Implémentation et analyse de programmes de prédiction dans le jeu “Pierre-Feuille-Ciseaux” avec un réseau de neurones artificiels et une chaîne de Markov.

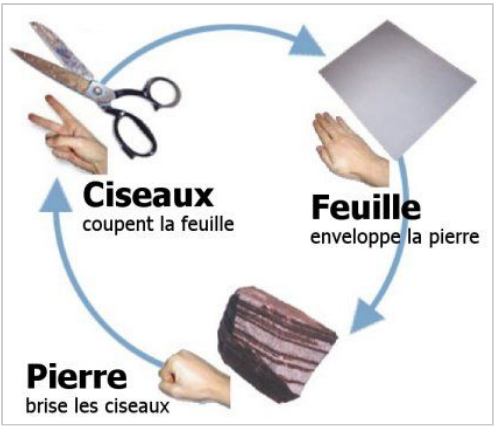
Equipe de suivi : M. Thierry Hamon et Mme Sophie Toulouse
Clients : M. Nadi Tomeh et M. Olivier Bodini
Equipe de projet : Sara Abbih, Badre-Dine Aloui, Wael Antit, Badreddine Farah et Cheikh Kébé

Informatique 2ème année, Conduite et gestion de projet, Sup Galilée, Université Sorbonne Paris Nord, 2019/2020.

Résumé :
Bien que le jeu Pierre-Feuille-Ciseaux soit généralement un jeu de pure chance dans lequel les coups sont parfois sélectionnés au hasard, un joueur peut prédire le prochain coup de l'adversaire en fonction de ses coups précédents. Des milliers de rounds pour des centaines de parties ont été joués contre l’ordinateur par nous-mêmes et on a utilisé ces données pour analyser les résultats dans une chaîne de Markov en variant l’ordre et un réseau de neurones. On a constaté que la machine peut gagner presque 80% des parties avec un taux de prédiction de près de 45%.

I. Introduction

Les règles du jeu Pierre-Feuille-Ciseaux sont simples : à chaque manche, deux adversaires choisissent simultanément “Pierre”, “Feuille” ou “Ciseaux”. “Pierre” bat “Ciseaux”, “Ciseaux” bat “Feuille” et “Feuille” bat “Pierre”.



Si les coups sont choisis complètement au hasard, aucun joueur ne devrait avoir un avantage et les probabilités que l'un des joueurs gagne ou que le round se termine sur un nul sont toutes équiprobables.

Il est impossible d'obtenir un avantage sur un adversaire vraiment aléatoire. Cependant, en exploitant les coups d'adversaires intrinsèquement non aléatoires, il est possible d'obtenir un avantage significatif [1] [2].
En effet, les joueurs ont tendance à être non aléatoires.

De ce fait, après plusieurs tours de jeu, un programme peut apprendre les probabilités de transition des coups précédents de l'adversaire pour prédire le coup suivant. Cela suggère que tout schéma qu'un joueur développe peut être appris par la machine et utilisé pour gagner la partie.

Dans ce rapport, nous proposons de vérifier cette hypothèse dans le cadre de programmes qui exploitent la stratégie du joueur pour prédire son coup prochain et gagner le plus de parties possibles et nous avons conduit des expériences pour confirmer les résultats.

II. Les chaînes de Markov

1. Principe d'une chaîne de Markov

Le premier programme que nous avons implémenté est un modèle basé sur une "*chaîne de Markov*". Le principe de ce modèle est que l'état actuel ne dépend que d'un nombre fini d'états antérieurs [3].

Le plus simple processus de Markov est une chaîne de Markov du 1er ordre où la prédiction du futur est entièrement contenue dans l'état présent du processus et n'est pas dépendante des états antérieurs :

$$P(X_{n+1} = k \mid X_1 = k_1, X_2 = k_2, \ldots, X_n = k_n) = P(X_{n+1} = k \mid X_n = k_n)$$

où les $\mathbf{X_i}$ sont des variables aléatoires représentant le fait d'avoir joué un coup.

Les chaînes de Markov peuvent être généralisées aux cas de dépendance à court terme, en tenant compte des états passés récents de la chaîne. Par exemple, pour une chaîne de Markov du 2e ordre, des combinaisons de deux coups sont utilisées pour prédire le prochain :

$$P(X_{n+1} = k \mid X_1 = k_1, X_2 = k_2, \ldots, X_n = k_n) = P(X_{n+1} = k \mid X_{n-1} = k_{n-1}, X_n = k_n)$$

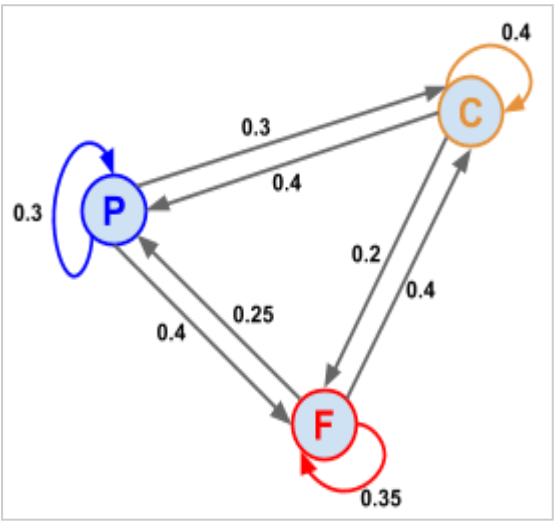
De manière générale, une chaîne de Markov du $\mathbf{m}^{e}$ ordre considère les **m** précédents coups pour prédire le choix futur du joueur :

$$P(X_{n+1} = k \mid X_1 = k_1, X_2 = k_2, \ldots, X_n = k_n) = P(X_{n+1} = k \mid X_n = k_n, X_{n-1} = k_{n-1}, \ldots, X_{n-m-1} = k_{n-m-1})$$

avec **n** > **m** et où **m** est un nombre fini.

2. Méthodologie

Dans notre cas, le processus a trois états possibles **P**, **F** et **C** qui correspondent respectivement aux coups “Pierre”, “Feuille” et “Ciseaux”.
Supposons que nous sommes maintenant dans l'état **P** (on vient de jouer “Pierre”). Quelle est la probabilité de rester dans l'état **P** (rejouer “Pierre”)? Aussi, quelle est la probabilité d'aller à l'état **F** ou **C** ?
Concernant une chaîne de Markov avec trois états possibles, la somme de ces probabilités doit être égale à 1. De manière adéquate, il y aurait trois probabilités pour l'état actuel **P**.
On peut voir un exemple du mécanisme sur l'image ci-contre.



Pour implémenter une chaîne de Markov, on utilise une “*matrice de transition*” pour calculer les probabilités de transition.

	P	**F**	**C**
PP	0.3	0.4	0.3
PF	0.25	0.35	0.4
PC	0.4	0.2	0.4
FF	0.3	0.4	0.3
FP	0.3	0.4	0.3
FC	0.25	0.35	0.4
CC	0.4	0.2	0.4
CP	0.3	0.4	0.3
CF	0.25	0.35	0.4

Exemple de matrice de transition d'une chaîne de Markov du 2e ordre

Chaque état actuel de l'espace d'états est inclus sous forme de ligne et les états de sortie sous forme de colonne, et chaque cellule de la matrice comporte la probabilité de passer de l'état de la ligne à l'état de la colonne.
Au départ, toutes les chances sont équiprobables donc égales à 1/3 et toutes les séquences (état actuel-état de sortie) par exemple ont la même fréquence. Au fur et à mesure du déroulement de la partie, les fréquences observées sont

incrémentées, ce qui augmenterait leur probabilité. Un exemple vaut mieux qu'un long discours :

Supposons que le joueur effectue une séquence comme ceci :

P F C P F C F .....

Pour une chaîne de Markov du 2e ordre, si on prend chaque triplet de cette séquence pour les associer à une fréquence, on aura une chose de ce genre, d'abord pour les séquences :

PFCPFCF	P**FCP**FCF	PF**CPF**CF	PFC**PFC**F	PFCP**FCF**

Et pour les fréquences :

Séquence	PFC	FCP	CPF	FCF
Fréquence	2	1	1	1

Ce qui veut dire, si le joueur joue "Pierre" puis "Feuille" de suite, il y a plus de chance que son prochain coup soit "Ciseau" qu'autre chose.
De ce fait, le modèle s'adaptera donc plus rapidement aux changements de comportement des adversaires et pour la prédiction, il choisit le mouvement avec la probabilité la plus élevée.

Nous allons tester notre programme sur un jeu de données contenant des centaines de parties jouées par nous-mêmes. Nous allons essayer de varier l'ordre de notre processus de Markov pour mieux voir les performances de notre algorithme à savoir la vitesse de détection d'un changement de stratégie de l'utilisateur, le pourcentage de victoires de la machine mais aussi les taux de réussite de prédiction.

3. Résultats et analyse

Une fois la phase d'implémentation terminée, on a souhaité tester le fonctionnement du programme à commencer par varier l'ordre de notre processus de Markov pour évaluer le taux de prédiction moyen et la moyenne de victoires de l'algorithme. Nous avons codé des fonctions d'affichage des résultats, puis utilisé celles de la librairie *Matplotlib* de Python pour la génération des graphiques.

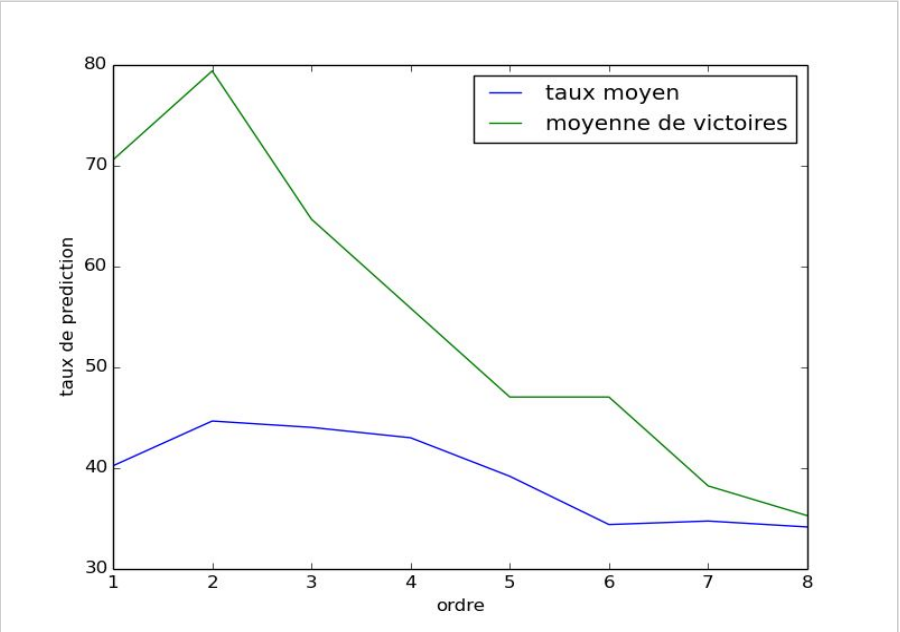


Fig. 1 : Courbes des taux moyen de prédiction et de victoires en fonction de l'ordre

Ce graphique ci-dessus nous présente les résultats des tests réalisés sur nos jeux de données. Les données sont obtenues ainsi :

- pourcentage moyen de victoires :

$$\frac{nombre\ de\ parties\ gagnées}{nombre\ total\ de\ parties}$$

- taux de réussite de prédiction moyen :

$$\frac{nombre\ de\ bonnes\ prédictions}{nombre\ total\ de\ prédictions}$$

On peut voir que le taux de prédiction réussie par le programme est assez proportionnel au pourcentage de victoires, ce n'est pas vraiment une surprise. On peut, cependant, noter qu'on a obtenu un taux de prédiction plus élevé avec une chaîne de Markov du deuxième ordre (environ 45%), de même que le nombre de victoires de la machine paraît également beaucoup plus élevé par rapport aux chaines d'autres ordres. Cela peut s'expliquer par le fait qu'un processus de

deuxième ordre est plus susceptible, non seulement de reconnaître la stratégie d'un joueur, mais surtout de repérer un changement de stratégie.

On peut voir un peu plus en détails les pourcentages de victoires ou de nuls du programme selon la variation de l'ordre.

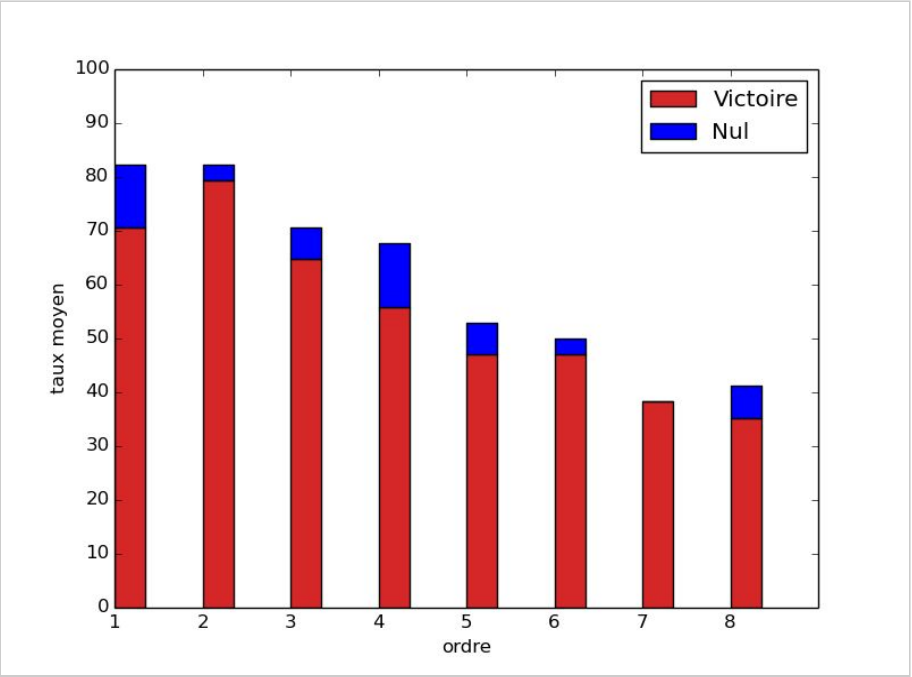


Fig. 2 : Histogramme des taux de victoires et de nuls en fonction de l'ordre

La machine est capable de battre le joueur, avec un modèle du second ordre, avec un pourcentage de victoires de presque 80% des parties jouées. Le site rpscontest.com organise des compétitions de programmation pour voir qui peut développer l'algorithme le plus efficace contre les humains. Il s'est avéré que l'algorithme le plus performant donne un pourcentage de victoires entre 70 et 80% [1].

En somme, on peut voir que les 4 premiers ordres sont les plus performants pour prédire le coup du joueur car ils sont mieux adaptés aux variations de stratégies et on a de plus remarqué que plus nous augmentons l'ordre de la chaîne de Markov, plus la prédiction devient imprécise.

On peut aller plus loin dans nos expériences en analysant de plus près les évolutions du taux de prédiction dans une partie donnée, mais en les testant uniquement sur des processus du premier au quatrième ordre.

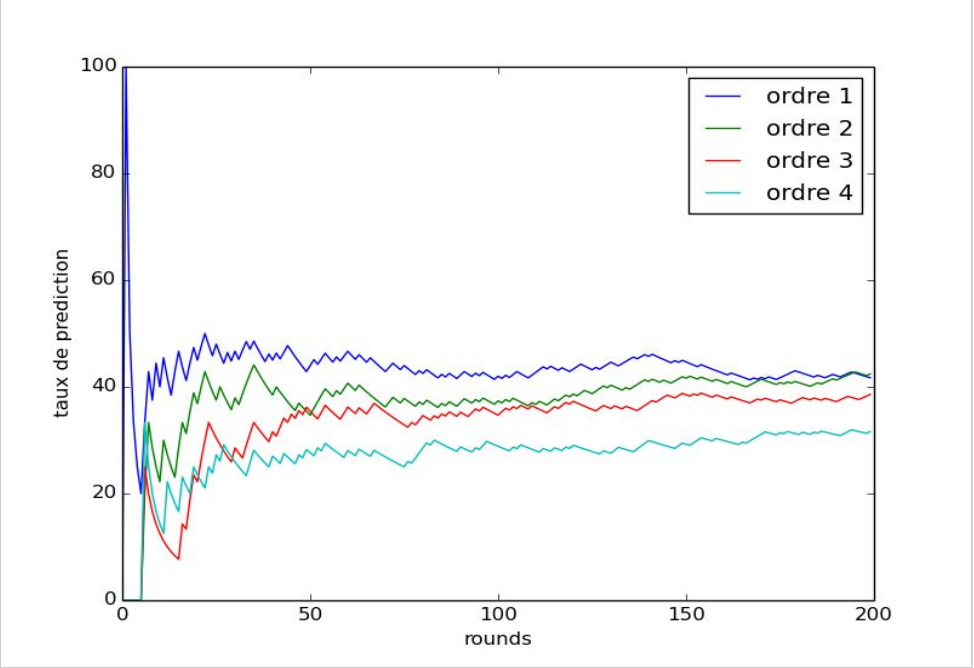


Fig. 3 : Evolution du taux de réussite de prédiction dans une partie donnée

On remarque sur le graphique 3, qu'avec la partie de 200 rounds prise en compte pour le test, une chaîne de Markov du 1er ordre fournit une plus bonne prédiction que celle du second ordre globalement. Mais, on observe, à partir du 190e round, une nette montée de la courbe du processus du deuxième ordre et une baisse du premier. Pour les modèles restants, le taux de prédiction est relativement bas, même si le processus du 3e ordre s'est rapproché un peu du 2e ordre au milieu de la partie.

On va, étudier ici, la vitesse de détection de changement de stratégie de notre algorithme avec une séquence simple et testée sur des chaines de 1er au 8e ordre. En gros, on veut connaître ici, après combien de coups en moyenne notre programme a détecté le changement de tactique de l'utilisateur.
Si, par exemple, nous choisissons de jouer "Pierre", puis "Feuille", puis "Ciseaux", puis nous répétons cette séquence quelques fois (4 ici), nous allons changer de stratégie subitement et choisir de jouer "Feuille", puis "Pierre", puis "Ciseaux" de suite et observer la détection de changement par le programme et la modification de son jeu.

Les résultats obtenus sont retranscrits dans le tableau ci-dessous où ***ordre*** représente la chaîne de Markov testée avec l'ordre correspondant et ***nombre de coups***, l'effectif de coups après lequel le programme a détecté et modifié son jeu.
On considère, ici, que l'ordinateur a détecté le changement s'il obtient une suite de gains.

Ordre	1	2	3	4	5	6	7	8
Nombre de coups	15	6	5	5	6	8	8	9

Grâce au tableau, on peut maintenant représenter les données sur un graphique, ce qui nous donne :

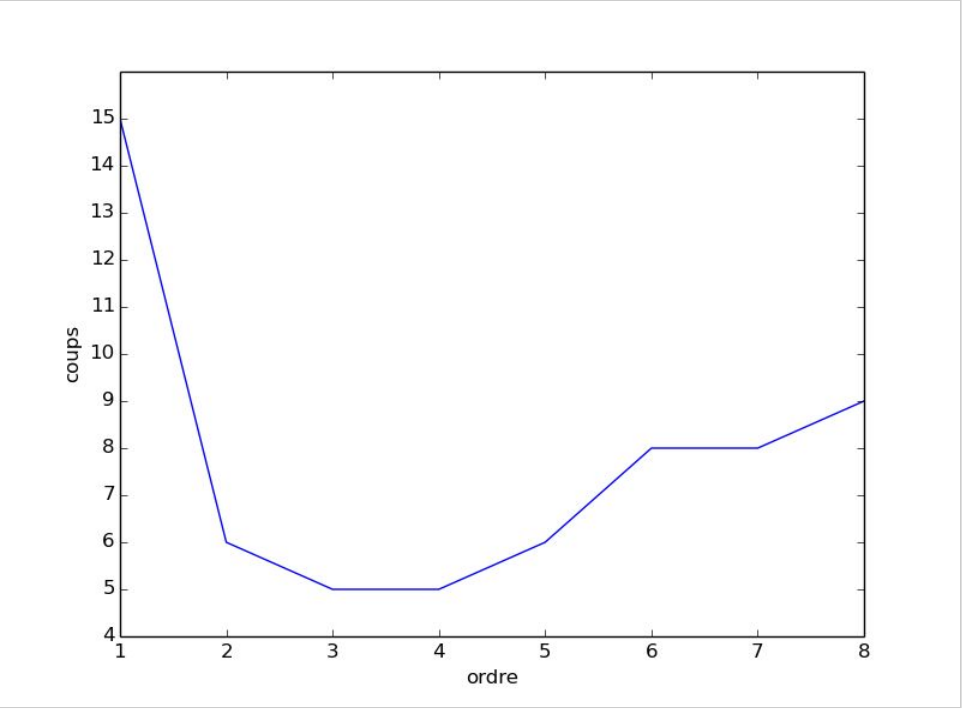


Fig. 4 : Evolution de la détection de changement de stratégie en fonction de l'ordre

On observe que, de manière paradoxale, une chaîne de Markov du 1er ordre met plus de "temps" à détecter le changement de stratégie que les autres modèles. L'explication vient du fait que les fréquences de combinaisons déjà observées pour la séquence "Pierre"-"Feuille"-"Ciseaux" jouée 4 fois au début ont augmenté très rapidement. Ce qui fait que, après un changement de tactique brusque, il faut un nombre assez important de coups pour renverser la tendance et pour que le programme puisse enfin modifier son jeu.

Exemple : Supposons qu'on a joué 4 fois la séquence "Pierre"-"Feuille"-"Ciseaux". Puisque c'est un processus du 1er ordre, le programme ne se base que sur le dernier coup pour prédire, et les fréquences obtenues sont donc les suivantes :

Séquence	PP	PF	PC	FP	FC	FF	CP	CF	CC
Fréquence	0	4	0	0	5	0	3	0	0

On voit donc que, si nous changeons subitement de stratégie et que le joueur a joué dernièrement “Feuille” par exemple, le programme va considérer, pendant un assez long moment, que ce qui va suivre est “Ciseau” car cette probabilité a augmenté très vite. Ceci reste valable pour les autres coups.

Néanmoins, les modèles du 2e, 3e et 4e ordre décèlent le changement de stratégie et s’adaptent plus rapidement, à l’inverse des modèles restants, 5e, 6e, 7e et 8e notamment qui, non seulement détectent lentement la séquence “Pierre”-”Feuille”-”Ciseaux” du début, mais aussi mettent évidemment du “temps” pour modifier leur jeu s’il y a changement.

III. Les réseaux de neurones artificiels

1. Principe d’un réseau de neurones

Un réseau de neurones s'inspire du fonctionnement des neurones biologiques et prend corps dans un ordinateur sous forme d'un algorithme. Le réseau de neurones peut se modifier lui-même en fonction des résultats de ses actions. Cela se fait en utilisant une “fonction de coût” - la différence entre les résultats attendus et les résultats donnés par ce dernier - qu’il doit minimiser pour s’approcher au maximum des données d'entraînement, ce qui lui permet d’apprendre [4].

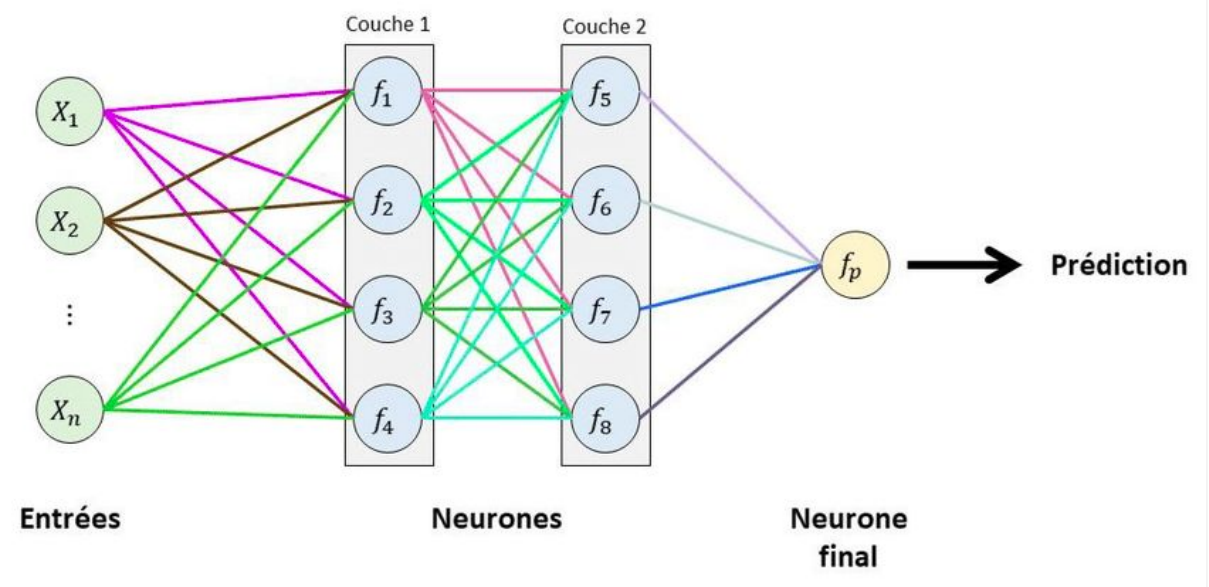


Fig. 5 : Schéma d’un réseau de neurones artificiels

Dans notre cas, on utilise un type de réseau de neurones qui est le plus adapté au traitement de données séquentielles en l'occurrence un réseau de neurones récurrent ou RNN (Recurrent Neural Network en anglais) qui apprend à

prédire à partir de l'historique dans la séquence. Ce type de réseau de neurones, contrairement aux réseaux classiques, utilise non seulement les données en entrées à un instant T mais aussi les données en entrées de T-1 et T-2, etc. Et on peut remarquer assez rapidement que c'est adapté à notre programme vu la pertinence des coups précédents dans la prédiction du prochain coup.

Exemple d'utilisation de RNN:

Soit une séquence S de coups de ce type :

S = (P F F C …) où P, F et C représentent respectivement les coups "Pierre", "Feuille" et "Ciseaux", on met :

- **P** en première entrée pour prédire **F**,
- **F** en deuxième entrée pour prédire **F** comme prochain coup
- **F** en troisième entrée pour prédire **C** comme prochain coup et ainsi de suite…

Cette architecture est proposée par Paul Klinger [5].

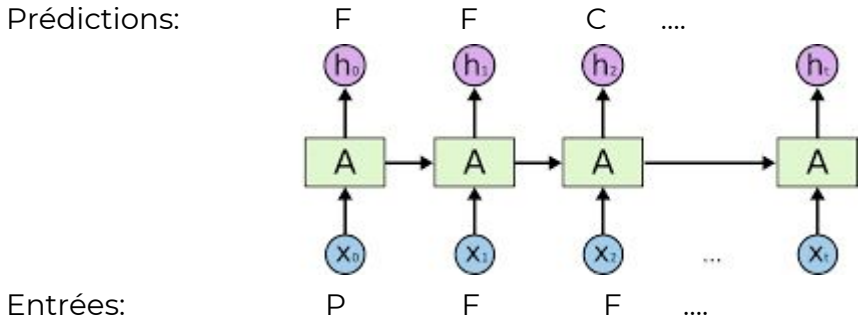


Fig. 6 : *Schéma du processus de prédiction pour une séquence (many to many)*

2. ChiFouMi et le deep learning

Il a été prouvé qu'il existe des architectures de réseau de neurones qui sont optimales pour ce problème [6] : cette architecture est celle du "many to one" qui prend une séquence en entrée et nous donne un coup en sortie. Elle est constituée de deux couches : une couche de GRU (Gated Recurrent Unit) suivie d'une couche complètement connectée (Fully connected layers).

Paramètres	Le paramètre sélectionné
Nombre de neurones dans GRU	96
Nombre de couches GRU	1
Optimiseur	Adam [11]
Fonction de perte	Categorical cross-entropy

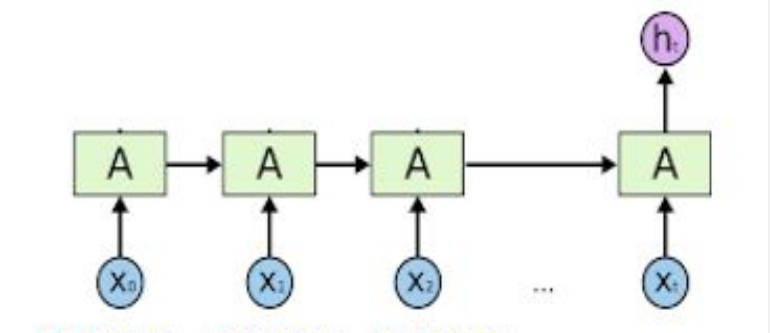


Fig.7 : Architecture d'un réseau de neurone (many to one)

3. Méthodologie

Comme pour chaque tâche d'apprentissage automatique, notre modèle a besoin de données pour s'entraîner et apprendre. Pour ce faire, on a réussi à trouver un fichier csv [7] contenant des centaines de milliers de parties jouées entre deux joueurs humains.
Le fichier se présente ainsi :

```
1,1,1,1
1,1,2,3
1,2,3,2
1,3,1,1
1,3,1,2
1,4,2,3
2,5,1,0
3,6,1,1
```

Les colonnes 1 et 2 représentent respectivement l'identifiant de la partie et le numéro du round.
La 3ème colonne contient les coups du premier joueur et la dernière comporte les choix du second joueur.
Il faut noter que le chiffre '1' correspond au coup "Pierre", '2' à "Feuille" et '3' à "Ciseaux".

On peut remarquer, cependant, la présence de '0' dans la colonne des coups de joueurs : cela veut dire que le joueur n'a pas joué, ce qui nous pousse à ignorer ces lignes dans le processus de l'entraînement.

Nous allons expérimenter différents encodages de données ainsi que différents types d'entrées pour le réseau de neurones. D'abord, on essaiera d'entraîner cette architecture (voir *Fig.5*) sur ces données avec une variation de la taille des séquences à prendre en considération pour prédire le prochain coup, puis on va les tester sur différents types d'entrées (différents encodages) et enfin on suivra l'évolution du taux de prédiction dans une partie de jeu.

4. Résultats et analyse

Dans la littérature, on trouve diverses méthodes de quantifications de performance d'un bot dans le jeux de ChiFouMi. La plupart se focalise sur le taux de coups gagnants du bot en ignorant les parties où le score est nul. On utilise donc l'équation ci-dessous pour calculer les performances d'un programme face à un humain :

$$\frac{\textit{nombre de parties gagnées}}{\textit{nombre de parties gagnées} + \textit{nombre de parties perdus}}$$

Dans notre cas, puisqu'on essaie de prédire le comportement d'un humain, on va plutôt se focaliser sur le taux de prédiction qui sera quantifié par l'équation suivante :

$$\frac{\textit{nombre de bonnes prédictions}}{\textit{nombre de bonnes prédictions} + \textit{nombre de mauvaises prédictions}}$$

Pour les expériences, on a utilisé la librairie *Keras* [8] avec un backend *Tensorflow* [9] et pour l'entraînement c'est *Google Colab* qui nous a permis d'utiliser des GPU (processeur graphique) pour un entraînement plus rapide.

On commence d'abord par tester le format de données d'entrées du réseau de neurones. Pour cela, on utilisera 2 formats différents :

	Encodage simple	***One hot encoding***
Pierre	0	[1 0 0]
Feuille	1	[0 1 0]
Ciseau	2	[0 0 1]

Pour comparer ces deux encodages, on entraîne deux réseaux de neurones avec ces deux types. L'entraînement se fera dans des conditions similaires (batch_size=5 , learning_rate=0.001 , nombre d'epochs=10).

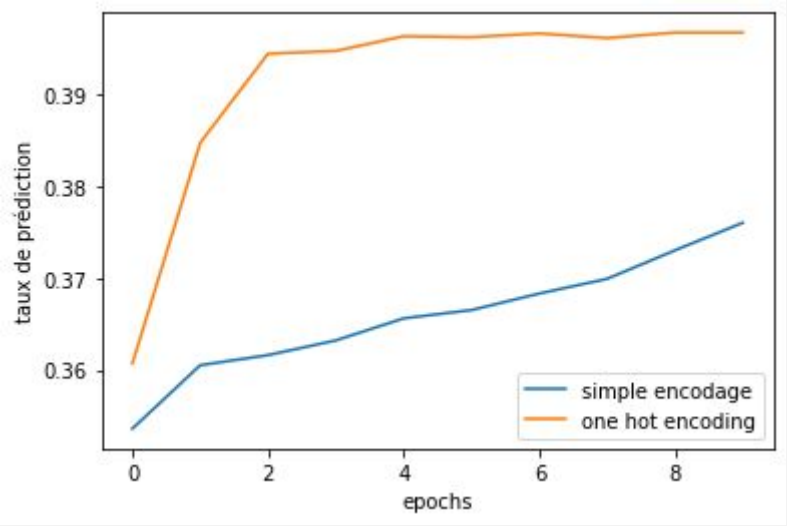


Fig.10 : Comparaison entre 'one hot encoding' et 'encodage simple'

Sur la figure ci-dessus, on peut clairement voir que le *one hot encoding* fait mieux que le deuxième encodage. Cela est dû au principe même des réseaux de

neurones qui sont en réalité une fonction avec des opérations mathématiques (addition et multiplication principalement). Donc, si on encode de façon simple, le '2' (pour "Ciseau") va finir par être plus important que le '0' ("Pierre"). C'est pour cela qu'on remarque que le *one hot encoding* est le mieux adapté pour ce type de problème.

Aussi, a-t-on pu remarquer que le *one hot encoding* marche mieux que l'encodage par des entiers. Dans cette étape, on étudiera l'impact de la variation des entrées du réseau de neurones. En fait, une équipe chinoise a prouvé que le joueur fait des choix en réaction à ceux de son adversaire; et également selon le score de la partie, un joueur a tendance à changer de coup s'il perd ou garder le même coup s'il gagne [10]. Donc ici, on va tester différents types d'entrées :

	Joueur 1	**J1 vs J2**	**J1 vs J2 & score**
(Pierre,Ciseau)	[0 0 1]	[1 0 0 0 0 1]	[1 0 0 0 0 1 1]
(Feuille,Feuille)	[0 1 0]	[0 1 0 0 1 0]	[0 1 0 0 1 0 0]
(Ciseau,Feuille)	[0 1 0]	[0 0 1 0 1 0]	[0 0 1 0 1 0 -1]

On a testé 3 types d'entrées :

- **Joueur 1** : on prend en entrée que les coups du joueur à prédire en *one hot encoding*.
- **J1 vs J2** : on prend les coups des deux joueurs, ici la machine et l'adversaire à prédire. Les trois premiers bits sont pour la machine et le reste est pour le coup du joueur.
- **J1 vs J2 & score** : dans ce cas, on ajoute à J1 vs J2 un autre bit pour coder le score : -1 si la machine perd, 1 si la machine gagne, 0 sinon.

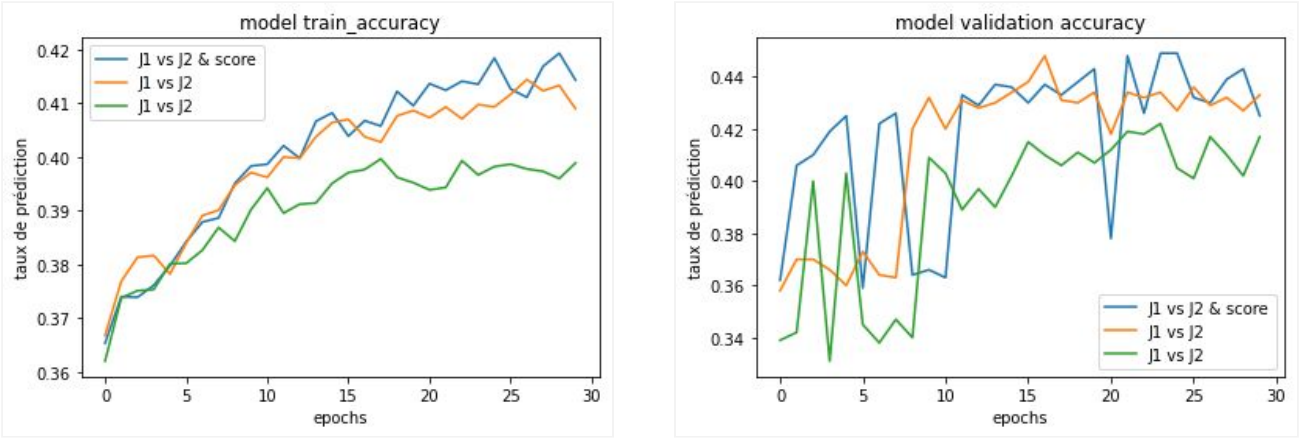


Fig.11 : Evolution du taux de prédiction sur 30 epochs pour les 3 type d'entrées

Sur la figure 11, on peut voir nettement la supériorité des deux types **J1 vs J2** et **J1 vs J2 & score** que ce soit en training ou en validation. On ne peut pas dire grand

chose sur ces deux types; cependant il est redondant de calculer le score vu que les informations nécessaires pour calculer celui-ci sont déjà disponibles dans l'encodage **J1 vs J2** et cela valide ce qui été dit dans l'étude [10]. On a aussi pu remarquer que le réseau de neurones commence à surinterpréter les données après 40 *epochs* d'entraînement, il commence à apprendre les données.
De même, avec l'encodage **J1 vs J2 & score**, le modèle tend également à surinterpréter : en effet, on peut voir que sa courbe de validation *accuracy* est instable contrairement au deux autres modèles.

Dans cette partie on va se focaliser sur l'évolution du taux de prédiction dans une partie de jeu. On verra quel modèle va s'adapter le plus à la stratégie du joueur et pour cela on va comparer plusieurs modèles de prédiction : des modèles déjà entraînés et un modèle vierge (non entraîné au début et qui va être entraîné durant l'évolution de la partie) :

- un modèle déjà entraîné sans le réentrainer sur les données du joueur. Ce modèle est entraîné sur 20 *epochs* (cf. *alrd_train* 20 sur les graphiques).
- un modèle déjà entraîné qui va être réentrainer sur les données avec 20 epochs pour chaque coup (cf. *alrd_train* and *retrain* 20 sur les graphiques).
- un modèle déjà entraîné qui va être réentrainer sur les données avec 10 *epochs* pour chaque coup (cf. *alrd_train* and *retrain* 10).
- un modèle vierge qui n'est pas entraîné préalablement qui va s'entraîner durant la partie (cf. *base_train*).

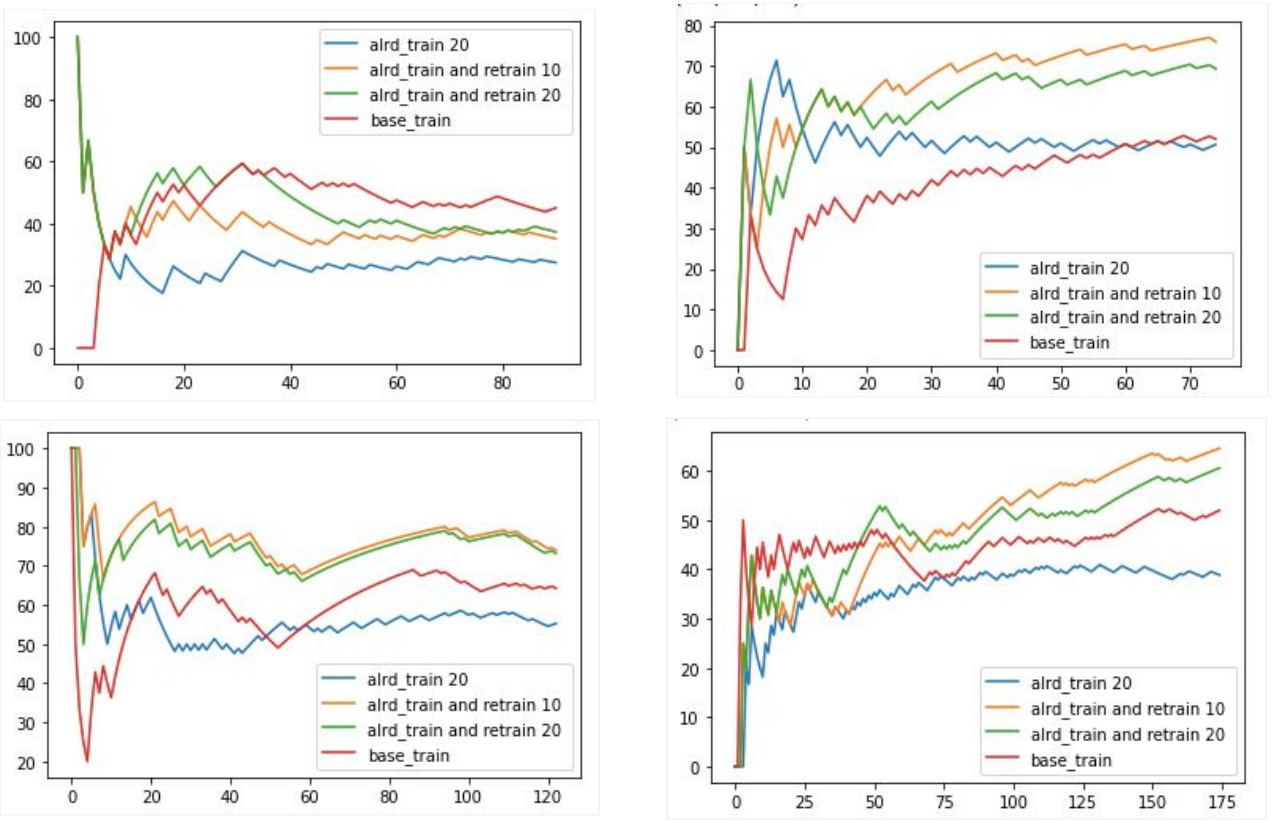


Fig.12 : Evolution des taux de prédiction dans une partie pour les modèles

Sur les graphiques ci-dessus, l'axe des abscisses représente le numéro du coup de jeu et en ordonnée on a le pourcentage d'accuracy.
L'expérience a été faite sur un ensemble de 30 parties où on peut remarquer facilement que les modèles qui s'entraînent durant la partie et qui s'adaptent à la stratégie du joueur ont un taux d'accuracy plus élevé que le modèle déjà entraîné et sans réentrainement. Cela s'explique par le fait que ces données sont presque aléatoires, donc il faudrait que le modèle apprenne à s'adapter à l'aléa du joueur ou à sa stratégie qui peut être absente parfois dans les données de l'entraînement initial.

On peut ajouter la remarque qu'en moyenne les deux modèles qui sont entraînés et réentraînés font mieux que le modèle vierge qui, lui, essaie de s'adapter au changement de stratégie dans une partie sans avoir eu beaucoup d'informations sur d'autre stratégies contrairement aux autres modèles qui ont été déjà exposés à des milliers de parties.

Un autre constat intéressant est le fait de savoir que le modèle qui se réentraîne sur 10 *epochs* fournit des taux de prédiction différents du modèle qui se réentraîne sur 20 *epochs*. Cela est probablement dû au fait que entrainer un modèle un grand nombre de fois sur un même coup le pousse à surinterpréter (*overfitting*) ces données d'entraînement, ce qui ne l'aide pas à se généraliser sur d'autres données.

Conclusion

Après avoir ajusté notre premier algorithme pour le tester avec une chaîne de Markov du 1er au 8e ordre, on peut dire que ce programme a pu atteindre un pourcentage de victoires de près de 80% pour un modèle du second ordre. Étant donné que ce pourcentage a persisté après des centaines de rounds contre des joueurs humains, il est probablement statistiquement significatif. De plus, on a pu constater qu'une chaîne de Markov de ce même ordre était plus performant en termes de prédiction mais aussi en détection rapide de changement.

Dans les expériences menées sur les réseaux de neurones, on a pu comparer plusieurs approches pour ce problème. On a pu atteindre un taux de prédiction de presque 45% en moyenne avec un modèle qui arrive à se généraliser sur de nouvelles données. En effet, formaliser ChiFouMi en un problème d'apprentissage supervisé qui a causé un peu de difficultés. Principalement, le nombre limité des entrées possibles vu la structure simple du jeu peut mener à des mappages contradictoires, un problème qui limite drastiquement les taux de prédiction. Néanmoins, on a pu atteindre des taux de prédiction assez significatifs pour exploiter ces modèles comme bot dans notre application et qui pourrait gagner dans plus de 60% des coups dans une partie.

Pour établir un bilan de notre travail, des améliorations supplémentaires sont sans doute envisageables au niveau des programme mais aussi en ce qui concerne la pertinence des tests ou encore la clarté des descriptions et explications apportées.

Références

1. Knoll, Byron. "Rock Paper Scissors Programming Competition"
2. Morgan, James (2 Mai 2014). "How to win at rock–paper–scissors". BBC News. BBC.
3. Markov chain, Wikipedia.
4. Ian Goodfellow ,Yoshua Bengio and Aaron Courville (https://www.deeplearningbook.org/)
5. https://github.com/PaulKlinger/rps-rnn
6. Mathias Zink , Paulina Friemann , and Marco Ragni :Predictive Systems: The Game Rock-Paper-Scissors as an Example.
7. Site internet https://roshambo.me/ téléchargé via lien
8. Chollet, F., et al.: Keras (2015). https://github.com/fchollet/keras
9. Abadi, M., et al.: TensorFlow: large-scale machine learning on heterogeneous systems (2015). https://www.tensorflow.org/
10. Wang, Z., Xu, B., Zhou, H.J.: Social cycling and conditional responses in the rock paper-scissors game. arXiv preprint arXiv:1404.5199 (2014)
11. Adam: A Method for Stochastic Optimization https://arxiv.org/abs/1412.6980