_{t+1})$ qu'il aura la semaine prochaine si le plan finissait la semaine prochaine soit plus petite que $R(s_t)$.

NOTE

Par convention, l'on cherche à *maximiser* la somme des récompenses $R(s_t)$. Pour *minimiser* un critère (comme les ennuis du contremaître), il suffit d'en prendre l'opposé. Ici une différence entre la production et le plan entraînera une récompense d'autant plus négative que la différence est grande. La récompense maximale atteignable par le contremaître est 0.

Cette méthode correspond à l'intuition derrière certains algorithmes de programmation linéaire, qui ne s'embêtent pas à calculer les ennuis pour chaque état possible, mais suivent un chemin qui amène à la solution optimale sans explorer tout l'espace.

Les algorithmes de programmation dynamique arriveraient au même résultat, mais les calculs sont un peu plus laborieux :

- il faut résoudre la chaîne de markov en calculant pour chaque état s la somme des ennuis $V^{\pi}(s)$ que l'on va avoir en écoutant, à partir de cette semaine et jusqu'à la fin du Plan le contremaître et sa politique π. Cette somme d'ennuis est appelée la valeur d'un état.
- Il faut ensuite encourager le contremaître à améliorer sa politique π en une politique π' en choisissant à chaque état s_t l'action a_t qui nous fait aller dans l'état s_{t+1} dont la valeur $V^{\pi}(s_{t+1})$ est supérieure à celles des autres états qu'il est possible d'atteindre à partir de s_t.
- La politique étant modifiée, la chaîne de Markov a changé. Il faut maintenant calculer la nouvelle valeur $V^{\pi'}$ de tous les états.

Le cycle recommence jusqu'à ce que la politique ne varie plus : un des résultats centraux de la programmation dynamique et que ces itérations successives d'amélioration de la politique et de mise à jour de la valeur des états mène forcément vers un point fixe où la politique améliorée est la même que celle qui a été évaluée : c'est l'une des politiques optimales.

La programmation dynamique peut se résoudre avec cette famille d'algorithmes qu'on appelle itération généralisée de la politique (*Generalized Policy Iteration*, GPI) **[Sutton1998]**, où les étapes d'amélioration et d'évaluation de la politique se succèdent plus ou moins rapidement. Dans les cas extrêmes, soit l'on met à jour la valeur de tous les états avant de changer la politique (c'est l'algorithme « d'itération de la politique », *Policy Iteration*), soit l'on change la politique dès qu'on réévalue la valeur d'un état (itération de la valeur, *Value Iteration*).

Ici, le problème pouvant s'exprimer par des contraintes linéaires sur l'espace d'état, la programmation linéaire nous permet d'arriver plus rapidement à la solution optimale.

Stochasticité et non linéarité

Mais le contremaître n'est pas un bleu. Il sait qu'il ne faut pas trop en demander aux machines : à chaque fois qu'on en utilise une sans l'avoir arrêtée une semaine pour la réviser, la probabilité de la voir bloquée par une panne augmente.

Avec un programme linéaire, on ne pourrait prendre ces contraintes en compte qu'en réduisant la capacité de production estimée des machines. Ce chiffre restant forcément une sur- ou sous-estimation du vrai chiffre selon que le contremaître organise correctement la maintenance, la solution trouvée n'est plus optimale.

Avec la programmation dynamique, en revanche, on peut facilement rajouter 8 composantes $m_1,...,m_8$ à l'espace d'état, qui indiquent pour chaque machine le nombre de semaines où elles ont fonctionné depuis la dernière maintenance, ou si une machine est en panne (avec une composante négative indiquant la gravité de la panne).

Il suffit alors de coder dans la dynamique l'évolution de la probabilité des pannes (qui n'est vraisemblablement pas linéaire), et dans le critère R le coût de réparation (qui n'est sans doute lui non plus pas linéaire, les pannes dues à une absence totale de maintenance sont souvent beaucoup plus graves que les pannes de routine) :

$d \neq 0$ et $m_1 \geq 0, ..., m_8 \geq 0 \Rightarrow R(s_t) = 0$

$\exists\, i, m_i < 0 \Rightarrow R(s_t) = -m_i^2$

$d=0 \Rightarrow R(s_t) = -|q^*_1 - q_1| - ... - |q^*_5 - q_5|$

La dynamique indique, pour chaque état s_t et chaque action a_t non pas l'état s_{t+1}, comme je l'ai fait entendre jusqu'à présent, mais la distribution de probabilité de

l'état s_{t+1} sur l'espace d'état : utiliser la machine 3 peut nous faire arriver dans un état s_{t+1} où les quantités q_2, q_4 et q_7 augmentent, ou dans un état où elles restent identiques et la machine 3 est en panne. Les mécanismes stochastiques sont ainsi pris en compte au cœur de la méthode.

Utiliser une machine n'ayant pas bénéficié d'une maintenance récente devient ainsi un pari, et le calcul de la valeur d'un état revient au calcul du compromis entre assumer le coût de la panne ou désobéir au planificateur.

La programmation dynamique permet au contremaître de remplir le Plan au mieux, en alternant les périodes de travail et de maintenance de ses machines.

Résumé

Pour résoudre un problème avec la programmation dynamique, il faut le modéliser sous la forme :

- d'un espace d'état qui encode toutes les données pertinentes sous la forme d'un nombre fini de vecteurs *s* ;
- d'un espace d'action qui présente, à chaque pas de temps, les choix *a* disponibles à l'opérateur ;
- d'un critère *R* que l'on appelle fonction de récompense, et qui est le coût instantané d'une action dans un état.

La famille d'algorithmes d'itération généralisée de la politique fournit alors une politique π^* qui permet de maximiser la somme des récompenses (et donc d'accepter une perte à un moment si elle permet un gain futur, ou d'éviter une perte plus grosse encore).

POLITIQUE AUTOMATIQUE DE SÉCURITÉ

Faites vos jeux

Fort de ces enseignements, vous décidez d'utiliser les indicateurs de votre système de détection d'intrusion (*Intrusion Detection System*, IDS) comme espace d'état :

- nombre d'utilisateurs connectés,
- nombre de CVE ces dernières 24 heures,
- taux de warnings et d'erreurs,
- *load average*, mémoire utilisée,
- débit entrants et sortants,
- nombre de flux simultanés,
- âge moyen des quelques plus anciennes connexions,
- etc.

Vous rencontrez là un des premiers pièges de la modélisation pour le RL : le vecteur d'état *s* doit contenir toute l'information nécessaire pour prendre la décision *a*. La politique π, pour trouver $a=\pi(s)$, ne doit pas avoir besoin de regarder la suite d'états qui nous a menés jusqu'à *s*.

Cela implique qu'il ne faut pas se contenter de placer les indicateurs instantanés de l'IDS, mais aussi quelques valeurs historiques plutôt récentes.

Par exemple, une diminution subite du nombre de login ratés peut indiquer que l'attaquant s'est lassé, qu'il a été banni par un dispositif du style *fail2ban*, ou plus embêtant, qu'il a réussi. Il peut être alors judicieux de savoir si le nombre d'utilisateurs connectés a augmenté de 1. Cela implique d'avoir les valeurs instantanées et historiques dans l'état *s*, ce qui a pour effet de faire gonfler spectaculairement la taille de l'espace d'état.

Afin d'endiguer un peu cette augmentation exponentielle de la taille, vous êtes amené à discrétiser certaines caractéristiques : ainsi par exemple les débits ne vont pas se mesurer en kbps, mais être divisés en trois valeurs : basse, moyenne, haute.

Les actions possibles sont assez simples :

- alerter l'équipe de réponse à incidents,
- éteindre telle ou telle machine,
- durcir ou adoucir les règles du firewall selon des configurations préétablies,
- éclencher une sauvegarde,
- etc.

C'est en cherchant à définir les probabilités de transition que les limites de la programmation dynamique vous frappent de front. Comment parvenir à chiffrer exactement, par exemple, les probabilités des variations relatives des flux entrants et sortants et du taux de warnings et d'erreurs ?

Un problème réel, même modérément complexe, est impossible à modéliser dans le cadre de la programmation dynamique, sauf au prix d'approximations qui rendent très théorique l'optimalité de la solution trouvée.

C'est là que le cadre devient celui de l'apprentissage par renforcement. Il va falloir bâtir un simulateur du système afin de pouvoir évaluer la valeur des différents états.

La méthode, le MCMC pour *Markov Chain Monte-Carlo* est conceptuellement simple (mais il existe de nombreuses subtilités **[Andrieu2003]**). Il s'agit de placer le simulateur dans un état *s* et de laisser la politique π réagir. Plusieurs fois. Pour tous les états *s*. À la fin, on a une très bonne approximation de la valeur V^{π}. La politique sera mise à jour à certains intervalles, dont le choix dépend du compromis entre itération de la politique et itération de la valeur.

Un premier souci est que les garanties d'optimalité de la politique qui existent pour les algorithmes de programmation dynamique tombent dès que l'on utilise des approximations. Le nombre d'échantillons qu'il est nécessaire de recueillir pour satisfaire les bornes théoriques est bien trop grand. En pratique cependant le MCMC fonctionne tant que la dynamique n'est pas complètement aléatoire, et les dynamiques réelles le sont rarement.

Le deuxième souci est la construction du simulateur. Bâtir un simulateur pour un jeu vidéo ou un jeu de plateau est comparativement assez trivial. Dans notre exemple, il faudrait :

- faire un parc de machines virtuelles pour simuler toutes les machines du réseau ;
- user et abuser des snapshots de ces machines pour revenir à un

état *s* après un run de la simulation, afin de lancer une nouvelle branche du Monte-Carlo ;

- demander au CERT et à la Red Team d'élaborer des scénarios d'attaque scriptés qui doivent permettre entre autres de définir la récompense *R* ;
- être capable de placer le simulateur dans chacun des états *s*.

La récompense *R* est calculée en fonction des actions de la politique de sécurité (par exemple : pénalité pour chaque utilisateur connecté à une machine que l'on choisit d'éteindre, pénalité pour avoir réveillé l'astreinte pour rien), mais aussi en fonction des attaques scriptées : pénalité pour chaque instant où l'attaque parvient à extraire des données, ou à miner des cryptomonnaies, ou à chiffrer les fichiers, etc., mais bonus pour un réveil de l'astreinte alors que l'attaque est en cours ou l'extinction d'une machine où l'attaquant se trouve.

Créer un tel simulateur nécessite une rigueur de développement considérable, mais cela reste encore dans le domaine du possible. Cela présente en outre l'avantage de pouvoir tester des politiques de sécurité définies à la main, et d'entraîner la Red Team et la Blue Team.

Il n'est pas nécessaire pour les besoins de la simulation d'utiliser de vrais exploits (sauf si ceux-ci se voient dans les indicateurs de l'IDS). Il suffit de les simuler en donnant aux scripts de la Red Team les informations nécessaires à se logger sur les machines cibles. Cela simplifie grandement le développement de ces scripts. Certains scripts peuvent être aussi simples que de déposer une charge virale sur une machine.

La seule difficulté insurmontable dès que le réseau dépasse quelques machines est de pouvoir placer le simulateur dans des points de départ qui correspondent à chacun des états *s*. Cette tâche est rendue ardue d'une part par le simple nombre d'états, qui même dans les problèmes triviaux dépasse allègrement le millier, et d'autre part par le fait que certains états sont très très improbables et correspondent à des configurations complètement tordues, qu'il sera difficile, voire impossible, de reproduire dans le simulateur.

Approximations

Tout n'est pas perdu cependant. Il n'est pas nécessaire, comme nous l'avons fait jusqu'à présent, de maintenir une représentation tabulaire de la valeur $V^{\pi}(s)$ des états *s*. C'est au contraire le meilleur moyen de faire exploser le temps de calcul.

En pratique, les valeurs de deux états proches ont toutes les chances d'être proches. Il existe bien sûr des discontinuités, qu'il faut prendre en compte, car ce sont elles qui permettent au RL de résoudre des problèmes non linéaires.

Il faut non pas garder un dictionnaire qui à chaque *s* associe sa valeur $V^{\pi}(s)$, mais utiliser une méthode de régression (faire du ML. supervisé, donc) afin de généraliser à tout l'espace d'état la valeur trouvée par la méthode MCMC sur quelques points de départ dans lesquels il est possible de placer le simulateur. Il n'est plus non plus nécessaire de discrétiser les données fournies par l'IDS (on peut à nouveau compter les débits en kpbs).

D'un point de vue pratique, on cumule les avantages : le simulateur est plus simple à coder et les calculs deviennent moins gourmands.

D'un point de vue théorique, c'est une catastrophe : on approxime une fonction V^{π} avec une méthode entraînée sur quelques points eux-mêmes approximés (en effet, MCMC ne renvoie qu'une approximation de $V^{\pi}(s)$).

Pour contourner cette catastrophe théorique, il faut s'assurer que la méthode d'approximation de la fonction de valeur travaille dans un espace d'hypothèses qui permet de représenter la vraie fonction de valeur. Il faut donc lui représenter l'espace d'état par le biais d'une fonction d'attribut (*feature*) qui met en avant les relations pertinentes.

Ce travail, mêlant à la fois connaissance théorique des méthodes de ML et connaissance métier du domaine (ici la détection et réponse à incidents de sécurité) est appelé le *feature engineering* et représente en pratique la part la plus importante de la mise en place d'une solution de RL.

Il n'existe pas de méthode générique. Le *feature engineering* est vu comme de « la triche » dans le milieu académique, car il permet à n'importe quelle méthode de briller. C'est pourquoi depuis quelques années la recherche se porte sur la détection automatique de features. Les réseaux de neurones s'en sortent très bien, mais il faut pouvoir les entraîner, il est difficile de comprendre les features qu'ils fournissent **[Samek2017]** et ils sont sujets à des attaques **[Papernot2016]**.

A/B TESTING

Découragé par le travail de titan qu'est la programmation du simulateur, vous décidez plutôt de vous consacrer à un problème plus simple, dont la formulation date des années 50 et qui peut avoir un effet rapide sur votre entreprise : le *A/B Testing*.

Il s'agit de présenter aux visiteurs d'un site web deux versions du site, et de choisir automatiquement la version qui déclenche le comportement souhaité. Les considérations éthiques dépassent le cadre de cet article, mais sachez que ce type d'optimisation amène par exemple Facebook à enfermer ses utilisateurs dans une bulle d'opinion politique (ce qui ne les empêche pas de blâmer les utilisateurs **[Bakshy2015]**), ce qui peut amener à une radicalisation.

Ce problème est connu sous le nom du problème des bandits manchots, et est étudié pour tenter de résoudre le dilemme exploration exploitation : faut-il continuer à jouer à une machine à sous dont l'espérance est la meilleure connue (la version A du site), ou doit-on essayer la machine d'à côté dont l'espérance est peut-être supérieure (la version B du site) ? Quand doit-on s'arrêter ? Après tout, on a peut-être joué de malchance, si l'espérance de la machine B est inférieure, c'est dû au hasard, en jouant plus dessus on en aurait le cœur net.

Le RL permet de résoudre ce problème, avec des stratégies qui vont du très simple (**http://stevehanov.ca/blog/index.php?id=132**) au très complexe (**http://downloads.tor-lattimore.com/banditbook/book.pdf**). Néanmoins, il est important de noter que ce problème en apparence plus simple que le problème précédent présente en réalité une caractéristique qui le rend plus complexe : il est impossible de bâtir un simulateur.

La seule solution pour évaluer une politique π consiste donc à la mettre aux commandes et à la laisser choisir la version A ou la version B. Pour le A/B Testing, il n'est pas forcément très grave de présenter une version sous-optimale du site à quelques visiteurs, surtout si les deux versions ont été approuvées par les collègues du contrôle qualité. De toute façon, l'algorithme de RL convergera très vite vers la meilleure version.

Dans les tests cliniques, en revanche...

Il n'est pas toujours possible de bâtir un simulateur réaliste, et il n'est parfois pas possible de laisser une politique encore non optimale aux commandes du système. Comment dans ce cas évaluer la politique, étape nécessaire à son amélioration ?

APPRENDRE AVEC LES LOGS

Revenons au cadre de l'apprentissage d'une politique automatique de sécurité.

Si l'on ne donne pas à l'équipe de développement les moyens de développer le simulateur qui permettrait d'apprendre la politique de sécurité optimale, tout espoir n'est pas perdu. Il reste possible d'utiliser une classe d'algorithmes dits *off-policy*. Il existe des algorithmes de RL capables d'apprendre la politique optimale, même s'ils ne peuvent influencer les choix des actions afin d'en apprendre les conséquences.

La convergence des algorithmes *off-policy* est forcément plus lente et, si la politique de contrôle est lacunaire, la politique apprise ne pourra pas miraculeusement être optimale. Mais l'on dispose maintenant d'une solution qui permet d'apprendre à contrôler un système sans le casser pendant l'apprentissage.

Il suffit maintenant de logger les actions des administrateurs réseau pour pouvoir, avec le log de l'IDS, créer les données historiques s_t, $a_t \rightarrow R(s_t)$, s_{t+1} nécessaires à l'entraînement des algorithmes *off-policy*. Plus besoin de simulateur.

APPRENDRE À DÉFAIRE UN ADVERSAIRE

Certains problèmes présentent un élément incertain particulièrement difficile à analyser : un adversaire intelligent.

Une méthode qui a fait ses preuves en intelligence artificielle est la famille d'algorithmes dits *mini-max* : il s'agit de prendre l'action qui *mini*-mise l'espérance de gain de l'adversaire, en supposant que celui-ci joue parfaitement (qu'il *max*-imise son gain).

Pour appliquer la méthode MCMC, cela présuppose que l'on connaît la meilleure action que l'adversaire pourrait effectuer. Ce qui implique d'avoir déjà une stratégie optimale sous la main.

Dans les problèmes symétriques, il existe une astuce qui consiste à utiliser la même politique des deux côtés. La politique s'améliorant, on joue au fur et à mesure de l'entraînement contre un adversaire de plus en plus performant. C'est le principe utilisé par AlphaGo Zero **[Silver2017]**, à la différence de la version originale, qui avait analysé un énorme nombre de parties jouées par les meilleurs joueurs du monde.

Si le problème est asymétrique, comme l'est l'attaque et la défense d'un périmètre informatique, cette astuce ne peut fonctionner.

CONCLUSION

Le RL permet de résoudre des problèmes stochastiques non linéaires. L'effort de modélisation doit nécessairement passer par la présentation du problème sous la forme d'un processus de prise de décisions séquentielles, avec un espace d'état dont les éléments s_t donnent toute l'information nécessaire à la prise de la décision a_t, entraînant le système à transitionner dans l'état s_{t+1} et à recevoir la récompense $R(s_t)$ au passage.

Selon les cas, on peut ensuite exprimer la dynamique sous la forme de probabilités de transitions explicites comme dans l'exemple d'allocation des ressources, sous la forme d'un simulateur comme dans l'exemple de la politique de sécurité, en mettant la machine aux commandes comme pour le *A/B Testing* ou enfin en analysant les données recueillies comme dans le cas des algorithmes *off-policy*.

Dans la plupart des cas, la modélisation mettra en jeu une approximation de la fonction de valeur (la somme des récompenses tout au long du fonctionnement du système). Trouver les bonnes *features* pour que cette approximation ne trahisse pas la théorie derrière les méthodes, là est l'art du Renforcement. ▌

RÉFÉRENCES

La bible (régulièrement rééditée) est le livre de Sutton et Barto **[Sutton1998]**, la bible en Français est le « PDMIA » **[Sigaud2008]**.

Il n'y a pas d'implémentation libre de référence. Le cœur des algorithmes est en général assez trivial à implémenter (par exemple pour un petit jeu vidéo, l'auteur a codé Q learning en 30 lignes de Python disponibles sur : **https://github.com/edouardklein/Outsmart/blob/master/rl.py#L34**). La difficulté résidant dans le choix d'un bon schéma d'approximation, qui dépend du domaine d'application, il est difficile de proposer une librairie de référence comme l'est Scikit-Learn, par exemple. On pourra aussi lire **[Geron2017]** pour avoir des exemples.

[Spufford2012] F. Spufford, *Red Plenty*, Graywolf Press, 2012.

[Sutton1998] R. S. Sutton & A. G. Barto, *Reinforcement learning: An introduction*, MIT Press Cambridge, 1998. Disponible gratuitement en ligne (**http://incompleteideas.net/book/bookdraft2017nov5.pdf**).

[Andrieu2003] C. Andrieu, N. de Freitas, A. Doucet & M.-I. Jordan, *An introduction to MCMC for machine learning, Machine learning*, Machine Learning, pp 5-43, 2003.

[Samek2017] W. Samek, T. Wiegand & K.-R.Müller, *Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models*, **https://arxiv.org/abs/1708.08296**

[Papernot2016] Papernot, McDaniel, Jha, Fredrikson, Celik & Swami, *The limitations of Deep Learning in adversarial settings*, IEEE European Symposium on Security and Privacy (EuroS&P), 2016.

[Bakshy2015] Bakshy, Messing & Adamic, *Exposure to ideologically diverse news and opinion on Facebook*, Science, 5 june, VOL 348 Issue 6239, 2015.

[Silver2017] Silver, Schrittwieser, Simonyan, Antonoglou, Huang, Guez, Hubert, Baker, Lai, Bolton & others, *Mastering the game of Go without human knowledge*, Nature, 2017.

[Sigaud2008] 0. Sigaud & 0. Buffet, *Processus décisionnels de Markov en intelligence artificielle*, Hermes Science Publications, 2008.

[Geron2017] A. Géron : *Deep Learning avec TensorFlow , Mise en Œuvre et cas concrets*, Dunod, 2017.