

Jenselme Pierre

Kujundzic Jordan

Minimisation de regret hypothétique profond et Poker.



Sommaire

1. Introduction

2. Partie théorique

3. Partie pratique

4. Organisation

5. Elargissement

6. Conclusion

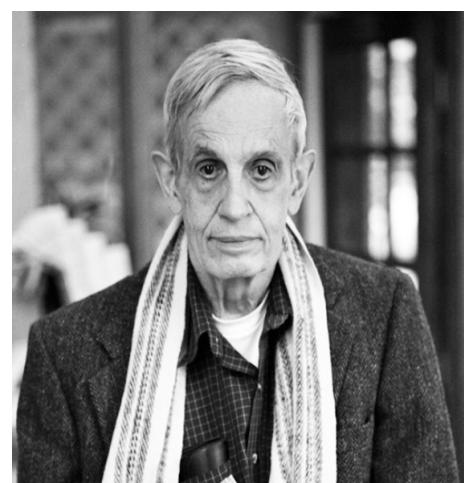
Introduction :

« Un grand maître d'échecs battu par un logiciel », « Victoire finale d'un algorithme contre les meilleurs joueurs de Go », « Une intelligence artificielle s'appuie sur sa mémoire pour maîtriser différents jeux vidéo » ... Presque quotidiennement, nous sommes submergés de nouvelles concernant des machines qui apprennent à jouer. Cependant, les victoires récentes au poker du logiciel Libratus et de l'algorithme DeepStack changent véritablement la donne. Nous nous efforcerons donc tout au long de ce rapport de vous proposer un document le plus complet possible portant sur les différentes intelligences artificielles ayant été développées pour le poker ainsi que sur la minimisation de regret hypothétique appliquée à ce jeu.



Tout d'abord, la théorie des jeux est définie comme étant “l'étude de modèles mathématiques de conflit et de coopération entre preneurs de décisions rationnels et intelligents”.

Devenue un champ de recherche à part entière pendant les années 1940, avec la définition de l'équilibre de Nash dans les années 1950, la théorie des jeux a permis une meilleure compréhension de certains phénomènes jusque-là peu étudiés et en particulier, les conditions qui permettent d'atteindre une coopération entre plusieurs agents.



On peut observer deux catégories de jeux : ceux dont on connaît toutes les informations comme par exemple les échecs ou le jeu de dames, où dans ce cas, l'expérience, la santé mentale et physique ainsi que la stratégie des joueurs sont les seules composantes qui déterminent le gagnant.



De l'autre côté, nous avons ceux dont on ne connaît pas toutes les informations (les jeux à information imparfaite), où le hasard rentre en jeu comme nouvelle composante. Le poker en fait partie, ce qui rend d'autant plus compliqués les implémentations d'algorithmes d'intelligence artificielle dans cette catégorie de jeu et c'est notamment une des raisons pour lesquelles la première IA de poker n'a été créée qu'en 2015 grâce à Cepheus alors que dès 1997, Deep Blue, une IA créée par IBM battait le champion du monde d'échec de l'époque Garry Kasparov.



Il est également intéressant de noter que le poker appartient à la catégorie des jeux à somme nulle. En effet, dans la théorie des jeux, un jeu à somme nulle est une représentation mathématique d'une situation dans laquelle les gains et les pertes de chaque participant sont parfaitement équilibrés par les gains et les pertes des autres joueurs, c'est-à-dire que lors d'une partie où 2000 euros sont autour de la table, le vainqueur de la partie ne gagnera jamais plus de 2000 euros.

A l'inverse des jeux à information parfaite, les plus grandes difficultés au poker sont la dissimulation d'informations (on ne voit pas les cartes de l'adversaire) et la falsification d'informations (on bliffe en faisant croire qu'on a certaines cartes en main).

De plus, dans un Texas Hold'em no-limit à deux joueurs, il y a 10^{161} points de décision. Pour travailler dans ce type d'environnement, il fallait donc une intelligence artificielle spécifique, dont l'algorithme évolutif permettrait d'exploiter le peu d'informations dont elle dispose...

La théorie des jeux apporte des clés, et depuis une dizaine d'années, a pu trouver des applications algorithmiques grâce aux nouveaux algorithmes de minimisation de regret, et notamment la minimisation de regret hypothétique.

Partie théorique :

CFR (Counterfactual Regret Minimization) et explications du regret

Commençons tout d'abord par définir chaque terme :

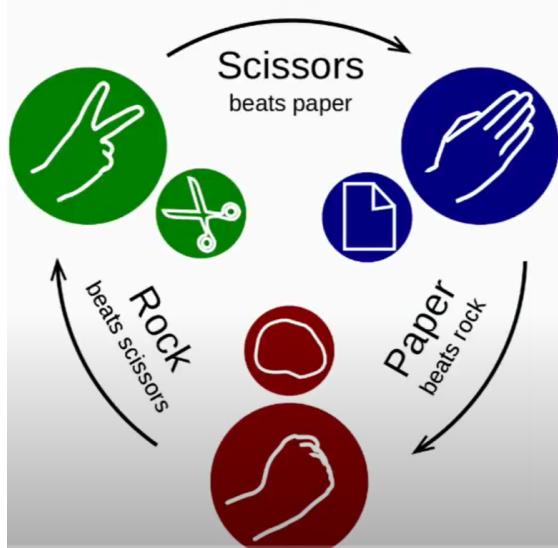
Counterfactual signifie "hypothétique", c'est-à-dire ce qui ne s'est pas passé mais aurait pu se passer sous différentes conditions. Un exemple assez simple et très récurrent de ce phénomène au poker est "Si j'avais fait tapis, j'aurai énormément gagné sur ce coup". En pratique, pour appliquer cela, on va donc explorer dans notre algorithme d'autres actions que celle finalement choisie et regarder le potentiel gain ou perte qu'on aurait eu avec.

Regret signifie ici quelque chose qu'on a fait et qu'on aurait finalement préféré ne pas faire. Dans le monde du poker, un exemple concret et assez récurrent encore une fois est de se coucher alors qu'on aurait pu gagner le pot. En pratique, on va regarder les gains possibles qu'on aurait pu avoir et les soustraire au gain réel qu'on a reçu, ainsi, c'est comme ça qu'on va calculer le regret.

Enfin, Minimization est le but de l'algorithme car on veut en effet minimiser le regret de chaque action que l'on entreprend.

Prenons un exemple simple pour commencer afin de bien comprendre le procédé, au jeu de Pierre-Feuille-Ciseaux, si l'on joue pierre et que notre adversaire joue feuille, on perd. On regrettera donc de ne pas avoir joué Ciseaux. Le regret sera alors le suivant: "si dans toutes les parties précédentes, est-ce qu'en jouant ciseaux, j'aurais plus gagné qu'en suivant ma stratégie actuelle?". Si on n'a fait qu'une partie,

la réponse sera bien évidemment oui. Mais si notre adversaire jouait souvent Pierre, alors la réponse pourrait être non et la stratégie serait peut-être de plus jouer Feuille.



Au départ, la stratégie dite de blueprint est aléatoire, mais après chaque partie toutes les décisions prises sont revues et on peut alors les modifier grâce au regret. Si le regret est positif, il faudra alors changer plus souvent l'action pour celle que l'on estime plus optimale. S'il est négatif, alors on a fait la meilleure chose possible et il faut continuer ainsi. Notre stratégie est ainsi revue pour qu'elle tienne compte des regrets positifs (la probabilité de faire une action dépend donc de son intérêt pour nous faire gagner globalement).

Le CFR est donc le leader des algorithmes pour la gestion des jeux à information imparfaite. De plus, c'est un algorithme d'entraînement avec soi-même, en effet, cela veut dire que l'Intelligence Artificielle apprend en jouant contre elle-même.

Deep CFR

Le Deep CFR est un type de CFR un peu plus poussé, son but est d'approcher le comportement du CFR sans calculer et accumuler les regrets à chaque donnée rencontrée, en généralisant à travers des données similaires et en utilisant l'approximation de fonction via des réseaux neuronaux profonds.

Le deep CFR est une méthode pour trouver des équilibres approximatifs dans de grands jeux à information imparfaite en combinant l'algorithme CFR avec l'approximation de fonctions de réseaux neuronaux profonds. Cette méthode permet d'obtenir de bonnes performances dans les grands jeux de poker par rapport aux techniques d'abstraction. Il s'agit de la première variante non tabulaire de la CFR à être performante dans des jeux de grande envergure. Le Deep CFR et d'autres méthodes neuronales pour les jeux à information imparfaite offrent une direction

prometteuse pour aborder les grands jeux dont le nombre de décisions à prendre est trop grand pour les méthodes tabulaires et où l'abstraction n'est pas directe.

Des travaux récents dans le domaine du deep reinforcement learning ont montré que les réseaux neuronaux peuvent prédire et généraliser efficacement les avantages dans des environnements difficiles avec beaucoup de mains différentes. Les réseaux de neurones, en tant que systèmes capables d'apprendre, mettent en œuvre le principe de l'induction, c'est-à-dire l'apprentissage par l'expérience.

Cela signifie donc que les algorithmes jouant au Poker et contenant le deep CFR jouent comme des êtres humains car ils apprennent de leurs erreurs tout en ayant la mémoire d'un ordinateur, ce qui rend ces algorithmes «inhumains» et comme vous l'aurez compris, inexploitables.

Equilibre de Nash pour la stratégie des jeux

Autrement dit, on ne cherche pas à gagner une partie mais le maximum de parties. L'algorithme ne converge pas toujours, sauf dans le cas du poker où il est possible d'atteindre ce qu'on appelle "l'équilibre de Nash".

L'équilibre de Nash est une stratégie où aucun des joueurs ne va changer la sienne même si elle est moins avantageuse, au fur-et-à-mesure de la partie.

C'est une théorie des jeux pensée par John Forbes Nash, un immense mathématicien ayant reçu le prix nobel d'économie en 1994 qui détermine une solution optimale pour un jeu non coopératif. Pour atteindre cette stratégie optimale au poker on va donc entraîner deux algorithmes reposant sur du CFR l'un contre l'autre jusqu'à qu'ils trouvent la meilleure stratégie possible. A titre d'information, le jeu Pierre Feuille Ciseau à besoin d'environ 10 000 itérations avant de converger vers un équilibre de Nash.

Prenons l'exemple du dilemme du prisonnier pour illustrer cet équilibre. Nous avons deux prisonniers qui veulent sortir de prison, ils ne peuvent pas se concerter et ils ont le choix entre mentir ou se confesser. Voici les différents cas de figures :



- Si les deux mentent et nient les faits alors ils écopent d'une peine réduite de 2 ans de prison chacun.
- Si les deux se confessent, ils écopent de 5 ans de prison chacun.
- Si l'un avoue et l'autre ment alors là, celui qui ment écope de prison et celui qui avoue prend 10 ans.

Ici, on a qu'un équilibre de nash et c'est l'équilibre de double confession. Si le joueur 2 confesse, le joueur 1 confesse également et il n'a pas d'avantage à changer sa stratégie, en effet, il reste dans la confession car si il décide de mentir sa peine passe de 5 à 10 donc il ne changera pas de stratégie, c'est exactement la même chose pour le joueur 2. Nos deux joueurs, lorsqu'ils se retrouveront dans l'équilibre de confessions, ne changeront donc pas de stratégie unilatéralement. Dans l'équilibre de Mensonge si le joueur 1 initialement mentais, il aurait plus d'utilité à se confesser car sa peine passerait de 2 à 0 et pareil pour le joueur 2, cette équilibre n'est donc pas un équilibre de Nash mais un équilibre de Pareto. C'est un bel exemple qui prouve que l'équilibre de Nash n'est donc pas efficace au sens de Pareto car nos deux joueurs préféreraient se retrouver dans un équilibre de mensonge mais cet équilibre n'est pas soutenable comme nous venons de le démontrer, au sens de Nash.

		Joueur 2	
		Confesser	Mentir
Joueur 1	Confesser	-5, -5	0, -10
	Mentir	-10, 0	-2, -2

Maintenant que l'on a expliqué l'équilibre de Nash, il faut expliquer pourquoi on peut le qualifier de stratégie gagnante si elle est parfaitement respectée. En effet, sur un grand nombre de parties, on obtient qu'en moyenne:

- Si l'adversaire suit une stratégie de Nash, on finira à égalité s'il joue une stratégie parfaite.
- Sinon, à la moindre erreur, je gagnerai.

Le concept d'abstraction pour les algorithmes

Pour tous les algorithmes que nous allons voir, nous devons utiliser l'abstraction. C'est un concept ou une idée générale, plutôt que quelque chose de concret ou de tangible. En effet, on peut qualifier l'abstraction comme une version simplifiée de quelque chose de technique, comme une fonction ou un objet dans un programme informatique.

Il y a plusieurs formes d'abstraction mais celle qui nous intéresse est l'abstraction d'actions prises par hasard, c'est-à-dire l'abstraction de cartes, dans le cas du poker. On peut faire des clusters de mains dites similaires grâce aux probabilités de gain au lieu de les traiter individuellement. Cela veut donc dire qu'au lieu d'avoir des millions de mains dans un jeu de poker, on en a plus que 200.

Les différents noms des algorithmes mis au point au cours du temps

- ***Cepheus => 2015***

Cepheus est le premier programme ayant pu résoudre le jeu de Texas Hold'em à deux joueurs (donc pas encore le sans limite ni le multijoueur). Les personnes qui ont travaillé sur *Cepheus* ont réussi à simuler toutes les réponses possibles pour n'importe quelle situation possible lorsque l'on joue au Texas Hold'em à deux joueurs puisque *Cepheus* a joué plus de 60 millions de mains contre lui-même. Pour arriver à ses fins, il a fallu deux mois d'entraînement, 4 000 processeurs tournant simultanément, et six milliards de mains considérées à la seconde.

- ***DeepStack => 2017***

Environ 2 ans après *Cepheus*, un autre robot de poker à succès a été révélé. Cette fois, il pouvait gagner contre les humains dans une version sans limite de Texas Hold'em à deux joueurs. Le robot s'appelait *DeepStack* et il utilisait la résolution continue assistée par les réseaux neuronaux.

L'approche directe consisterait à appliquer le solveur Counterfactual Regret Minimization mais c'est pratiquement impossible. Deepstack gèrait cela en limitant à la fois la profondeur et la largeur du solveur CFR. La largeur est limitée par l'abstraction d'action que nous avons vu un peu plus haut (seulement les actions de pliage, d'appel, de 2 ou 3 mises et all-in sont valides) tandis que la profondeur est limitée par l'utilisation de la fonction de valeur à une certaine profondeur de procédure de calcul de valeur contrefactuelle.

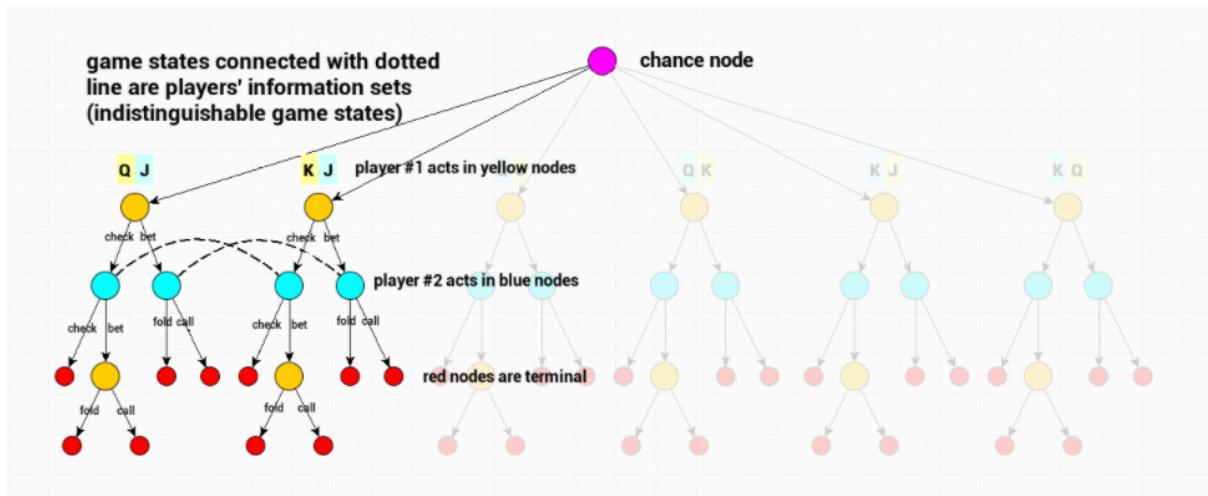
DeepStack a donc joué en 2017 au Texas Hold'em à deux joueurs contre des joueurs de poker professionnels de la Fédération Internationale de Poker. Après

avoir joué 44 852 parties, les résultats de DeepStack étaient dix fois supérieurs à ce qu'un joueur de poker professionnel considère comme une marge importante.

- ***Libratus***

En janvier 2017, une autre étape vers une performance de Poker surhumaine a été franchie. Libratus, le nom de l'intelligence artificielle, a battu une équipe de quatre joueurs professionnels. C'était lors d'un événement qui s'est déroulé dans un casino de Pittsburgh et a duré 20 jours. Ceci a abouti à environ 120 000 mains jouées. Les participants ont été considérés comme plus forts que les adversaires de DeepStack. Malgré tout, l'intelligence artificielle a gagné le tournoi.

Les actions des adversaires (taille des mises) ont été enregistrées pendant l'événement pour prolonger la stratégie de blueprint après chaque jour. Des paris fréquents qui avaient été loin de l'abstraction actuelle de l'action ont également été ajoutés au plan durant les nuits entre les jours de l'événement.



Bien que les approches purement basées sur l'abstraction ont produit de fortes IA pour le poker, l'abstraction seule n'a pas été suffisante pour atteindre des performances surhumaines au Texas Hold'em. En plus de l'abstraction, Libratus s'appuie sur des recherches antérieures sur la résolution de sous-jeux, dans lesquelles une stratégie plus détaillée est calculée pour une partie particulière du jeu atteinte pendant le jeu. Libratus présente de nombreuses avancées dans la résolution de sous-jeux qui se sont avérées essentielles pour atteindre des performances surhumaines.

Libratus joue selon la stratégie du plan abstrait uniquement dans les premières parties de No Limit, où le nombre d'états possibles est relativement faible et où nous pouvons nous permettre une abstraction extrêmement détaillée. En atteignant le troisième tour de pari, ou tout autre moment du jeu où l'arbre de jeu restant est

suffisamment petit, Libratus construit une nouvelle abstraction plus détaillée pour le sous-jeu restant et le résout en temps réel.

Cependant, un sous-jeu ne peut être résolu de manière isolée car sa stratégie optimale peut dépendre d'autres sous-jeux non atteints. Les IA antérieures qui utilisaient la résolution de sous-jeux en temps réel ont résolu ce problème en supposant que l'adversaire joue selon la stratégie du plan. Cependant, l'adversaire peut exploiter cette hypothèse en passant simplement à une stratégie différente. Pour cette raison, la technique peut produire des stratégies bien pires que la stratégie de base et est appelée "résolution de sous-jeux non sûre".

- ***Pluribus***

Un programme d'intelligence artificielle développé par l'université Carnegie Mellon en collaboration avec Facebook AI a battu des professionnels de premier plan au poker Texas Hold'em No-Limit à six joueurs, la forme de poker la plus populaire au monde.

L'IA, appelée Pluribus, a battu le professionnel de poker Darren Elias, qui détient le record du plus grand nombre de titres du World Poker Tour, et Chris Ferguson, vainqueur de six événements des World Series of Poker. Chaque professionnel a joué séparément 5 000 mains de poker contre cinq copies de Pluribus et l'IA les a tous battus.

Les algorithmes de Pluribus ont créé des caractéristiques surprenantes dans sa stratégie. Par exemple, la plupart des joueurs humains évitent le "donk betting", c'est-à-dire le fait de terminer un tour par un call et de commencer le tour suivant par une mise. C'est considéré comme un mouvement qui n'a généralement pas de sens stratégique. Or, Pluribus l'a fait bien plus souvent que les professionnels qu'il a battus.

Alors que Libratus a utilisé environ 15 millions d'heures de calcul pour développer ses stratégies et, pendant les parties en direct, 1 400 processeurs, Pluribus a, quand à lui, calculé sa stratégie en huit jours en utilisant seulement 12 400 heures de calcul et n'a utilisé que 28 processeurs pendant les parties en direct.

Partie pratique :

A présent, il est temps de passer à la partie pratique où nous avons décidé de nous approprier un code codé en Python fonctionnant dans le cadre du poker de Kuhn.

En effet, pour rappel, ce type de poker est une version simplifiée du No Limit Texas Holdem où le jeu ne comprend que trois cartes à jouer , par exemple un roi, une reine et un valet. Une carte est distribuée à chaque joueur, qui peut placer des paris de la même manière qu'un poker standard. Si les deux joueurs parient ou que les deux joueurs passent, le joueur avec la carte la plus élevée gagne, sinon, le joueur parieur gagne. L'équilibre de Nash dans le cas du Poker de Kuhn est atteint avec cette stratégie pour le joueur 1 :

Player 1 Blueprint Strategies

Card & History	Pass	Bet
Jack	80%	20%
Jack + bet	100%	0%
Queen	100%	0%
Queen + bet	40%	60%
King	25%	75%
King + bet	0%	100%

et pour le joueur 2 :

Player 2 Blueprint Strategies

Card & History	Pass	Bet
Jack + pass	67%	33%
Jack + bet	100%	0%
Queen + pass	100%	0%
Queen + bet	65%	33%
King + pass	0%	100%
King + bet	0%	100%

Nous expliquerons cela dans la partie résultats. Pour l'instant, assez brièvement, l'algorithme que nous allons vous présenter fonctionne de la manière suivante pour le CFR :

- 1 - Sélection de l'action qu'on veut faire
- 2 - On calcule le reward reçu
- 3 - On calcule le Counterfactual reward (par exemple, d'autres rewards possibles d'autres actions qu'on aurait pu entreprendre)
- 4 - On soustrait le Counterfactual reward au reward actuel et on obtient le Regret
- 5 - On stock dans un tableau le regret

On se retrouve donc avec des tableaux comme ça (c'est un exemple) :

Regrets

Choice 1	Choice 2	Choice 3
25	15	10

Strategy

Choice 1	Choice 2	Choice 3
50%	30%	20%

Afin de vous donner une vue d'ensemble, voici le code en entier :

```

class Kunh:

    def __init__(self):
        self.nodeMap = {}
        self.expected_game_value = 0
        self.n_cards = 3
        self.nash_equilibrium = dict()
        self.current_player = 0
        self.deck = np.array([0, 1, 2])
        self.n_actions = 2

    def train(self, n_iterations=50000):
        expected_game_value = 0
        for _ in range(n_iterations):
            shuffle(self.deck)
            expected_game_value += self.cfr('', 1, 1)
            for _, v in self.nodeMap.items():
                v.update_strategy()

        expected_game_value /= n_iterations
        display_results(expected_game_value, self.nodeMap)

    def cfr(self, history, pr_1, pr_2):
        n = len(history)
        is_player_1 = n % 2 == 0
        player_card = self.deck[0] if is_player_1 else self.deck[1]

        if self.is_terminal(history):
            card_player = self.deck[0] if is_player_1 else self.deck[1]
            card_opponent = self.deck[1] if is_player_1 else self.deck[0]
            reward = self.get_reward(history, card_player, card_opponent)
            return reward

        node = self.get_node(player_card, history)
        strategy = node.strategy

        # Counterfactual utility per action.
        action_utils = np.zeros(self.n_actions)

        for act in range(self.n_actions):
            next_history = history + node.action_dict[act]
            if is_player_1:
                action_utils[act] = -1 * self.cfr(next_history, pr_1 * strategy[act], pr_2)
            else:
                action_utils[act] = -1 * self.cfr(next_history, pr_1, pr_2 * strategy[act])

        # Utility of information set.
        util = sum(action_utils * strategy)
        regrets = action_utils - util
        if is_player_1:
            node.reach_pr += pr_1
            node.regret_sum += pr_2 * regrets
        else:
            node.reach_pr += pr_2
            node.regret_sum += pr_1 * regrets

        return util

    @staticmethod
    def is_terminal(history):
        if history[-2:] == 'pp' or history[-2:] == "bb" or history[-2:] == 'bp':
            return True

    @staticmethod
    def get_reward(history, player_card, opponent_card):
        terminal_pass = history[-1] == 'p'
        double_bet = history[-2:] == "bb"
        if terminal_pass:
            if history[-2:] == 'pp':
                return 1 if player_card > opponent_card else -1
            else:
                return 1
        elif double_bet:
            return 2 if player_card > opponent_card else -2

    def get_node(self, card, history):
        key = str(card) + " " + history
        if key not in self.nodeMap:
            action_dict = {0: 'p', 1: 'b'}
            info_set = Node(key, action_dict)
            self.nodeMap[key] = info_set
        return info_set
        return self.nodeMap[key]

```

```

class Node:
    def __init__(self, key, action_dict, n_actions=2):
        self.key = key
        self.n_actions = n_actions
        self.regret_sum = np.zeros(self.n_actions)
        self.strategy_sum = np.zeros(self.n_actions)
        self.action_dict = action_dict
        self.strategy = np.repeat(1/self.n_actions, self.n_actions)
        self.reach_pr = 0
        self.reach_pr_sum = 0

    def update_strategy(self):
        self.strategy_sum += self.reach_pr * self.strategy
        self.reach_pr_sum += self.reach_pr
        self.strategy = self.get_strategy()
        self.reach_pr = 0

    def get_strategy(self):
        regrets = self.regret_sum
        regrets[regrets < 0] = 0
        normalizing_sum = sum(regrets)
        if normalizing_sum > 0:
            return regrets / normalizing_sum
        else:
            return np.repeat(1/self.n_actions, self.n_actions)

    def get_average_strategy(self):
        strategy = self.strategy_sum / self.reach_pr_sum
        # Re-normalize
        total = sum(strategy)
        strategy /= total
        return strategy

    def __str__(self):
        strategies = ['{:03.2f}'.format(x)
                      for x in self.get_average_strategy()]
        return '{} {}'.format(self.key.ljust(6), strategies)

def display_results(ev, i_map):
    print('player 1 expected value: {}'.format(ev))
    print('player 2 expected value: {}{}'.format(-1 * ev))

    print()
    print('player 1 strategies:')
    sorted_items = sorted(i_map.items(), key=lambda x: len(x[0]) % 2 == 0, reverse=True)
    for _, v in filter(lambda x: len(x[0]) % 2 == 0, sorted_items):
        print(v)
    print()
    print('player 2 strategies:')
    for _, v in filter(lambda x: len(x[0]) % 2 == 1, sorted_items):
        print(v)

if __name__ == "__main__":
    time1 = time.time()
    trainer = Kunh()
    trainer.train(n_iterations=25000)
    print(abs(time1 - time.time()))
    print(sys.getsizeof(trainer))

```

A présent, analysons chaque partie :

1 - le Constructeur de la classe Kuhn :

```
def __init__(self):
    self.nodeMap = {}
    self.expected_game_value = 0
    self.n_cards = 3
    self.nash_equilibrium = dict()
    self.current_player = 0
    self.deck = np.array([0, 1, 2])
    self.n_actions = 2
```

- Dans le constructeur, on crée un dictionnaire appelé nodeMap où on stockera les différentes étapes de jeu représenté par des noeuds (Node en anglais).
- On crée une variable appelée expected_game_value qu'on initialise à 0.
- On initialise le nombre de cartes du jeu à 3 via la variable n_cards.
- On crée un dictionnaire appelé nash equilibrium via nash_equilibrium = dict().
- On définit une variable indiquant le joueur actuel, à l'instant t, appelé current_player et on l'initialise à 0.
- On crée un jeu de cartes via une matrice que l'on appelle deck et dans laquelle on insère 0, 1 et 2 qui correspondent aux 3 cartes du poker de Kuhn, l'équivalent du Valet, Dame, Roi.
- Enfin, on crée une variable n_actions que l'on initialise à 2 : Pass ou Bet.

2- Le constructeur de la classe Node :

```
def __init__(self, key, action_dict, n_actions=2):
    self.key = key
    self.n_actions = n_actions
    self.regret_sum = np.zeros(self.n_actions)
    self.strategy_sum = np.zeros(self.n_actions)
    self.action_dict = action_dict
    self.strategy = np.repeat(1/self.n_actions, self.n_actions)
    self.reach_pr = 0
    self.reach_pr_sum = 0
```

- Ici, on crée une variable Key que l'on initialise à la key qu'on rentre dans le constructeur.
- Pareil pour le nombre d'actions n_actions qui est égal à 2.

- Puis on crée une matrice appelée regret_sum. Quand on va calculer nos regrets, on va les stocker à l'intérieur avec une ligne pour chaque action que l'on a.
- On crée une variable action_dict que l'on initialise avec la valeur donnée au constructeur.
- On crée une matrice strategy qu'on va remplir avec $1/n_actions$ autant de fois qu'il y a d'actions, c'est à dire 2.
- On créer un variable reach_pr qui correspond à la probabilité d'attendre un nœud en particulier.
- Enfin, On créer une variable reach_pr_sum où l'on stockera la somme des probabilités d'atteindre ce nœud.

3- La fonction CFR :

Voici la fonction CFR :

```
def cfr(self, history, pr_1, pr_2):
    n = len(history)
    is_player_1 = n % 2 == 0
    player_card = self.deck[0] if is_player_1 else self.deck[1]

    if self.is_terminal(history):
        card_player = self.deck[0] if is_player_1 else self.deck[1]
        card_opponent = self.deck[1] if is_player_1 else self.deck[0]
        reward = self.get_reward(history, card_player, card_opponent)
        return reward

    node = self.get_node(player_card, history)
    strategy = node.strategy

    # Counterfactual utility per action.
    action_utils = np.zeros(self.n_actions)

    for act in range(self.n_actions):
        next_history = history + node.action_dict[act]
        if is_player_1:
            action_utils[act] = -1 * self.cfr(next_history, pr_1 * strategy[act], pr_2)
        else:
            action_utils[act] = -1 * self.cfr(next_history, pr_1, pr_2 * strategy[act])

    # Utility of information set.
    util = sum(action_utils * strategy)
    regrets = action_utils - util
    if is_player_1:
        node.reach_pr += pr_1
        node.regret_sum += pr_2 * regrets
    else:
```

Nous avons donc une fonction qui s'occupe de faire tourner le CFR, elle est récursive comme vous pouvez le voir et c'est pour cela que l'on doit avoir notre `is_terminal(history)`, la dernière étape où on s'est arrêté qui correspond tout simplement aux différentes étapes qui correspondraient à une fin de jeu : si les deux joueurs passent, parient ou si un joueur parie et l'autre passe.

```
@staticmethod
def is_terminal(history):
    if history[-2:] == 'pp' or history[-2:] == "bb" or history[-2:] == 'bp':
        return True
```

Nous avons aussi la fonction `get_reward` où simplement, à chaque fois que les joueurs montrent leurs mains pour savoir qui gagne la main, on détermine quelle carte à la valeur la plus élevée et si un joueur passe alors que l'autre a misé, on attribue à l'autre le reward de 1.

```
@staticmethod
def get_reward(history, player_card, opponent_card):
    terminal_pass = history[-1] == 'p'
    double_bet = history[-2:] == "bb"
    if terminal_pass:
        if history[-2:] == 'pp':
            return 1 if player_card > opponent_card else -1
        else:
            return 1
    elif double_bet:
        return 2 if player_card > opponent_card else -2
```

On utilise ces fonctions dans la fonction CFR de la manière suivante :

```
if self.is_terminal(history):
    card_player = self.deck[0] if is_player_1 else self.deck[1]
    card_opponent = self.deck[1] if is_player_1 else self.deck[0]
    reward = self.get_reward(history, card_player, card_opponent)
    return reward
```

Maintenant, on doit déterminer à quel nœud on est via la fonction `get_node` et ensuite récupérer la stratégie du nœud.

```
node = self.get_node(player_card, history)
strategy = node.strategy
```

Ensuite, on va appeler de manière récursive la fonction CFR qui va nous retourner le reward qu'on va stocker dans `actions_utils`.

```

action_utils = np.zeros(self.n_actions)

for act in range(self.n_actions):
    next_history = history + node.action_dict[act]
    if is_player_1:
        action_utils[act] = -1 * self.cfr(next_history, pr_1 * strategy[act], pr_2)
    else:
        action_utils[act] = -1 * self.cfr(next_history, pr_1, pr_2 * strategy[act])

```

Puis on va additionner tous les reward multipliés par les stratégies et soustraire chaque actions à ce reward total dans la variable Regrets. (Le coeur de notre algorithme, expliqué page 13)

```

util = sum(action_utils * strategy)
regrets = action_utils - util

```

Enfin, pour finir, on va actualiser reach_pr qui correspond pour rappel à la probabilité d'atteindre le nœud en question. On va également actualiser la somme des regrets que l'on va actualiser avec la variable regrets multiplié par la probabilité d'atteindre ce nœud.

```

if is_player_1:
    node.reach_pr += pr_1
    node.regret_sum += pr_2 * regrets
else:
    node.reach_pr += pr_2
    node.regret_sum += pr_1 * regrets

return util

```

3- la fonction Train

```

def train(self, n_iterations=50000):
    expected_game_value = 0
    for _ in range(n_iterations):
        shuffle(self.deck)
        expected_game_value += self.cfr('', 1, 1)
        for _, v in self.nodeMap.items():
            v.update_strategy()

    expected_game_value /= n_iterations
    display_results(expected_game_value, self.nodeMap)

```

Cette classe va servir à entraîner l'algorithme contre lui-même grâce à une boucle de 50 000 itérations.

- La fonction shuffle va mélanger notre deck de manière aléatoire à chaque tour de boucle.
- Puis expected_game_value va s'incrémenter à chaque tour avec ce que renvoie notre fonction CFR, c'est-à-dire la variable util qui correspond à la somme des rewards.
- Enfin, pour chaque nœud, on va utiliser la méthode update_strategy contenue dans la classe Node ce qui va nous aider à trouver la stratégie finale que l'on va utiliser.
- Puis on divise notre expected_game_value par le nombre d'itérations utilisé.

La fonction get_strategy :

On va prendre les regrets qu'on va sommer pour les normaliser puis on les divisent par leur nombre ce qui va tout simplement nous donner un pourcentage.

```
def get_strategy(self):
    regrets = self.regret_sum
    regrets[regrets < 0] = 0
    normalizing_sum = sum(regrets)
    if normalizing_sum > 0:
        return regrets / normalizing_sum
    else:
        return np.repeat(1/self.n_actions, self.n_actions)
```

La fonction get_average_strategy :

Elle retourne tout simplement la moyenne des stratégies utilisées.

Affichage :

```
def display_results(ev, i_map):
    print('player 1 expected value: {}'.format(ev))
    print('player 2 expected value: {}'.format(-1 * ev))

    print()
    print('player 1 strategies:')
    sorted_items = sorted(i_map.items(), key=lambda x: x[0])
    for _, v in filter(lambda x: len(x[1]) % 2 == 0, sorted_items):
        print(v)
    print()
    print('player 2 strategies:')
    for _, v in filter(lambda x: len(x[1]) % 2 == 1, sorted_items):
        print(v)

if __name__ == "__main__":
    time1 = time.time()
    trainer = Kuhn()
    trainer.train(n_iterations=25000)
    print(abs(time1 - time.time()))
    print(sys.getsizeof(trainer))
```

Résultat :

Nous allons maintenant pouvoir analyser les résultats de cet algorithme, mais avant tout, voici une vue d'ensemble de l'affichage. On observe directement que notre code fonctionne plutôt bien puisque on obtient les mêmes résultats que les tableaux présentés en page 10 qui portaient sur les stratégies pour chaque joueur respectant un équilibre de Nash exact.

```
In [7]: runfile('D:/kuhn_cfr.py', wdir='D:')

player 1 expected value: -0.050832241960913245
player 2 expected value: 0.050832241960913245

player 1 strategies:
0      ['0.75', '0.25']
0 pb   ['1.00', '0.00']
1      ['0.99', '0.01']
1 pb   ['0.40', '0.60']
2      ['0.23', '0.77']
2 pb   ['0.00', '1.00']

player 2 strategies:
0 b    ['1.00', '0.00']
0 p    ['0.68', '0.32']
1 b    ['0.64', '0.36']
1 p    ['1.00', '0.00']
2 b    ['0.00', '1.00']
2 p    ['0.00', '1.00']

4.565027475357056
48
```

Ici, l'équilibre de Nash est atteint pour une valeur d'environ 1/17 pour le joueur 1 et forcément l'inverse pour le joueur 2 ce qui correspond à -1/17 et en effet, nous pouvons voir grâce à l'expected value que c'est bien le cas.

Deuxièmement, ici, l'algorithme va nous afficher les stratégies gagnantes pour chacun des joueurs.

Commençons par le joueur 1 :

Ligne 1 :

```
0      ['0.75', '0.25']
```

Si il a un 0 alors il a la main la plus faible, il n'a donc pas grand intérêt à miser c'est pour cela qu'il va checker à 75% du temps et miser à 25% du temps pour tenter de faire passer un bluff.

Ligne 2 :

```
0 pb  ['1.00', '0.00']
```

le “pb” signifie “pass bet”, ainsi, si comme prévu le joueur 1 check ou pass, ici c'est la même chose et que par contre, le joueur 2 mise alors il se couchera à 100% du temps ce qui est assez logique vue sa main.

Ligne 3 :

```
1  ['0.99', '0.01']
```

Si le joueur 1 a un 1, il checkera à 99% du temps et il misera uniquement à 0,01% du temps.

Ligne 4 :

```
1 pb  ['0.40', '0.60']
```

Ici, si le joueur 2 mise après que le joueur 1 est checker alors le joueur 1 se couchera à 40% du temps et misera à 60% du temps car il a une main moyenne donc une possibilité d'avoir une meilleure carte que son adversaire.

Ligne 5 :

```
2  ['0.23', '0.77']
```

Ici, si le joueur 1 a 2 comme carte alors il a forcément la meilleure carte du jeu, il checkera donc à 23% du temps et misera à 77% du temps.

Ligne 6 :

```
2 pb  ['0.00', '1.00']
```

Ici, si le joueur 2 mise après que le joueur 1 est checker alors dans ce cas là, le joueur 1 va suivre la mise à 100% du temps et ne se coucher dans aucun cas car il a la meilleure main.

Passons maintenant du côté du joueur 2 :

Ligne 1 :

0 b ['1.00', '0.00']

Si le joueur 1 bet et que le joueur 2 a un 0 alors il a la pire main et se couchera donc dans 100% des cas.

Ligne 2 :

0 p ['0.68', '0.32']

Par contre, si le joueur 1 check alors il checkera aussi dans 68% des cas et misera uniquement dans 26% des cas pour tenter un bluff.

Ligne 3 :

1 b ['0.64', '0.36']

Ici, si le joueur 2 a un 1 et que le joueur 1 mise alors il se couchera dans 68% des cas et suivra la mise dans 36% des cas car il a une main moyenne.

Ligne 4 :

1 p ['1.00', '0.00']

Par contre, si le joueur 1 check alors il checkera aussi et ne misera dans aucun cas car ca serait trop risqué.

Ligne 5 :

2 b ['0.00', '1.00']

Ici, si le joueur 2 a un 2 et que le joueur 1 mise alors il suivra la mise dans 100% des cas car il a la meilleure main du jeu.

Ligne 6 :

```
2 p  ['0.00', '1.00']
```

Par contre, si le joueur 1 check alors dans ces cas là, il mettra la pression en misant car ici aussi, il a forcément la meilleure main.

Organisation:

On a reçu nos premières instructions à propos du projet le 2 mars. La date de rendu était fixée au 7 avril, ce qui nous donnait un peu plus d'un mois. Nous avions nos partiels les 12, 29 et 30 mars. Cela nous a donc un peu ralenti dans nos recherches, avec des pauses puis des reprises. Nous avons d'abord effectué des recherches sur la partie théorique avec une consultation des articles donnés sur le Dropbox de Campus et d'autres articles ou vidéos trouvés. Après nos premiers partiels, nous avons continué nos recherches en effectuant des recherches de code fonctionnel sur le CFR puis nous avons fait un travail de compréhension de celui-ci ce qui n'était pas chose facile au début mais à force de persévérance, nous vous avons expliqué le code et les résultats du mieux possible.

Enfin, lorsque nos recherches étaient finies, nous avons effectué une mise en forme du document à rendre et un travail de relecture du document pour être sûr de bien avoir expliqué les éléments évoqués puis nous avons apporté certaines précisions si cela était nécessaire.

Vous pouvez d'ailleurs voir ceci sur le diagramme de Gantt ci-dessous :



Pour ce qui est de la répartition des tâches, cela s'est effectué plutôt naturellement, j'avais commencé les recherches sur la partie théorique parce que Pierre avait quelques soucis sur son projet PPE. Il m'a ensuite rapidement rejoint pour m'aider dans les recherches théoriques puis est passé aux recherches sur un code CFR pouvant tourner sur nos ordinateurs et a fouillé un peu partout dans les dépôts git afin d'en trouver un assez compréhensible. Enfin, nous avons pris ces derniers jours avant le rendu pour bien vérifier nos sources et compléter nos recherches.

Conclusion :

Pour conclure, nous aimions apporter un bilan personnel sur notre ressenti par rapport à ce projet d'intelligence artificielle.

Pierre : En tant que grand fan de poker et joueur amateur, j'ai directement voulu prendre ce sujet car il réunissait 2 sujets me tenant à cœur : l'intelligence artificielle qui est le véritable cœur de métier d'un ingénieur aujourd'hui selon moi, en tout cas elle est amenée à le devenir de plus en plus et le poker, un jeu auquel je joue régulièrement. Je m'intéressais déjà un peu avant de découvrir ce sujet aux différentes IA existantes au poker et connaissait surtout Deepstack qui est sûrement celle la plus connue du grand public. Je compte à présent essayer d'implémenter des algorithme d'IA sur du NLHE à 6 joueurs même si c'est beaucoup plus complexe car en travaillant sur ce sujet et en fouillant dans les différents dépôts github, je suis tombé sur des forums de passionnés de poker qui en discutaient et leurs discussions étaient très intéressantes, je me suis d'ailleurs créé un compte dessus. De plus, la technologie derrière comme le CFR et les réseaux neuronaux m'ont évidemment impressionné en tant qu'élève ingénieur et c'est une autre raison pour laquelle je compte bien continuer à faire des recherches sur ce sujet.

Jordan: Je joue actuellement plus aux échecs qu'au poker mais le poker était un jeu que je pratiquais très souvent il y a quelques années donc ce sujet a tout de suite attiré mon attention lorsque vous avez fait la présentation des différents sujets. Avec Pierre, nous nous sommes directement orientés vers le Poker et j'étais content d'avoir un binôme partageant le même avis que moi sur la question du sujet à prendre. Cela s'est ressenti dans notre travail et je suis très satisfait de ce que l'on a fourni. Malgré quelques calculs difficiles à comprendre, c'est sûrement le sujet qui m'a passionné le plus et j'ai vraiment aimé faire ces recherches. Ça m'a passionné de voir comment les ordinateurs créés grâce à l'intelligence artificielle pouvaient maintenant battre les meilleurs joueurs de Poker au monde et même d'être qualifiés d'«inhumains» par ces mêmes joueurs. Et cela m'a même donné envie de reprendre le Poker pour de nouveau y jouer.

Elargissement :

Les concepteurs de Pluribus nous ont également fait part de leur confiance sur la possibilité que l'intelligence artificielle puisse résoudre de plus en plus de problèmes du monde réel où, comme au poker, des informations sont cachées ou indisponibles.

Par exemple, un des créateurs de Pluribus, Noam Brown, y voit des applications potentielles dans les domaines de la cybersécurité, de la détection des fraudes, ou même dans le développement des voitures autonomes.

Ceci constitue définitivement un élargissement très important de l'application des techniques d'Intelligence Artificielle exploitées pour la résolution du Texas Hold'em no limit à six joueurs.

Ressources utilisées en plus de campus:

<https://www.youtube.com/watch?v=htRtfyab-Ns>

<https://www.youtube.com/watch?v=JuvN4mi861k>

<https://penseeartificielle.fr/ia-poker-libratus-bat-professionnels/>

<https://www.google.fr/amp/s/www.developpez.com/actu/112318/Jeu-de-poker-l-IA-meilleure-que-l-homme-Des-chercheurs-declarent-avoir-battu-des-professionnels-avec-l-IA-DeepStack-et-l-IA-Libratus-est-en-tete/%3famp>

<https://arxiv.org/pdf/2007.13544.pdf>

<https://github.com/IanSullivan/PokerCFR>