Projet de Reinforcement Learning : Can Deep Reinforcement Learning Solve Erdos-Selfridge-Spencer Games?

Jourdan Fanny, Kläy Léna, Lo Ndeye

Janvier 2020

Table des matières

Ce rapport se base sur l'article : *Can Deep Reinforcement Learning Solve Erdos-Selfridge-Spencer Games ?* [1]

1 Introduction

Malgré la quintessence de l'apprentissage par renforcement ces dernières années et de leur nombreux succès ; développer des algorithmes d'apprentissage robustes dans certains environnements reste un défi.

Les repères standard comme MuJoCo et Atari fournissent des paramètres riches pour l'expérimentation, mais les spécificités des environnements sous-jacents diffèrent les unes des autres de multiples façons, et donc déterminer les principes sous-jacents à toute forme particulière de comportement sous-optimal est difficile. Trouver le comportement optimal est complexe car il n'est pas entièrement caractérisé.

Pour remédier à ce problème il faudrait pouvoir se placer dans un cadre idéal pour étudier les forces et les limites des algorithmes d'apprentissage par renforcement.
Ce dernier devra être :
- une famille d'environnements simplement paramétrés
- où le comportement optimal peut être complètement caractérisé
- et l'environnement est suffisamment riche pour soutenir l'interaction et le jeu multi-agents.

Ce type d'environnement se retrouve dans les jeux Erdos-Selfridge-Spencer (ESS).

2 Règles du jeu

Les jeux Erdos-Selfridge-Spencer sont des jeux dans lesquels deux joueurs sélectionnent à tour de rôle des objets à partir d'une structure combinatoire. Les stratégies optimales de ces jeux peuvent être définies par des fonctions potentielles dérivées d'attentes conditionnelles sur un jeu futur aléatoire. Dans cette famille de jeux, on va particulièrement s'intéresser au jeu attaquant-défenseur de Spencer.

Le jeu attaquant-défenseur de Spencer est composé d'un plateau sur lequel sont disposés des pions et fait s'affronter deux joueurs : l'attaquant et le défenseur. Nous considérons un plateau avec K+1 lignes (numérotées de 0 à K) et N pions. L'objectif de l'attaquant est d'amener au moins un pion sur la ligne K, le but du défenseur est de débarasser le plateau de tous ses pions. A chaque tour, l'attaquant choisit une partition composée de deux sous-ensembles (A et B) parmi les N pions. Le défenseur joue alors et choisit un sous-ensemble de pions qui sera retiré du plateau (disons A). Les pions restant (B) sont conservés et progressent d'une ligne vers le haut du plateau, avant que l'attaquant ne rejoue.

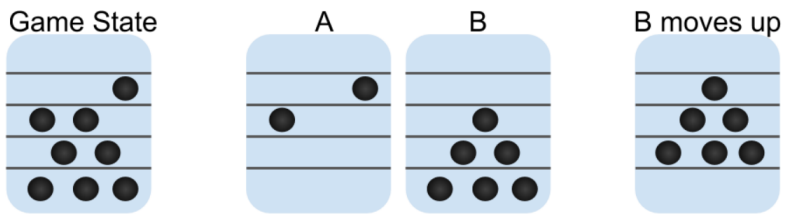


FIGURE 1 – Illustration d'un tour de jeu attaquant-défenseur de Spencer

3 Stratégies naïves

Nous appelons stratégies naïves les stratégies qui consistent à jouer "au hasard". La stratégie naïve pour l'attaquant consiste à placer chaque pion dans le premier sous-ensemble avec une probabilité de 0.5. Tous les autres pions sont placés dans le second sous-ensemble. La stratégie naïve pour le défenseur consiste à choisir entre les deux sous-ensemble selon une loi de Bernouilli de paramètre 0.5.

4 Stratégies optimales

Il existe pour ce jeu, des stratégies optimales qui sont détaillées dans l'article [1]. Pour pouvoir les expliquer, nous introduisons la fonction potentiel, définie comme suit :

$$\phi(S) = \sum_{i=0}^{K} n_i 2^{-(K-i)}$$

où n_i est le nombre de pion sur la ligne i, et où $S = (n_0, n_1, ..., n_K)$ est l'état du jeu. On remarque que cette fonction est linéaire, en particulier pour une partition $\{A, B\}$ de S, on obtient $\phi(S) = \phi(A) + \phi(B)$. Une seconde remarque que nous pouvons faire concerne cette fois l'étape où les pions avancent d'une ligne : soit B l'état avant le déplacement et B' l'état après, il est facile de voir que $\phi(B') = 2\phi(B)$.

La fonction potentiel reflète la façon dont l'état S avantage tel ou tel joueur. En effet, il est possible de montrer que si $\phi(S) \geq 1$, alors l'attaquant en jouant de façon optimale est sûr de remporter la partie. A l'inverse si $\phi(S) < 1$, un défenseur optimal est sûr de gagner. Détaillons maintenant ces stratégies optimales.

Intuitivement, plus $\phi(S)$ est grand et plus l'attaquant est avantagé. En remarquant cela, la stratégie optimale pour le défenseur consiste à supprimer le sous-ensemble de plus haut potentiel.

La stratégie optimale pour l'attaquant face à ce type de défenseur consiste alors à maximiser le minimum des potentiels des sous-ensembles A et B. Il va pour ce faire répartir les pions dans les deux sous-ensembles de façon à avoir $\phi(A)$ et $\phi(B)$ les plus proches possibles.

Il peut être intéressant de discuter "l'optimalité" de cette dernière stratégie : en effet, cette stratégie n'est optimale que lorsque le potentiel de l'état initial S_0 est tel que $\phi(S_0) \geq 1$. En effet considérons que ce ne soit pas le cas : $\phi(S_0) < 1$. On peut alors imaginer que notre attaquant "optimal" proposera souvent deux sous-ensembles A et B tels que $\phi(A) < 0.5$ et $\phi(S_0) < 0.5$. Un défenseur, même aléatoire, sera donc toujours en position de force avec un jeu de potentiel strictement inférieur à 1 et finira par gagner. A l'inverse si l'attaquant joue aléatoirement et propose de temps en temps un sous-ensemble de potentiel supérieur à 0.5, il y a une probabilité non-négligeable pour que ce sous-ensemble soit sélectionné. Le jeu deviendrait alors de potentiel supérieur à 1, ce qui pourrait permettre à l'attaquant de gagner. Notons bien que si le défenseur avait été optimal, l'attaquant aurait perdu et cela quelque soit sa stratégie (car $\phi(S_0) < 1$).

Nous implémentons ces deux stratégies et les testons pour quelques valeurs de potentiel de l'état initial (première ligne). Nous regroupons ici les pourcentages de parties gagnées par l'attaquant, sur un total de 1000 parties.

Attaquant	Défenseur	0.7	0.8	0.9	1	1.1	1.2
Naïf	Naïf	55.5 %	59.4 %	64.5 %	71.6 %	74.1 %	77.7 %
Optimal	Naïf	69.6 %	78.6 %	90.6 %	100 %	100 %	100 %
Naïf	Optimal	0 %	0 %	0 %	22.7 %	22.7 %	26.7 %
Optimal	Optimal	0 %	0 %	0 %	100 %	100 %	100 %

On voit que pour $\phi(S_0) \geq 1$ l'attaquant optimal gagne toujours, tandis que pour $\phi(S_0) < 1$ c'est le défenseur optimal qui remporte toujours la partie. Ces résultats semblent donc valider les stratégies optimales et leurs implémentations.

5 Stratégies sous-optimales

On appelle stratégie sous-optimale une stratégie qui sera optimal 90% du temps et non-optimal les 10% restant. Dans le cas du défenseur, le sous-ensemble de plus fort potentiel est détruit en général, mais de temps en temps il est conservé. Dans le cas de l'attaquant, rappelons que la partition optimale $\{A, B\}$ de S est telle que $\phi(A) \simeq \phi(B) \simeq 0.5\phi(S)$. Cette stratégie sera suivie par l'attaquant sous-optimal dans 90% des cas, mais dans 10% des cas elle sera telle que $\phi(A) \simeq \pi\phi(S)$ et $\phi(B) \simeq (1 - \pi)\phi(S)$ avec π choisi uniformément dans $\{0.1, 0.2, 0.3, 0.4\}$.

6 Implémentation du jeu

Il est intéressant de noter que l'emplacement d'un pion sur la ligne n'a pas d'influence ; qu'il soit à droite, à gauche ou au milieu, la situation est la même. Nous implémentons donc l'état du jeu sous forme d'un vecteur comprenant K+1 coefficients et chaque coefficent sera égal au nombre de pions présents sur la ligne.

Les paramètres du jeu sont le nombre de lignes et de colonnes du plateau, et nous laissons également la possibilité à l'utilisateur de choisir arbitrairement le potentiel de l'état initial. Afin d'appréhender le jeu et de vérifier sa bonne fonctionnalité, nous implémentons également une partie contre l'ordinateur (en temps qu'attaquant ou défenseur).

7 Elaboration d'un algorithme de reinforcement learning

Nous souhaitons entraîner une intelligence artificielle à jouer soit dans le rôle de l'attaquant, soit dans celui du défenseur. Pour cela nous choisissons de développer une approche de reinforcement learning. Cette approche permettra à l'ordinateur d'apprendre de lui-même la meilleure stratégie en fonction du plateau de jeu, sans qu'on ait à lui fournir de données (qu'il serait long d'amasser).

Classiquement les algorithmes de reinforcement learning stoque pour chaque état de l'environnement un score qui leur permet d'établir une stratégie. Ce score va être mis à jour au fur et à mesure de la phase d'entrainement. Un état correspond pour nous à une configuration de jeu. Pour un plateau comportant 7 lignes, chacune d'elle pouvant accueillir de 0 à 5 pions, nous avons 6^6 c'est à dire 46 656 états possibles. [1] C'est un nombre d'états bien trop grand.

1. Nous ne considérons pas les états comportant des pions sur la dernière ligne car cela signifie la fin de la partie.

Afin de contourner ce problème, nous envisageons de voir le jeu différemment : au lieu d'attribuer un score à chaque état possible, nous attribuons un score à chaque pion du jeu. Le score de l'état correspondra alors à la somme des scores des pions le composant. Afin de rester cohérent avec ce que nous avons précédemment établit plus haut [6], on attribue des scores égaux aux pions présents sur une même ligne. En pratique il nous suffit donc de renseigner un score par ligne.

Nous entrainons notre joueur sur 2000 parties. Ce nombre a été choisi de façon à atteindre un état à peu près stable tout en conservant un temps de simulation raisonnable. Au début de la phase d'entrainement, toutes les lignes ont un score de 2000. Après chaque partie, on met alors à jour les scores des lignes en fonction de si l'algorithme a gagné ou perdu.

Supposons dans un premier temps que nous entrainons un défenseur. Si l'algorithme perd cela signifie que les sous-ensembles qu'il n'a pas éliminé au cours de cette partie étaient de potentiels élevés. Afin de prendre cette information en compte pour les prochaines parties, les scores des lignes correspondantes vont être augmentés. Pour chaque ligne, on augmente le score du nombre de pions non-détruits divisé par le nombre de pions rencontrés (dans cette ligne). A l'inverse si l'algorithme gagne, on diminue les scores des lignes de cette même quantité.

Si nous entrainons l'attaquant, l'approche est exactement la même en considérant que si l'attaquant perd alors le défenseur gagne, et inversement.

Cette méthode sera appelée méthode 1. Une amélioration à cette démarche que nous trouvions intéressant d'étudier constitue la méthode 2 : le dernier coup est celui qui permet de remporter la partie, on est donc à peu près sûr que ce coup est bon (pour le gagnant) tandis que les coups précédents ont pu potentiellement être médiocres. En partant de cette observation nous rajoutons la récompense suivante : en plus des scores mis à jour comme décrit précédemment, nous ajouterons ou retirons 1 (en fonction de si nous avons ajouté ou retiré le score) aux scores des lignes sur lesquelles se situaient les pions du dernier sous ensemble conservé.

Si nous savons maintenant comment mettre à jour les scores des lignes, il nous reste à préciser de quelle façon joue l'algorithme pendant la partie. Comme tout algorithme de reinforcement learning, à chaque fois que c'est à son tour de jouer notre algorithme choisit aléatoirement entre deux actions :

— explorer l'environnement c'est à dire suivre la stratégie naïve (jouer au hasard),
— exploiter une strategie se basant sur les scores des lignes (mis à jour)

S'il choisit la deuxième option, alors sa stratégie sera similaire à celle de la stratégie optimale, mais avec les scores des lignes mis à jour dans les parties précédentes, plutôt que la fonction potentielle. Ainsi le score d'un état S est :

$$\Gamma(S) = \sum_{i=0}^{K} l_i n_i \tag{1}$$

où l_i est le score de la i^{eme} ligne et n_i le nombre de pions sur cette ligne. La stratégie sera alors pour le défenseur de choisir le sous-ensemble ayant le plus petit score $\Gamma(S)$, et celle de l'attaquant de tenter d'équilibrer le score $\Gamma(A)$ et $\Gamma(B)$ toujours selon cette formule.

Le choix entre explorer l'environnement ou appliquer la strategie Γ, suit une loi de Bernouilli de paramètre ϵ. (succès : exploration / échec : exploitation). Nous choisissons le paramètre ϵ de façon

à ce que l'algorithme explore souvent au début puis de moins en moins au fil des itérations. Après plusieurs essais, nous choisissons la formule suivante :

$$\epsilon(p) = 1 - \frac{\log(p)}{\log(P)^{1.5}}$$

où P est le nombre de parties total et p le nombre de parties déjà jouées. Graphiquement nous optenons :

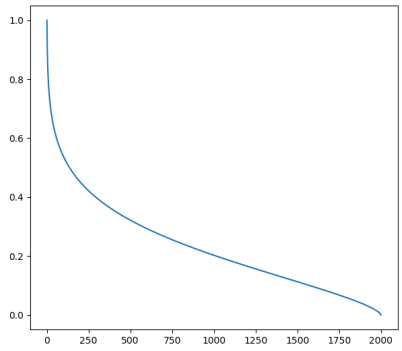


FIGURE 2 – Valeurs de ϵ en fonction du nombre de parties déjà jouées

7.1 Phase d'apprentissage

On entraine tout d'abord nos algorithmes contre les trois stratégies détaillées plus haut, et pour une palette de valeurs de potentiel de l'état initial. Dans tout ce rapport, nous entrainons notre algorithme défenseur à des potentiels inférieurs à 1 et notre algorithme attaquant à des potentiels supérieur à 1. Ce choix a été fait pour qu'il soit mesure de gagner de temps en temps, notamment si l'adversaire est optimal. On calcule le taux de victoires sur les 2000 parties d'entrainement et on choisit de faire 10 réplicats. Les moyennes sur ces réplicats des taux de victoires sont détaillés dans les tableaux suivants :

Apprentissage du défenseur

Contre l'attaquant naif :

	0.6	**0.7**	**0.8**	**0.9**
Méthode 1	0.86	0.81	0.76	0.74
Méthode 2	0.85	0.81	0.75	0.73

Contre l'attaquant sous-optimal :

	0.6	**0.7**	**0.8**	**0.9**
Méthode 1	0.59	0.40	0.20	0.16
Méthode 2	0.63	0.46	0.26	0.19

Contre l'attaquant optimal :

	0.6	**0.7**	**0.8**	**0.9**
Méthode 1	0.57	0.36	0.12	0.08
Méthode 2	0.62	0.43	0.22	0.12

Apprentissage de l'attaquant

Contre le défenseur naif :

	1	**1.1**	**1.2**	**1.3**
Méthode 1	0.90	0.84	0.90	0.91
Méthode 2	0.90	0.85	0.90	0.92

Contre le défenseur sous-optimal :

	1	**1.1**	**1.2**	**1.3**
Méthode 1	0.69	0.67	0.74	0.63
Méthode 2	0.68	0.41	0.68	0.70

Contre le défenseur optimal :

	1	**1.1**	**1.2**	**1.3**
Méthode 1	0.61	0.60	0.69	0.54
Méthode 2	0.61	0.38	0.60	0.62

On observe en règle général, que plus on s'approche de 1 et plus le taux de victoires est faible car la partie est alors plus difficile à gagner. Cependant cette observation n'est qu'une tendance et elle n'est pas immuable.

De même on observe qu'il est plus facile de gagner contre (dans l'ordre) l'adversaire naïf, puis sous-optimal, puis optimal, ce qui correspond à nos attentes.

La méthode 2 semble uniquement apporter un avantage significatif dans le cas de l'algorithme défenseur et contre l'attaquant optimal et sous-optimal.

Il est possible de tracer l'allule des courbes des taux de victoires en fonction du nombre d'itérations de l'algorithme pour chacun de ces cas. On voit qu'elles convergent toutes vers les valeurs indiquées dans le tableau, et pour les 10 réplicats considérés. Nous présentons les 4 premières courbes ci-dessous (dans le cas où nous entrainons le défenseur) :

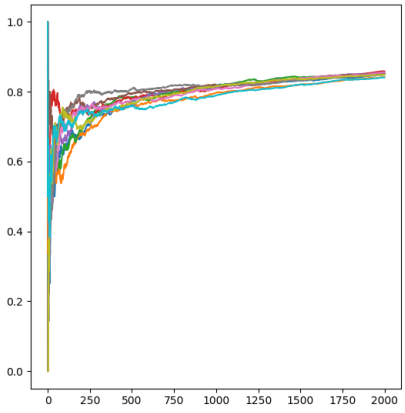


FIGURE 3 – Méthode 1, phi=0.6, Attaquant naïf

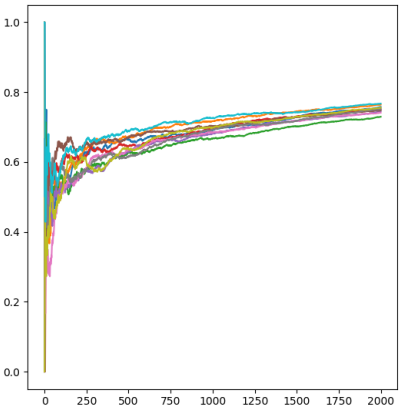


FIGURE 4 – Méthode 1, phi=0.7, Attaquant naïf

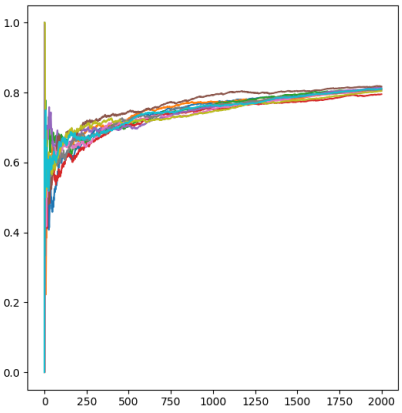


FIGURE 5 – Méthode 1, phi=0.8, Attaquant naïf

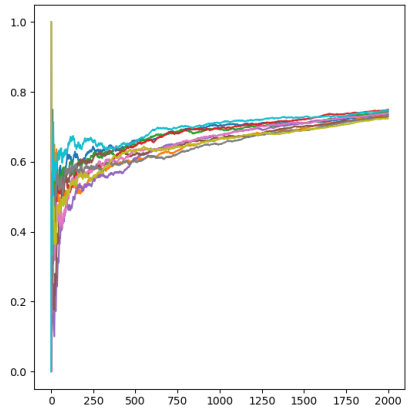
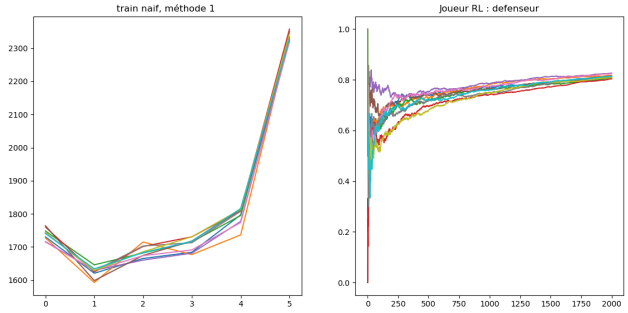


FIGURE 6 – Méthode 1, phi=0.9, Attaquant naïf

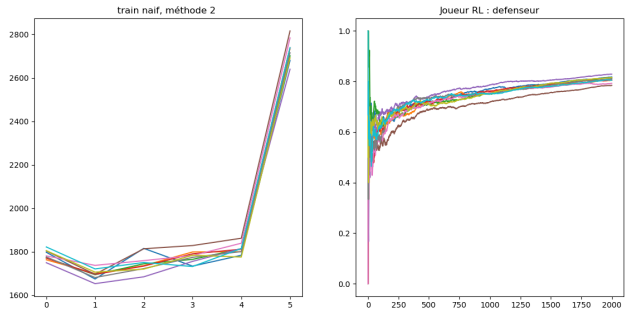
Afin d'établir des algorithmes les plus généraux possibles, nous choisissons dans la suite de définir le potentiel de l'état initial à chaque itération uniformément dans [0.5,0.9] pour l'algorithme défenseur et dans [1,1.4] pour l'algorithme attaquant. Nous présentons les résultats de convergence et de scores de lignes ci-dessous :

Attaquant naïf

Méthode 1

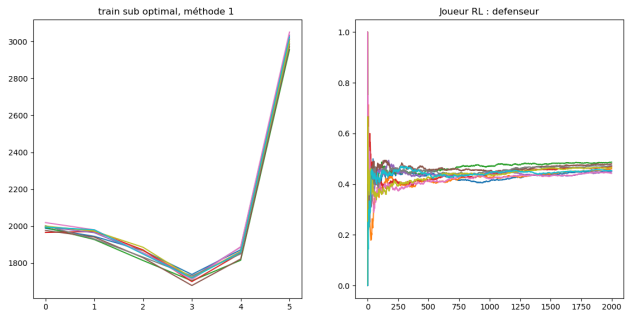


Méthode 2

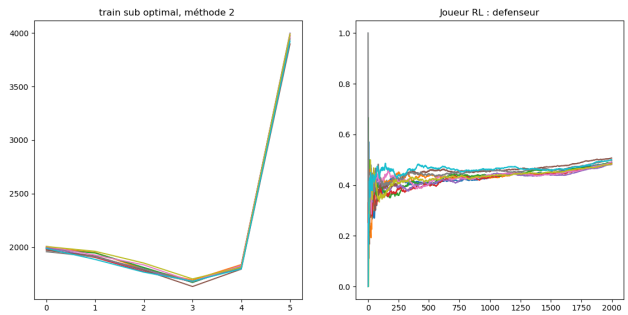


Attaquant sous-optimal

Méthode 1

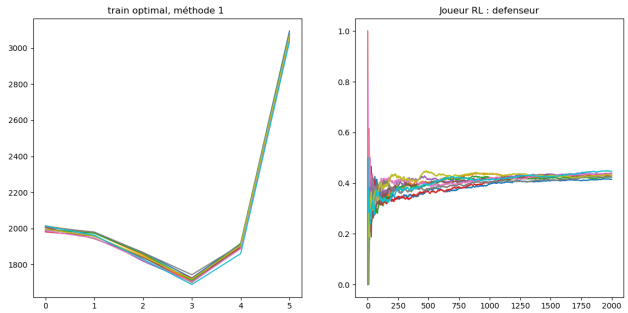


Méthode 2



Attaquant optimal

Méthode 1



Méthode 2

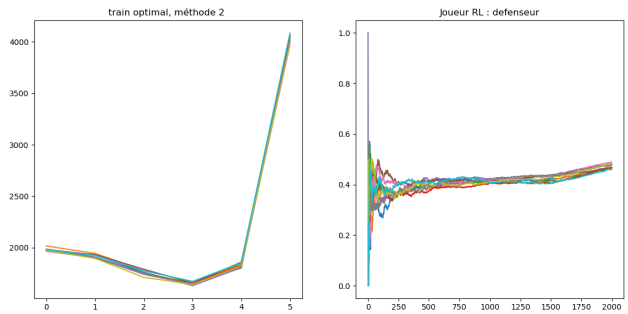
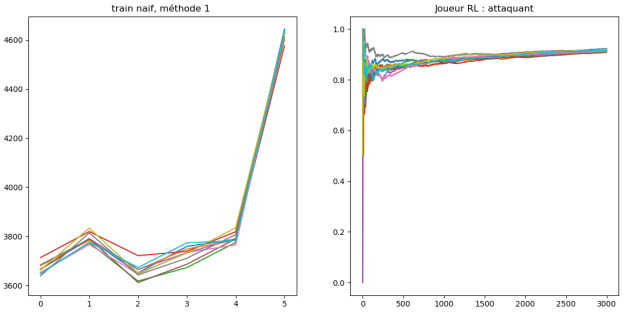


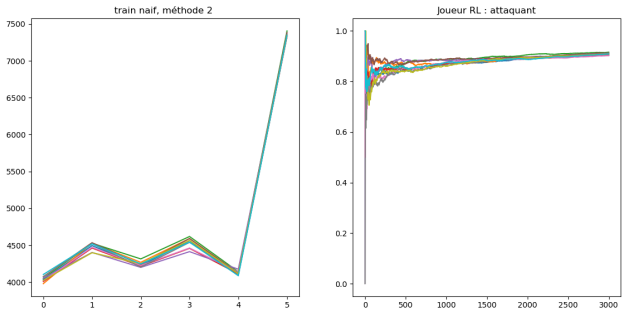
FIGURE 7 – Sous-graph de gauche : Score des lignes en fonction de leur position. Sous-graph de gauche : Taux de victoire de l'algorithme défenseur au cours de l'entrainement contre les différents attaquants.

Défenseur naïf

Méthode 1

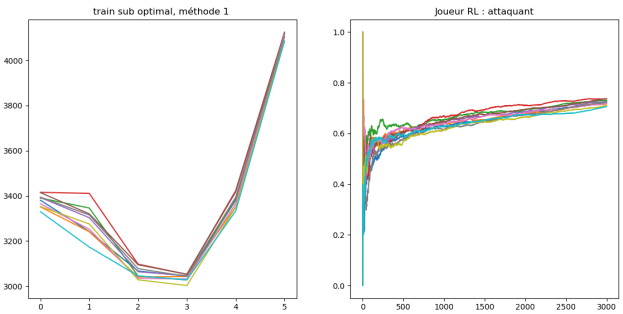


Méthode 2

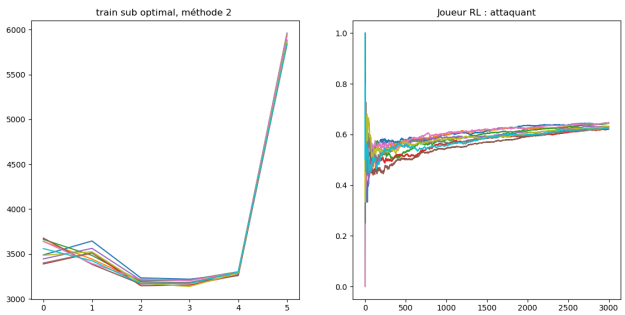


Défenseur sous-optimal

Méthode 1

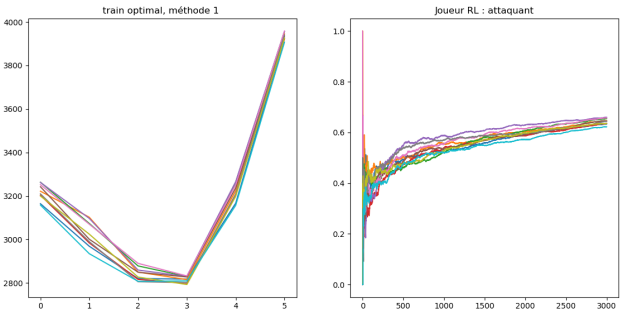


Méthode 2



Défenseur optimal

Méthode 1



Méthode 2

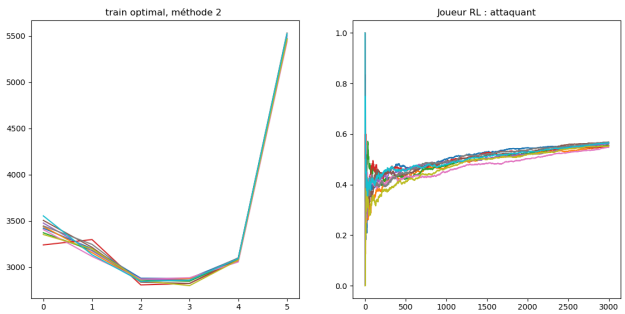


FIGURE 8 – Sous-graph de gauche : Score des lignes en fonction de leur position. Sous-graph de gauche : Taux de victoire de l'algorithme attaquant au cours de l'entrainement contre les différents défenseurs.

Tout d'abord, il semble que tous les algorithmes convergent quelques soient les conditions. Ils ne convergent cependant pas vers les mêmes taux de victoires (comme vu précédemment) et surtout pas vers les mêmes scores de lignes. Nous observons donc qu'en fonction de l'adversaire contre lequel il est entrainé, l'algorithme va attribuer des scores différents aux lignes. Nous voyons cependant grâce aux réplicats que ces scores ne sont pas du à une simple variabilité, mais bien à une adaptation de la stratégie à l'adversaire. Si on garde en tête que la stratégie optimale (avec la fonction potentielle) consiste à placer un score croissant sur chacune des lignes, alors on peut déduire deux choses :

— Ce score croissant se retrouve souvent sur les deux dernières lignes (et toujours sur la dernière ligne) : comme on pourrait s'y attendre l'algorithme accorde beaucoup d'importance à ces dernières lignes et les lignes se situant plus au début n'affectent pas beaucoup la stratégie (on a des profils très différents).

— Les profils des lignes du début (et milieu) ont beau être différents suivant les conditions, ils sont extrêment similaires au sein d'une seule condition (réplicats). On peut alors supposer que certaines configurations sont particulièrement dangereuses face à certains adversaires et ces scores de lignes les contrent.

7.2 Confrontation à un adversaire

Nous faisons maintenant jouer nos algorithme entrainés contre différents adversaires. Il n'y a plus de phase d'exploration, l'algorithme joue uniquement de façon "optimale en accord avec les scores des lignes". Nous confrontons les algorithmes aux adversaires qui les ont entrainés, mais aussi aux autres adversaires pour tester le caractère général de nos algorithmes.

7.2.1 Joueur entrainé : le défenseur

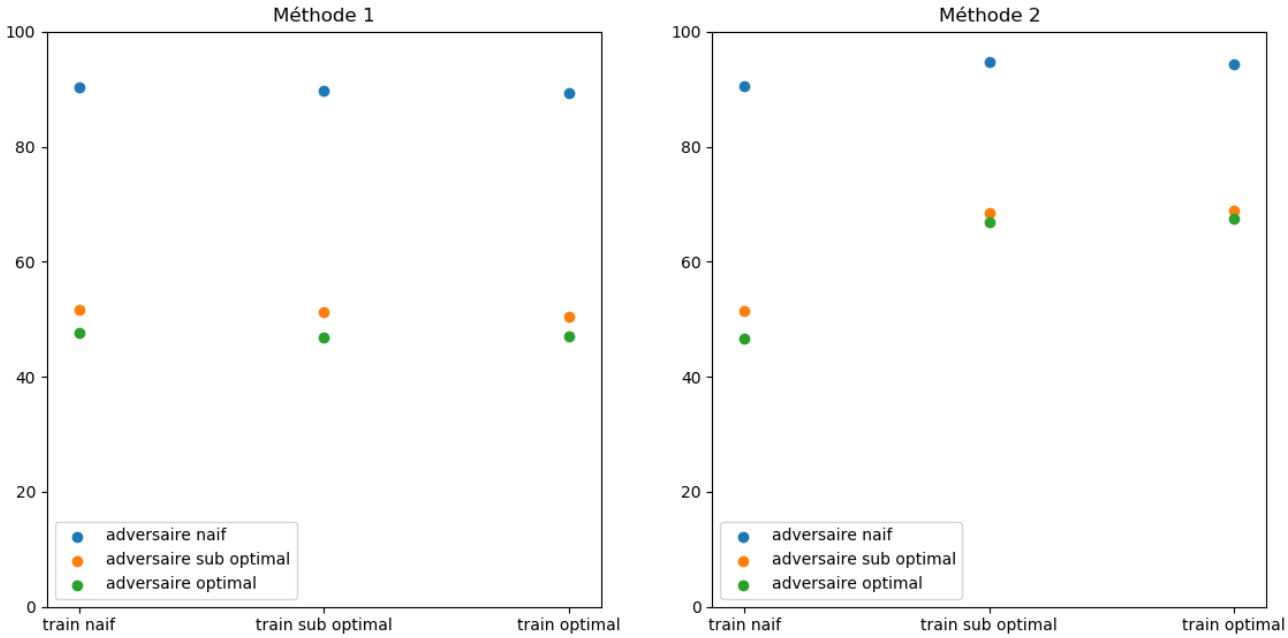


FIGURE 9 – Taux de victoire de l'algorithme de reinforcement learning défenseur : en abscisse l'adversaire contre lequel il a été entrainé, en couleur l'adversaire contre lequel il a été confronté après entrainement

On observe tout d'abord que l'adversaire naïf est toujours le plus simple à battre (quelque soit l'entrainement reçu), puis l'adversaire sous-optimal et enfin l'adversaire optimal. On voit que la méthode 1 est très générale, puisque quelque soit l'entrainement suivi, les taux de victoires contre les trois adversaires sont similaires. Cependant pour les adversaire sous-optimal et optimal, ces taux ne sont pas très élevés (environ 50%). En revanche la deuxième méthode nous permet d'améliorer significativement ces taux, dans le cas de l'entrainement contre l'adversaire sous-optimal et optimal. (pratiquement 70% contre ces mêmes adversaires). En répettant ces simulations, on retrouve toujours ces même tendances, les conclusions font donc réellement sens et sont en cohérence avec ce que nous avions déjà pu observer en comparant les taux de victoires pendant l'entrainement dans la Partie 7.1.

7.2.2 Joueur entrainé : l'attaquant

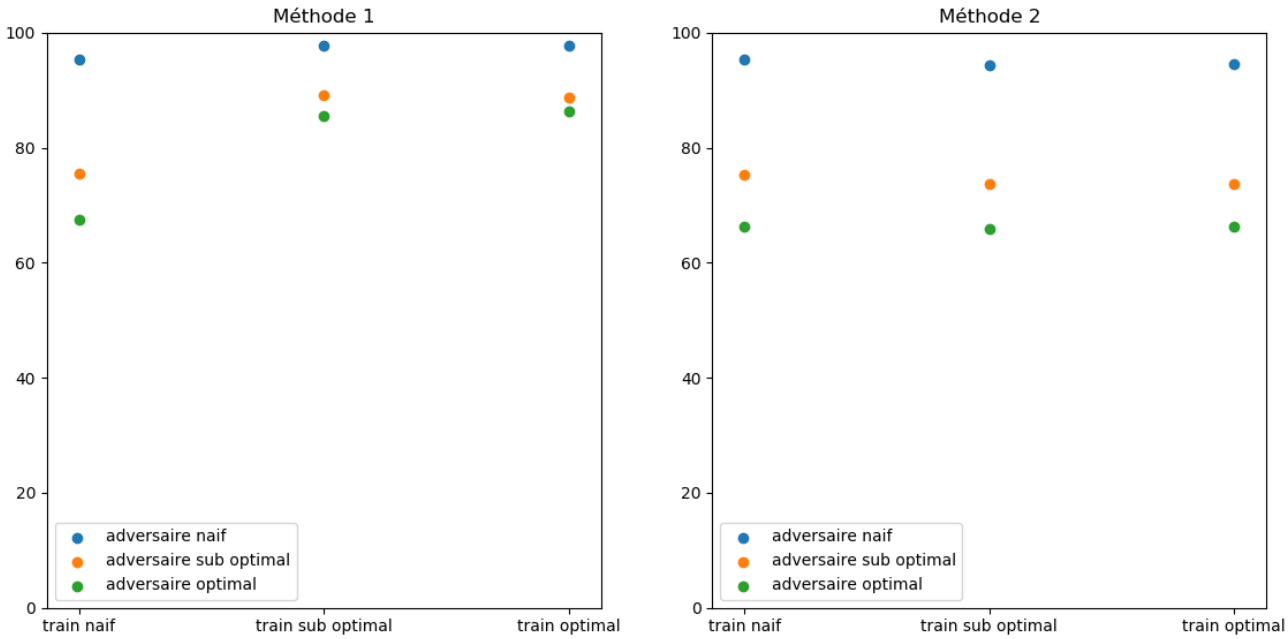


FIGURE 10 – Taux de victoire de l'algorithme de reinforcement learning attaquant : en abscisse l'adversaire contre lequel il a été entrainé, en couleur l'adversaire contre lequel il a été confronté après entrainement

L'ordre des adversaires qu'il est plus facile de battre est toujouts le même, quelque soit l'entrainement. On observe cette fois que la méthode 2 n'améliore pas la stratégie de l'attaquant, bien que celle-ci soit plus générale que la méthode 1 (taux de victoires environ égaux quelque soit l'entrainement). Nous pourrions avoir tendance à dire que la méthode 1 est spécifiquement adapté lors d'un entrainement avec les adversaires sous-optimal et optimal. Cependant en reproduisant l'expérience nous rendons compte d'une grande variabilité concernant cette méthode 1 :

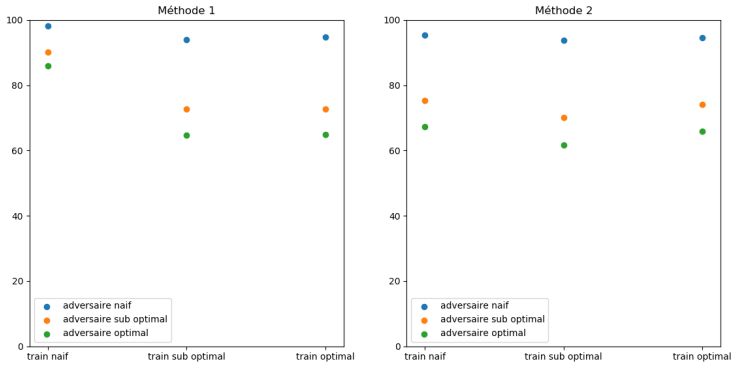


FIGURE 11 – Réplicat 1 de la figure 10

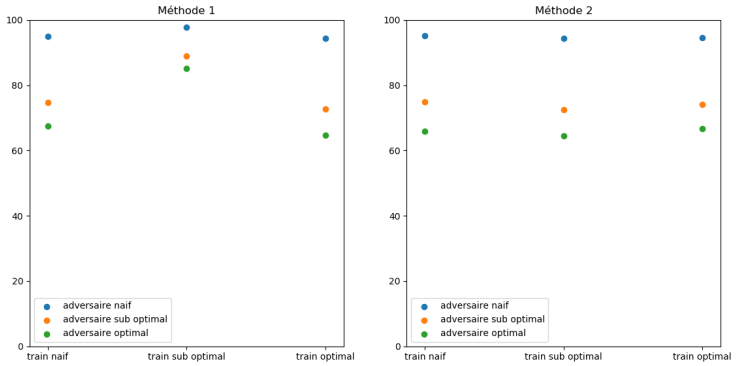


FIGURE 12 – Réplicat 2 de la figure 10

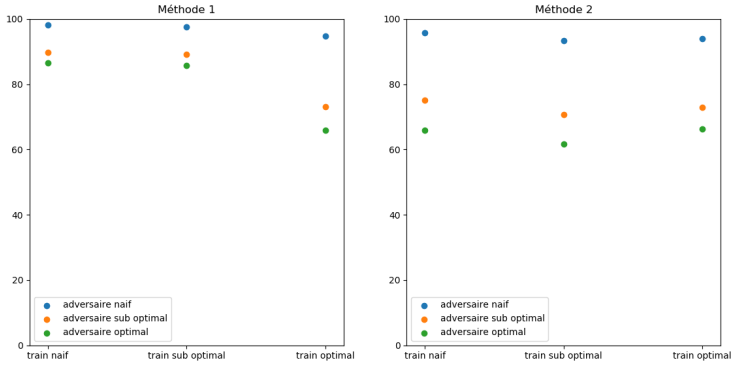


FIGURE 13 – Réplicat 3 de la figure 10

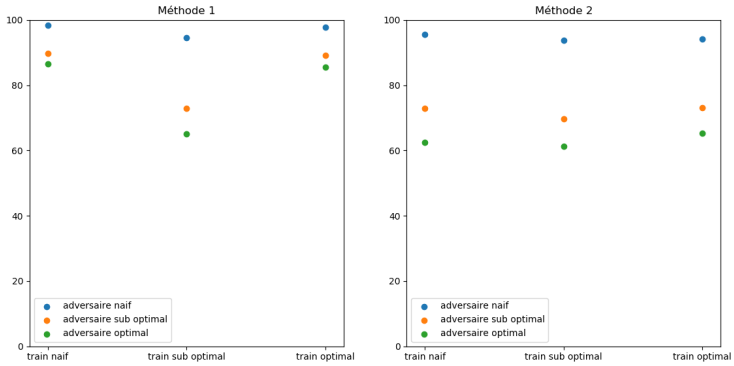


FIGURE 14 – Réplicat 4 de la figure 10

On observe que la méthode 1, bien que fournissant de très bons résultats reste tout de même très variable. Cela pourrait être du à un entrainement peut-être peu adapté où il serait trop simple de gagner et l'algorithme pourrait donc converger vers différentes stratégies acceptables plutôt qu'une stratégie optimale.

8 Entraîner le défenseur avec un réseau de neurones

Dans cette section on a résolu les jeux Erdos-Selfridge-Spencer en entraînant le défenseur à l'aide d'un réseau de neurones et en utilisant la stratégie optimale définie précédemment pour l'attaquant. Pour cela on a construit un échantillon d'apprentissage X contenant $2K + 3$ variables explicatives où les premières $K + 1$ variables correspondent à l'état du jeu, les $K + 1$ variables suivantes à la proposition de l'attaquant et la dernière colonne correspond au choix du défenseur. La variable à expliquer Y vaut 1 si le défenseur a gagné et 0 sinon.

Ainsi la stratégie avec le réseau de neurones consiste d'abord à avoir la probabilité de gagner en fonction de chaque choix du défenseur. Et par la suite on choisit l'option qui maximise la probabilité de gagner.

Ci-dessous les taux de réussite obtenus lorsque le défenseur est entraîné à l'aide d'un réseau de neurones pour différentes valeurs de potentiel :

	0.6	**0.7**	**0.8**	**0.9**	**1.0**	**1.1**	**1.2**
Attaquant sous-optimal	1.0	0.785	1.0	1.0	0.0	0.0	0.0
Attaquant optimal	1.0	1.0	1.0	0.535	0.0	0.0	0.0
Attaquant random	1.0	0.993	0.99	0.995	0.767	0.655	0.452

On observe que pour $\phi(S0) \leq 1$ le taux de réussite du défenseur est très grand, cependant il est faible (voire nul) lorsque le défenseur joue contre l'attaquant optimal ou l'attaquant sous-optimal pour un potentiel initial plus grand que 1 confirmant ainsi la théorie de la stratégie optimale énoncée dans l'article.

Une autre alternative pour résoudre ce problème serait d'utiliser l'algorithme deep Q learning. En effet, il fait partie de l'un des algorithmes utilisés dans cet article. Il combine Q-learning et deep learning. Le Q-learning permet à l'agent de déterminer exactement quelle action effectuer. Il est basé sur la mise à jour de la formule suivante :

$$Q'(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha * \big(r_t + \gamma * max_a Q(s_{t+1}, a) - Q(s_t, a_t)\big)$$

— α : learning rate. Elle détermine dans quelle mesure on modifie la valeur de Q
— $Q(s_t, a_t)$: ancienne valeur du tableau Q.
— r_t : la récompense obtenue en choissisant l'action a_t tout en étant l'état s_t
— γ : facteur d'actualisation
— $max_a Q(s_{t+1}, a)$: la récompense future maximale de l'État s_{t+1}. $Q(s_t, a_t)$: ancienne valeur du tableau Q.

Mais pour un environnement avec un nombre d'états très grand, cet algorithme peut vite devenir incontrôlable à cause de la taille énorme du tableau Q-value. C'est là qu'intervient la méthode du deep Q-learning. En effet, le deep Q-learning utilise un réseau de neurones pour approximer la Q-value. Le réseau prend l'état du jeu en entrée et la valeur Q de toutes les actions possibles est générée en sortie.

Cette méthode donne des résultats très satisfaisants pour résoudre les jeux d'Erdos-Selfridge-Spencer. Cela est dû en partie au fait que le nombre d'états des jeux d'Erdos-Selfridge-Spencer croit de manière exponentielle comme vu précédemment.

9 Théorie vue dans l'article

Dans cette section nous allons rentrer plus en détail dans l'explication des théorèmes de l'article et nous essayerons de compléter les preuves manquantes.

9.1 Définitions et théorèmes vus dans l'article

Faisons un rappel rapide des définitions et théorèmes vus dans l'article. Ces trois premiers théorèmes ont chacun leur preuve complète dans l'article.

Théorème 1 Soit une instance du jeu Attaquant-Défenseur avec K niveaux et N pièces, avec toutes les N pièces commençant au niveau 0. Alors, si $N < 2^K$, le défenseur peut toujours gagner.

Définition 1 La fonction potentielle : Soit $S = (n_0, ..., n_K)$ un état de jeu, nous définissons le potentiel de l'état comme : $\phi(S) = \sum_{i=0}^{K} n_i 2^{-(K_i)}$

Théorème 2 Considérons une instance du jeu Attaquant-Défenseur qui a K niveaux et N pièces, avec des pièces placées n'importe où sur le plateau, et laissez l'état initial être S_0. alors :
(a) Si $\phi(S_0) < 1$, le défenseur peut toujours gagner ;
(b) Si $\phi(S_0) \geq 1$, l'attaquant peut toujours gagner.

Théorème 3 Pour tout jeu Attaquant-Défenseur avec K niveaux, avec comme état de départ S_0 et $\phi(S_0) \geq 1$, il existe une partition A, B telle que $\phi(A) \geq 0,5$, $\phi(B) \geq 0,5$, et pour certains l, A contient des morceaux de niveau $i > l$, et B contient tous les morceaux de niveau $i < l$.

9.2 Théorème 4

La preuve du théorème 4 n'a quant à elle pas été rédigée. Nous allons donc nous en charger. Nous écrirons nos commentaires en italiques au cours de la preuve poru expliquer nos motivations.

9.2.1 Contexte

Au cours de nos essais machine, nous avons testé différents états de départ pour garantir une grande variété d'états vus. Cela nous questionne sur le fait de pouvoir généraliser à travers la distribution de l'état de départ si nous nous entraînons sur une seule distribution.
S'entraîner naïvement sur une distribution d'état de démarrage «facile» (où la plupart des états observés sont très similaires les uns aux autres) entraîne une baisse significative des performances lors du changement de distribution.
En fait, la quantité d'états de départ possibles pour un K donné et un potentiel $\phi(S_0) = 1$ croît de façon sur-exponentielle en le nombre de niveaux K (théorème 4).

9.2.2 Énoncé

Théorème 4 Le nombre d'états de potentiel 1 pour un jeu avec K niveaux croit comme $2^{\Theta(K^2)}$ avec $\frac{K^2}{4} \leq \Theta(K^2) \leq \frac{K^2}{2}$.

9.2.3 Preuve

Soit $S = (s_0, ..., s_K)$ un état de départ tel que $\phi(S) = 1$ et $\mathcal{A} = \{S \mid \phi(S) = 1\}$ l'ensemble des états de départ tel que $\phi(S) = 1$.

Borne supérieure : On a alors $\phi(S) = \sum_{i=0}^{K} s_i 2^{-(K-i)} = 1$. Par conséquent $s_i 2^{-(K-i)} \leq 1 \; \forall \; i = 1, ..., K$, ou encore : $s_i \leq 2^{K-i} \; \forall \; i = 1, ..., K$.

Posons $\mathcal{B} = \{S = (s_0, ..., s_K) \mid s_i \in [\![0, 2^{K-i}]\!], \; \forall i \in [\![0, K]\!]\}$. On a bien $\mathcal{A} \subset \mathcal{B}$.

Le cardinal de $\mathcal{B}$ est $\prod_{i=0}^{K}(2^{K-i} + 1)$ (produit des nombres de choix pour chaque s_i).

$$\begin{aligned}
\prod_{i=0}^{K}(2^{K-i} + 1) &\approx \prod_{i=0}^{K} 2^{K-i} \text{ car on cherche une borne (ie à l'infini)} \\
&= \prod_{i=0}^{K} 2^{i} \text{ par changement de variable} \\
&= 2^{\sum_{i=0}^{K} i} \\
&= 2^{\frac{K(K+1)}{2}} \\
&\approx 2^{\frac{K^2}{2}}
\end{aligned}$$

Puisque $\mathcal{A} \subset \mathcal{B}$, on a aussi $Card(\mathcal{A}) \leq Card(\mathcal{B}) \approx 2^{\frac{K^2}{2}}$. On a donc bien le nombre d'état de potentiel 1 borné supérieurement par $2^{\frac{K^2}{2}}$.

Borne inférieure (cas simple) : Prenons d'abord le cas où K est une puissance de deux (ie $log(K)$ est un entier).

L'article propose de poser $\mathcal{C} = \{S = (s_0, ..., s_K) \mid s_{j-1} 2^{-j+1} + s_j 2^{-j} = \frac{1}{K}, \; \forall j \in [\![log(K) + 1, K]\!]$ *avec les j pairs}. Cependant, après un certain nombre de calculs réalisés, il se trouve que la preuve est incohérente en prenant cet ensemble (car la condition* $\mathcal{C} \subset \mathcal{A}$ *qui nous permettrait de conclure n'est pas correcte.*

Posons $\mathcal{C} = \{S = (s_0, ..., s_K) \mid s_0 = 0, \; s_{2j-1} 2^{-(K-2j+1)} + s_{2j} 2^{-(K-2j)} = \frac{2}{K}, \; \forall j \in [\![1, K/2]\!]\}$
Soit $S \in \mathcal{C}$,

$$\begin{aligned}
\phi(S) &= \sum_{j=0}^{K} s_j 2^{-(K-j)} \\
&= \sum_{j=1}^{K/2} s_{2j-1} 2^{-(K-(2j-1))} + s_{2j} 2^{-(K-2j)} \\
&= \sum_{j=1}^{K/2} \frac{2}{K} \\
&= 1
\end{aligned}$$

Donc $S \in \mathcal{A}$, donc $\mathcal{C} \subset \mathcal{A}$.

Calculons le cardinal de $\mathcal{C}$:
Soit $\forall j \in [\![1, K/2]\!]$, on a $2^{K-log(K)-2j+1} + 1$ possibilités d'avoir $s_{2j-1} 2^{-(K-2j+1)} + s_{2j} 2^{-(K-2j)} = \frac{2}{K}$

(s_{j-1} et s_j étant des entiers naturels). Car :

$$\begin{aligned} s_{2j-1}2^{-(K-2j+1)} + s_{2j}2^{-(K-2j)} &= \frac{2}{K} \\ 2^{-(K-2j)}(\frac{s_{2j-1}}{2} + s_{2j}) &= \frac{2}{K} \\ \frac{s_{2j-1}}{2} + s_{2j} &= 2^{(K-2j)}\frac{2}{K} \\ \frac{s_{2j-1}}{2} + s_{2j} &= 2^{K-2j-log(K)+1} \end{aligned}$$

On a donc $2^{K-log(K)-2j+1}+1$ possibilités pour choisir s_{2j} et s_{2j-1} devient alors fixée.

On a donc :

$$\begin{aligned} Card(\mathcal{C}) &= \prod_{j=1}^{K/2}(2^{K-log(K)-2j+1}+1) \\ &\geq \prod_{j=1}^{K/2} 2^{K-log(K)-2j+1} \\ &= 2^{\sum_{j=1}^{K/2} K-log(K)-2j+1} \\ &= 2^{\frac{K^2}{2}-\frac{Klog(K)}{2}+\frac{K}{2}-2\sum_{j=1}^{K/2} j} \\ &= 2^{\frac{K^2}{2}-\frac{Klog(K)}{2}+\frac{K}{2}-\frac{K}{2}(\frac{K}{2}+1)} \\ &= 2^{\frac{K^2}{2}-\frac{Klog(K)}{2}-\frac{K^2}{4}} \\ &= 2^{\frac{K^2}{4}-\frac{Klog(K)}{2}} \end{aligned}$$

Puisque $\mathcal{C} \subset \mathcal{A}$, on a aussi $2^{\frac{K^2}{4}} \leq Card(\mathcal{C}) \leq Card(\mathcal{A})$. On a donc bien le nombre d'état de potentiel 1 borné inférieurement par $2^{\frac{K^2}{4}}$.

Borne inférieure (cas général) : *L'esquisse de démonstration de l'article donne un résultat seulement dans le cas simple. Essayons d'aller plus loin avec une extention de la preuve au cas général :*

Regardons maintenant le cas où K n'est pas forcément une puissance de deux. On peut tout de même prendre $m \in \mathbb{N}$ tel que $2^m \leq K \leq 2^{m+1}$.
Posons $\tilde{K} = 2^m$. On a alors : $\tilde{K} \leq K \leq 2\tilde{K}$.

Notre $\mathcal{A} = \{S \mid \phi(S) = 1\}$ dépend bien entendu de K, on peut donc renommer $\mathcal{A}$ en $\mathcal{A}_K$ pour plus de clareté dans la suite de la démonstration.
On a d'ailleurs : $k \longrightarrow Card(\mathcal{A}_k)$ qui est une fonction croissante.

$$\begin{aligned} Card(\mathcal{A}_K) &\geq Card(\mathcal{A}_{\tilde{K}}) \\ &\geq 2^{\frac{\tilde{K}^2}{4}} \text{ d'après le cas simple} \\ &\geq 2^{\frac{K^2}{16}} \text{ car } K \leq 2\tilde{K} \text{ donc } \frac{K^2}{4} \leq \tilde{K}^2 \end{aligned}$$

Références

[1] M. Raghu et al. "Can Deep Reinforcement Learning Solve Erdos-Selfridge-Spencer Games ?" In : *6th International Conference on Learning Representations* (2018). URL : `https://arxiv.org/pdf/1711.02301.pdf`.