Apprentissage automatique
Le jeu de Nim

par Camille Leplumey et Geoffrey Spaur

03 Mai 2017

Contents

1 Mode *easy*

Le CPU joue aléatoirement sans prendre de décisions. Il est donc possible qu'il perde bêtement. Le CPU jouera entre 1 et 3 bâtons comme indiqué dans les règles du jeu. Il lui sera cependant interdit de jouer 3 bâton alors qu'il en reste 2. Ainsi il jouera automatiquement 1 bâton, s'il reste 1 bâton en jeu. Et il jouera, avec 50% de chance, 1 ou 2 bâtons quand il restera 2 bâtons en jeu.

2 Mode *medium*

Le mode medium ne constitue en rien à de l'apprentissage. En effet nous codons les derniers coups que doit jouer le CPU. Le résultat sera donc identique à chaque parties sur les derniers coups et aléatoire sur les autres coups.

Dans notre cas le CPU jouera:

- 3 s'il reste 4 bâtons
- 2 s'il reste 3 bâtons
- 1 s'il reste 2 bâtons

Ormis c'est 3 cas spécifiques, le CPU jouera aléatoirement. Il est évident que le CPU jouera 1 s'il reste 1 bâton au même titre que dans le mode *easy*.

3 Mode *hard*

Ici en entraînant le CPU contre lui même en mode *hard*, il va apprendre la stratégie gagnante par lui même sans aide extérieur. C'est un apprentissage **non supervisé**. Nous utilisons cette méthode car nous pouvons l'automatiser facilement.

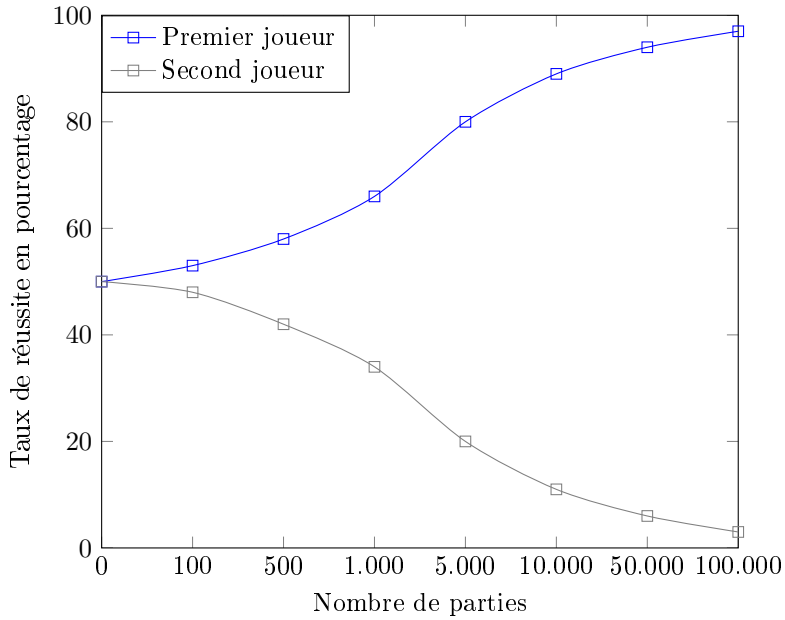
3.1 Protocole des réalisations des graphiques

Nous avons réalisé quelques statistiques concernant les différentes façons de jouer. Chaque points des courbes est une moyenne de 50 parties jouées. Chaque partie comprend une phase d'apprentissage, où le réseau de neurones se construira. Nous avons donc échelonné cette phase d'apprentissage pour réaliser nos graphiques. En conclusion le CPU apprendra sur 500 parties, nous effectuerons cet apprentissage 50 fois afin d'établir une moyenne de réussite. Nous procéderons de manière identique avec une phase d'apprentissage à 1.000 parties et ainsi de suite jusqu'à arriver à une phase d'apprentissage à 100.000 parties.

Il est important de remarquer nous notre échelonnage est **non proportionnel**. Il est donc possible de voir des légers paliers sur les différentes courbes des graphiques.

3.2 Graphique de réussite

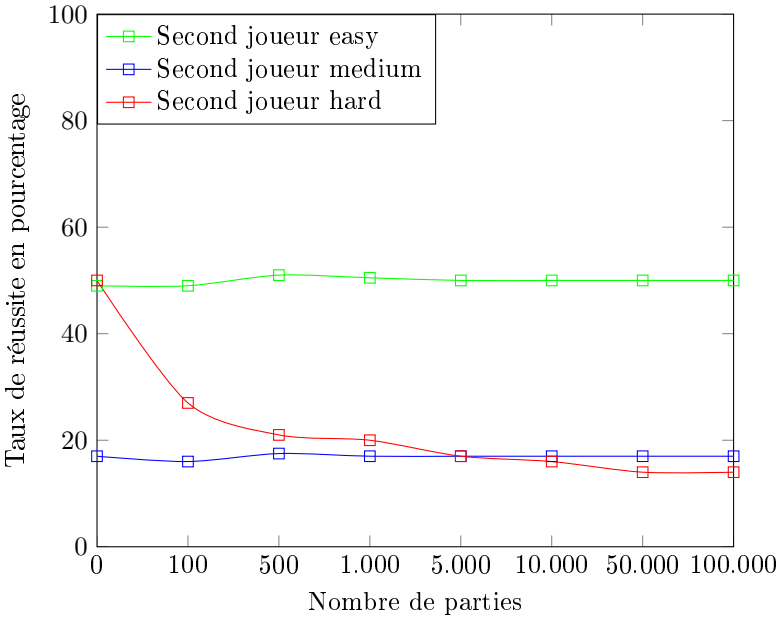
Évolution du taux moyen de réussite entre 2 joueurs **hard** sans changement de rôle



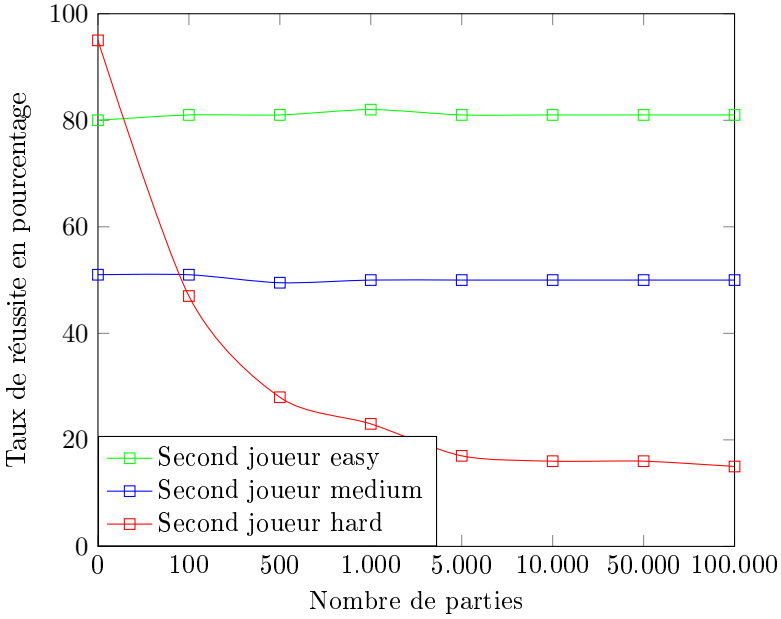
Nous voyons ici qu'en jouant 800 parties, le CPU obtiendrais une moyenne de 63% de réussite. De plus, nous pouvons affirmer qu'à partir de 50.000 parties jouées, le CPU ne fera quasiment plus d'erreurs.
En théorie les deux courbes présentent sur ce graphique devrait être symétrique. Cependant nous avons effectué des moyennes avec différent set de jeu. Cela explique l'asymétrie entre nos courbes.

3.3 Statistiques selon le mode de joueur

Évolution du taux moyen de réussite d'un joueur **easy** sans changement de rôle



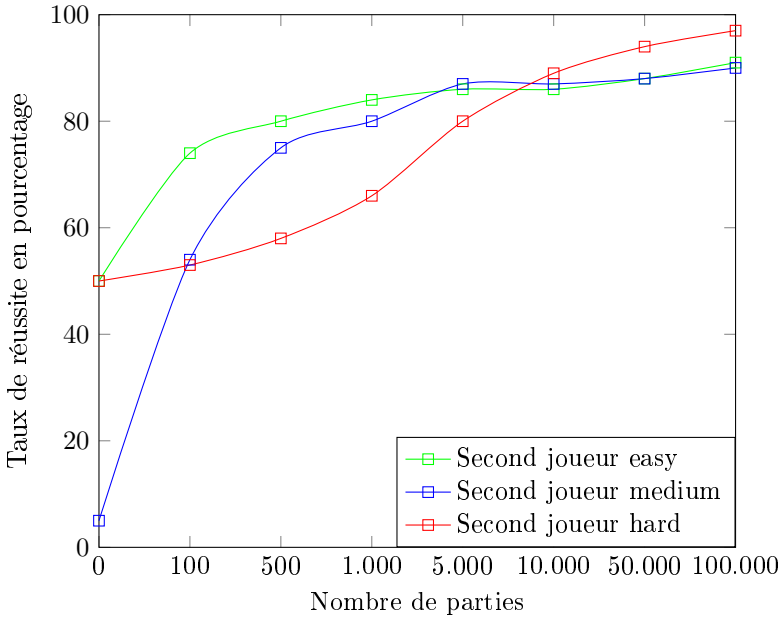
Évolution du taux moyen de réussite d'un joueur **medium** sans changement de rôle



Sans surprise sur ces graphiques, nous ne voyons aucune évolution du taux de réussite contre les joueurs *easy* et *medium*, car ces joueurs ne changent pas leur stratégie au cours du temps. Nous pouvons cependant constater de légère variation dans les premières parties due à un affinement

de la moyenne.
Il est à noté que le joueur *hard* apprend plus vite en commeçant en second contre un joueur *easy* que contre un joueur *medium*. Cela souligne les différentes façons de jouer des joueurs *easy* et *medium*. Pour rappelle le joueur *medium* joue aléatoirement tout comme le joueur *easy*, sauf pour le dernier coup, où le joueur *medium* tentera de gagner.

Évolution du taux moyen de réussite d'un joueur **hard** sans changement de rôle



De loin le schéma le plus intéressant. En effet dans la phase 100 à 10.000 parties, nous pouvons voir que le joueur *hard* commençant en premier, à un plus haut taux de réussite contre un joueur *medium*, que contre un second joueur *hard*. Cela s'explique par le fait que le joueur *medium* joue aléatoirement alors que le second joueur *hard* essaye de découvrir la stratégie gagnante, tout comme le premier joueur *hard*. Le premier joueur *hard*, à partir des 10.000 parties perfectionne tellement son réseau que le second joueur *hard* ne peut plus apprendre car il enchaînera les défaites.
Il n'est donc pas surprenant de voir le taux de réussite contre un joueur *medium* dépasser le taux de réussite contre un joueur *hard* car le joueur *medium* ne peut pas se tromper sur le dernier coup contrairement au second joueur *hard*.

3.4 Conclusion

Pour conclure, si nous voulons jouer contre le CPU en mode *hard*, il sera toujours possible de gagner. En effet le CPU choisi aléatoirement parmi ces connexions en favorisant les connexions à fort poids. Néanmoins la chance que le CPU choisisse une connexion perdante sera toujours non nul, car les poids initiaux des connexions sont non nul et ne décroissent jamais. Même si le CPU est bien entraîné, avec de la patience, il sera toujours possible de gagner une partie contre lui.

4 Correction apportée

Suite à nos différent tests, nous avons généré et testé un réseau entraîné à jouer en second contre un CPU *medium*. Durant notre phase de test, nous avons importé son réseau afin de jouer contre lui (en le plaçant dans le même rôle, c'est-à-dire qu'il va jouer en second). Nous nous sommes aperçus que le CPU n'arrivait pas à retrouver la stratégie gagnante en fin de partie. La raison était simple, ce réseau n'avait aucune connexion de poids supérieur à *BASE_WEIGHT* à partir du neurone 6. Pensant à un coup du sort, nous avons relancé l'algorithme sans succès. Il est à noter que ce problème ne survenait que contre un CPU *medium*.

Nous avons donc dans un premier temps changé d'algorithme dans *playHard* afin de vérifier l'intégrité du réseau de neurone. C'est un algorithme naïf qui sauvegardait et récompensait uniquement les connexions de distance 3. Dans ce cas toutes les connexions s'établissaient correctement. Seulement les performances devait être revu à la baisse. Et cela pour une raison simple, en supposant que le CPU applique la stratégie gagnante: dans le cas où nous sommes fasse à 13 bâtons, quelques soit le nombre de bâton que nous jouions, le CPU pourra toujours rendre le main avec 9 bâtons en jeu. Cependant avec notre algorithme naïf, ne récompensera qu'une des connexions suivantes (12->9), (11->9) ou (10->9); alors qu'avec l'algorithme proposé, le CPU récompense la connexion (13->9) pour les 3 cas de figure ce qui lui donne plus de poids.

Ayant une perte de performances, nous avons décidé de reprendre l'algorithme qui nous était suggérer. Pour éliminer toutes incertitudes liées à l'aléatoire, nous avons calculer le nombre de combinaisons possible: 5768, puis le nombre de combinaisons quand le second joueur passe par le neurone 6 et gagne la partie: 324. Après avoir fait tourner l'algorithme plus de 10 million de fois sans jamais avoir trouver de réseau avec au moins une connexion récompensée sur le neurone 6, nous pouvons affirmer qu'il y a bien une erreur dans l'algorithme.

En étudiant précisément les différents jeu exécuté, pour avons remarqué quelques similarités intéressantes. En effet quand si on rend 6 bâtons au CPU *medium*, alors il jouera toujours 1 bâton. Voici le code de *playRandom*:

```
return random.randint(1, (sticks%3)+1)
```

Or s'il reste 6 bâtons, *sticks* vaut 6, 6 modulo 3 vaut 0 et 0 plus 1 vaut 1. Donc s'il reste 6 bâtons *playRandom* renverra toujours 1, ce qui ai incohérent. C'est la raisons pour laquelle le CPU *medium* réussissait toujours à reprendre la stratégie gagnante avec 6 bâtons, et que par conséquent nous n'enregistrions pas de connexions avec notre réseau de neurone. Voici la correction apportée:

```
if sticks >= 3:
    return random.randint(1, MAX_DIST)
else:
    return random.randint(1, (sticks % MAX_DIST))
```

MAX_DIST équivaut à la distance maximal, dans notre cas *MAX_DIST* vaut 3.

5 Proposition

5.1 Agrégation

Nous avons remarqué que notre joueur *hard* est quasiment imbattable s'il commence en premier, résultant du fait que nous l'avons entraîné à jouer en premier. Seulement nous pouvons remarquer que si ce joueur commence en second sans entraînement préalable alors il sera aussi fort qu'un joueur en mode *easy*.

Pour améliorer le joueur *hard*, nous avons donc pensé à créer 2 joueurs *hard*. Le premier jouerai en premier contre en autre mode *hard*. Le second jouera en second contre un *medium*. Les 2 joueurs résultant de cette apprentissage devrons être alors agrégé afin que le joueur résultant soit capable de s'adapter à n'importe quelle situation, c'est-à-dire s'il commence en premier ou non.

Il sera cependant toujours possible de battre le réseau neuronal résultant, car le jeu de Nim admet une stratégie gagnante. Donc si vous connaissez cette stratégie et que vous commencer la partie toute en appliquant la stratégie gagnante, aucun réseau de neurones ne pourra vous battre.

Mais ici le but de notre réseau neuronal résultant est de reprendre la stratégie gagnante dans le cas où le joueur adverse commet une erreur.

Remarque Cette amélioration a été développé et est utilisé dans le script *ExportNetwork.py*.

5.2 Réduction des poids

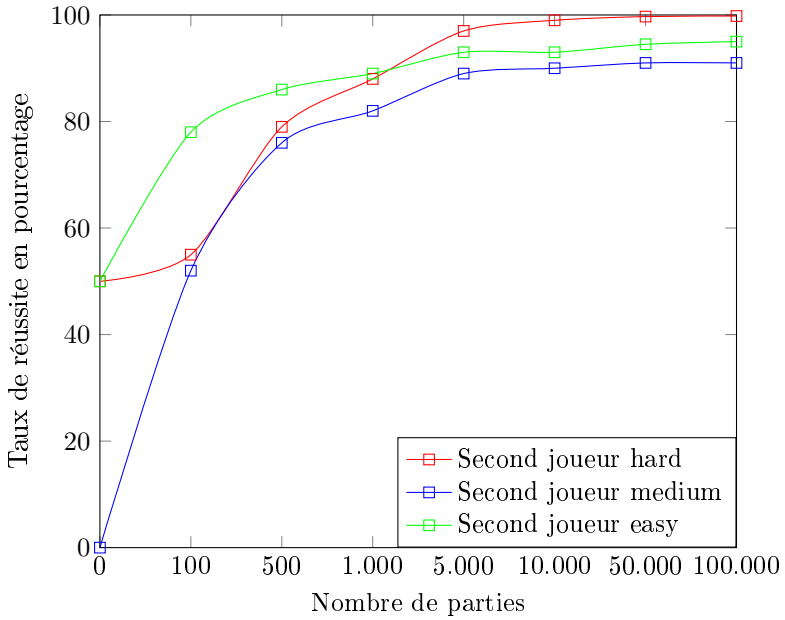
Une autre possibilité afin d'affiner notre réseau de neurone serait de *punir* le chemin menant à la perte d'une partie. Cette *punition* se traduirait par la réduction des poids des connexions empruntées par le chemin perdant. Il est cependant important de remarquer que si le poids d'une connexion tombe à 0, alors la connexion disparaît. Cela peut donc devenir problématique car le réseau neuronal pourrait s'enfermer dans des erreurs sans pouvoir se corriger.

Il est donc important que le poids d'une connexion ne tombe jamais à 0. Afin que le réseau puisse en permanence évaluer tout les cas possibles. C'est un processus obligatoire dans la correction d'erreurs. Cela implique - encore une fois - que le réseau ne pourra pas avoir un taux de réussite strictement égal à 100%.

Remarque Cette amélioration a été développé et est utilisé dans le script *ExportNetwork.py*.

5.2.1 Graphique de réussite avec réduction du poids

Évolution du taux moyen de réussite d'un joueur **hard** jouant en premier, avec réduction du poids



Évolution du taux moyen de réussite d'un joueur **hard** jouant en second, avec réduction du poids