

# ChessMate : jeu d'échecs et intelligence artificielle

Thomas Dahmen, Oscar Bouvier, Raphaël Macquet, Jean Forissier

24 mai 2018

PRO3600-18-BRU-31  
Télécom SudParis

## Préambule

Afin de mener à bien ce projet, nous avons réparti le travail de la façon suivante :

- Thomas : réalisation de l'interface graphique avec PyQt, du réseau de neurones artificiel, traitement des données, génération de la documentation en HTML, gestion d'équipe et rédaction du rapport.
- Oscar : réalisation du moteur de jeu, de la méthode min-max/alpha-bêta et de Monte-Carlo, rédaction du rapport.
- Raphaël : réalisation de la méthode des k plus proches voisins, test unitaire, rédaction de la partie KNN du rapport.
- Jean : intégration des différents modules, gestion du planning et réalisation des comptes rendus de réunion.

Concernant le dépôt Git, nous avons choisi d'utiliser la plateforme GitHub car nous l'avions déjà utilisée et elle est plus pratique pour partager publiquement notre projet. Le dépôt se trouve *ici*

## Table des matières

<b>1 Objectif et cahier des charges</b>	<b>3</b>
1.1 Contexte et historique . . . . .	3
1.2 Description de la demande . . . . .	4
1.3 Contraintes . . . . .	4
<b>2 Conception détaillée</b>	<b>6</b>
2.1 Regroupements modulaires . . . . .	6
2.2 Moteur de jeu . . . . .	6
2.3 Interface graphique . . . . .	7
2.4 Algorithme minimax par élagage alpha-bêta . . . . .	10
2.4.1 L'algorithme minimax . . . . .	10
2.4.2 L'élagage alpha-bêta . . . . .	12
2.5 Réseau de neurones artificiels . . . . .	12
2.5.1 Développement et historique des réseaux de neurones artificiels . . . . .	12
2.5.2 Intérêt des réseaux de neurones artificiels dans notre projet	15
2.5.3 Construction du jeu de données . . . . .	16
2.5.4 Mise en place du réseau de neurones . . . . .	18
2.6 Évaluation par la méthode des k plus proches voisins . . . . .	20
2.7 Méthode de recherche arborescente de Monte-Carlo . . . . .	22
<b>3 Tests et Analyses</b>	<b>25</b>
3.1 Tests Unitaires . . . . .	25
3.2 Test et Analyse de nos programmes . . . . .	26
3.2.1 Moteur de jeu et interface graphique . . . . .	26
3.2.2 L'algorithme minimax et élagage alpha-bêta . . . . .	26
3.2.3 Le réseau de neurones artificiel . . . . .	28
3.2.4 Les K plus proches voisins . . . . .	30
3.2.5 La méthode de recherche arborescente de Monte-Carlo . .	30
3.3 Perspectives et conclusion . . . . .	31
3.3.1 Améliorations possibles pour la méthode de Monte Carlo	31
3.3.2 Améliorations possibles pour le réseau de neurones . . .	32
3.3.3 Conclusion générale . . . . .	32
<b>4 Manuel utilisateur</b>	<b>33</b>
4.1 Modules utilisés . . . . .	33
4.2 Commandes d'utilisation . . . . .	33
4.3 Documentation . . . . .	34

# 1 Objectif et cahier des charges

## 1.1 Contexte et historique

Historiquement, le jeu d'échecs a été l'un des premiers défis en intelligence artificielle. Il existe même des championnats du monde d'échecs entre ordinateurs, le premier a eu lieu en 1974 et fut remporté par un programme soviétique. La première machine à avoir battu un champion du monde dans des conditions classiques (c'est à dire, il ne s'agissait pas d'une partie rapide) est Deep Blue, développé en 1997 par IBM. À l'époque, Deep Blue fonctionnait sur une machine très puissante :



DeepBlue est un super ordinateur possédant 32 cores, capable d'évaluer 200 millions de positions par seconde, avec une puissance de 11,4 GFlop/s, loin des ordinateurs personnels de l'époque. Une telle puissance permet alors d'obtenir une intelligence artificielle très performante en testant un nombre important de coups sans pour autant tous les tester (technique dite « min- max »). Ceci permet de battre n'importe quel joueur sous des conditions normales, mais pour autant le fait qu'un ordinateur puisse résoudre le jeu d'échecs (avoir une stratégie permettant de gagner dans absolument tous les cas) reste un problème ouvert. Après la victoire de la machine sur le champion du monde Kasparov, le défi informatique du jeu d'échecs s'est amoindri et s'est reporté sur le jeu de Go, réputé bien plus complexe pour un ordinateur.

Justement, en décembre 2017, AlphaZero (généralisation d'AlphaGo, intelligence artificielle développée par Google ayant battu le champion du monde de Go) a réalisé l'exploit d'apprendre à jouer en 4h, à la suite desquelles il a battu sur 100 parties la meilleure intelligence artificielle en date (Stockfish). La particularité d'AlphaZero est que l'apport humain a été très faible : on lui a seulement appris les règles basiques (quels coups sont jouables ou non). En partant d'une stratégie au hasard, avec des algorithmes d'apprentissage profond AlphaZero est devenu champion en jouant contre lui-même.

## 1.2 Description de la demande

Notre projet consiste à développer un jeu d'échecs muni d'une interface graphique et d'une intelligence artificielle. Nous nous attellerons principalement au développement de l'intelligence artificielle via un réseau de neurones. En effet, il serait possible, grâce aux immenses bases de données de parties disponibles sur le jeu d'échecs et moyennant une étude empirique (essai de différents nombre de couches par exemple), d'obtenir une intelligence artificielle via cette technique pour ce jeu. Il n'est cependant absolument pas dit qu'elle soit réellement « performante », d'où la nécessité de l'étude comparative. La méthode de prédiction KNN pourrait, elle aussi en se basant sur les mêmes bases de données, donner des résultats satisfaisant.

Nous développerons ce projet en utilisant le langage Python, muni des librairies usuelles comme Numpy, ainsi que d'une librairie dédiée à l'apprentissage machine comme Tensorflow et Keras, et de PyQt, librairie permettant d'avoir une interface graphique pour notre programme. Nous développons en premier lieu un jeu d'échec de A à Z (sans utiliser de librairies comme PyChess) avec une interface graphique.

Ensuite, nous tenterons de développer une IA à l'aide d'un réseau de neurones artificiel. Pour l'apprentissage, nous utiliserons une large base de données de parties des meilleurs joueurs humains comme Chessbase (8 millions de parties).

## 1.3 Contraintes

Même si nous l'avons évoqué, il ne s'agit pas de reproduire les récents exploits d'AlphaZero. Ceci est très probablement impossible avec nos outils et connaissances (besoin de GPU très puissants et d'algorithmes extrêmement optimisés). Nous allons utiliser les GPU disponibles sur nos ordinateurs personnels, ou alors ceux mis à disposition par l'école. Pour l'apprentissage et la construction du réseau de neurones, nous utiliserons Tensorflow, une librairie Python open-source. Notons que le réseau de neurones que nous développerons a un résultat inconnu. Il est en effet illusoire de croire que cette technique sera adéquate, car elle ne fonctionne pour l'instant que dans des domaines bien balisés comme la reconnaissance de caractères. Il est alors nécessaire d'avoir une approche empirique pour aborder le problème, tout en sachant qu'il peut ne pas être résolu par un réseau de neurones.

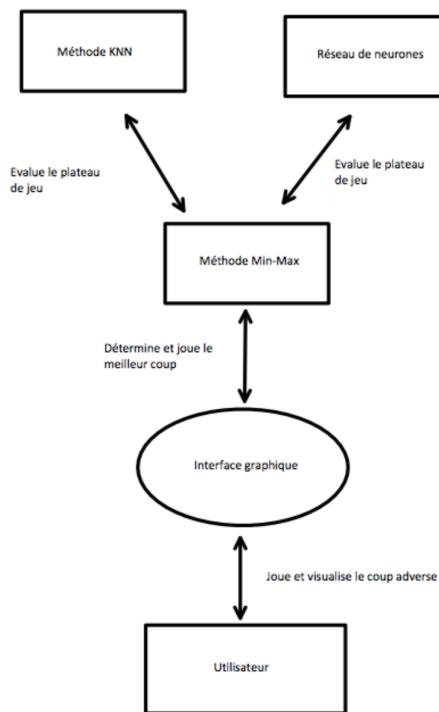
Pour la prédiction utilisant la méthode KNN (K plus proches voisins) nous utiliserons la bibliothèque libre Scikit-learn, spécialisée dans les algorithmes de classification. C'est une méthode plus «simple» que celle du réseau neuronal mais il se peut qu'elle soit plus adaptée à notre cas.

Le produit de notre projet est constitué de trois programmes. Le premier est un jeu d'échec muni d'une interface graphique que nous avons entièrement

développé ainsi que d'une intelligence artificielle. Pour réaliser l'interface graphique, nous utilisons la bibliothèque PyQt, développée par Riverbank Computing. C'est un logiciel libre distribué sous deux licences, la licence gratuite est sous licence GNU GPL (ce qui oblige ceux qui l'utilisent à mettre leurs programmes sous la même licence, et dans ce cas à rendre code source disponible), la licence commerciale permet de revendre son programme. Une version LGPL de PyQt existe (PySide), elle est développée par Nokia, et permet alors de s'affranchir du caractère héréditaire de la licence GPL (il devient donc possible de produire du code propriétaire). Comme dans ce projet nous ne souhaitons pas produire de code propriétaire et que la documentation est plus fournie sur PyQt, nous avons choisi ce dernier. Cependant, la transition entre PyQt et PySide est aisée. Ce programme sera développé en Python, il sera donc multi-plateforme sous réserve d'avoir un interpréteur Python installé (comme c'est le cas sur Windows, macOS ou Linux). Le second programme sera un réseau de neurones développé en Python avec Tensorflow et Keras.

Le troisième programme sera donc un programme de prédiction utilisant la méthode KNN, développé avec Scikit-learn et adapté à notre cas.

Voici un schéma récapitulatif rapide :



## 2 Conception détaillée

### 2.1 Regroupements modulaires

Nous avons décidé de séparer notre programme en différents modules :

- echeecs.py est le code initial de notre programme, c'est la base du moteur de jeu.
- gui.py est l'une des interfaces graphique de notre programme, elle permet à deux joueurs de jouer l'un contre l'autre.
- alphabeta.py contient les méthodes utilisées pour l'algorithme minimax par élagage alpha-bêta à la base de notre intelligence artificielle.
- IA\_denombrement.py contient les méthodes utilisées pour l'intelligence artificielle dite de «dénombrement».
- IA\_nn.py contient les méthodes utilisées pour l'intelligence artificielle utilisant notre réseau de neurones.
- IA\_knn.py contient les méthodes utilisées pour l'intelligence artificielle utilisant la méthode des k plus proches voisins.
- IA\_mcts.py contient les méthodes utilisées pour l'intelligence artificielle utilisant la méthode de parcours d'arbre de Monte Carlo.
- moveIAs.py recense toutes les fonctions de mouvement des pièces par les différentes intelligences artificielles.
- gui\_IA\_denombrement.py, gui\_IA\_nn.py, gui\_IA\_knn.py et gui\_IA\_mcts.py sont les différentes interfaces graphiques permettant de jouer contre les différents types d'intelligence artificielle.

### 2.2 Moteur de jeu

Le code initial echeecs.py de notre programme est notre moteur de jeu : il définit le plateau de jeu, les différentes pièces, les outils permettant de référencer leur position, ainsi que les différentes règles du jeu (mouvements possibles, mise en échec, mise en échecs et mat, ...).

Nous avons choisi de représenter le plateau de jeu de la façon suivante :

	0	1	2	3	4	5	6	7	8	9	10	11
0	-15	-15	-15	-15	-15	-15	-15	-15	-15	-15	-15	-15
1	-15	-15	-15	-15	-15	-15	-15	-15	-15	-15	-15	-15
2	-15	-15	-4	-2	-3	-5	-6	-3	-2	-4	-15	-15
3	-15	-15	-1	-1	-1	-1	-1	-1	-1	-1	-15	-15
4	-15	-15	0	0	0	0	0	0	0	0	-15	-15
5	-15	-15	0	0	0	0	0	0	0	0	-15	-15
6	-15	-15	0	0	0	0	0	0	0	0	-15	-15
7	-15	-15	0	0	0	0	0	0	0	0	-15	-15
8	-15	-15	1	1	1	1	1	1	1	1	-15	-15
9	-15	-15	4	2	3	5	6	3	2	4	-15	-15
10	-15	-15	-15	-15	-15	-15	-15	-15	-15	-15	-15	-15
11	-15	-15	-15	-15	-15	-15	-15	-15	-15	-15	-15	-15

Chaque pièce est représentée par une valeur dans un tableau carré de 12 cases, les pièces blanches correspondant aux valeurs strictement positives (par exemple : le cavalier est de valeur absolue 2 et le roi de valeur absolue 6) et inversement pour les pièces noires, les valeurs nulles du tableau correspondent aux cases vides du plateau. Enfin, nous avons arbitrairement choisi la valeur -15 pour caractériser les cases inaccessibles afin de pouvoir plus facilement déterminer les cases accessibles par une pièce. Nous utilisons deux couches de telles cases pour pouvoir empêcher tout problème de débordement dans le traitement du mouvement du cavalier pouvant bouger de manière horizontale ou verticale de deux cases.

De plus nous répertorions la position de chaque pièce individuellement (et non plus par valeur comme cavalier ou pion) dans une liste propre à sa couleur : sa position dans cette liste caractérise alors son individualité. Cela permet de connaître instantanément la position et la nature de chaque pièce et d'éviter ainsi de nombreuses confusions dans le traitement individuel de chaque pièce. Cependant, modifier les valeurs d'une liste sans modifier son ordre peut être très compliqué, c'est pourquoi nous utilisons également un dictionnaire dont les clés seraient les positions des pièces et ses valeurs la position de cette pièce dans la liste des positions.

Enfin de nombreuses fonctions définissant les règles du jeu sont rajoutées : on peut citer parmi elles la fonction permettant d'initialiser la configuration du plateau, la fonction permettant de connaître les valeurs accessibles par une pièce, la fonction permettant de connaître si un joueur est en échec ou en échec et mat, ainsi que la fonction permettant de bouger une pièce et qui actualise les différents réertoires.

### 2.3 Interface graphique

Pour réaliser l'interface graphique, nous utilisons la bibliothèque PyQt.

Voici un rendu du plateau de jeu utilisé :



Nous avons récupéré les images des pièces et du plateau en détournant avec Photoshop une capture d'écran d'un jeu iOS que nous avons trouvé particulièrement esthétique (développé par Optime Software). Pour le rendre public il faudrait donc demander l'accord du développeur, ou utiliser des éléments graphiques libres de droit.

Il est possible dans cette interface de sélectionner des pièces et de les mettre en surbrillance (pion blanc sur la capture d'écran). Voici un extrait du code de l'interface graphique, il s'agit de la méthode appelée lorsque l'on clique sur une pièce (qui est un bouton du point de vue de PyQt) :

```
# Fonction d'action lorsqu'une pièce est sélectionnée

def chk(self, button): # on passe en argument le sender
    global pChecked # pChecked est la pièce sélectionnée
    name = button.objectName()
    if pChecked == "": # Aucune pièce sélectionnée
        print(name + " sélectionnée")
        button.setIcon(QtGui.QIcon(name + "sel.png")) # on affiche la pièce rouge
        pChecked = button
    else: # Une pièce est déjà sélectionnée
        if button == pChecked: # On a sélectionné la pièce déjà sélectionnée
            print(name + " dé-sélectionné")
            button.setIcon(QtGui.QIcon(name + ".png")) # on affiche la pièce normale
            pChecked = ""
        else: # On a sélectionné une autre pièce
            print(pChecked.objectName() + " dé-sélectionné")
            print(name + " sélectionnée")
            pChecked.setIcon(QtGui.QIcon(pChecked.objectName() + ".png"))
            pChecked = button
            namePChecked = pChecked.objectName()
            pChecked.setIcon(QtGui.QIcon(namePChecked + "sel.png"))
```

On utilise pour le nom des pièces une convention très pratique. Par exemple, la tour blanche qui est à gauche au départ est appelée «tour1B», ce qui donne «tour2B» pour la noire. Ce nom est passé en argument (appelé «button»). Le nom du fichier de l'image associé est obtenu en rajoutant l'extension «.png» au nom de la pièce. Enfin, pour obtenir le nom du fichier image lorsque la pièce est sélectionnée, il suffit de rajouter l'extension «sel.png».

La prochaine étape consiste à relier l'interface graphique au moteur de jeu. Par exemple, lorsqu'une pièce est sélectionnée, nous mettons en rouge les pièces accessibles (qui sont déjà stockées dans une liste par le moteur de jeu), et une fois le mouvement validé, l'effectuer graphiquement. Pour cela, il suffit de changer les images des boutons associés. En définitive, les boutons de l'interface graphique sont fixes (ils ne correspondent réellement à des pièces qu'au départ), et les pièces sont représentées par les images de ces boutons, que l'on change en fonction des déplacements demandés. Cela donne ainsi concrètement :



De plus, grâce aux méthodes implémentées dans le moteur de jeu, l'interface graphique nous oblige à jouer des coups qui sont non-seulement valides, mais qui empêchent notamment une fin de partie prématurée. En effet, dans l'exemple suivant, le roi noir est mis en échec par le fou blanc. Ainsi l'interface graphique propose seulement les coups qui sortent de l'échec le roi noir. Il existe ainsi deux telles possibilités pour le cavalier noir :



En conclusion, grâce à ce simple programme, deux joueurs sont capables de jouer l'un contre l'autre. Une fin de partie est visible par l'impossibilité de quelconque mouvement d'un des joueurs.

## 2.4 Algorithme minimax par élagage alpha-bêta

### 2.4.1 L'algorithme minimax

L'algorithme minimax s'applique aux jeux opposants deux adversaires et à somme nulle : c'est à dire qu'un premier joueur cherche à maximiser un certain gain tout en sachant que le second cherche à minimiser ce même gain. Il paraît ainsi naturellement adapté au jeu d'échec.

Mais afin de pouvoir caractériser la valeur d'un certain gain, il faut mettre en place une fonction d'évaluation. La fonction d'évaluation prend comme paramètre une configuration de plateau et en détermine une valeur que l'on appelle le gain. Son but est de quantifier l'avantage d'un joueur par rapport à un autre. Dans le cas du jeu d'échec, cela peut par exemple consister à faire la différence entre les pions de l'adversaire mangés, et les pions que l'on s'est fait mangé (chaque pièce possédant une valeur plus ou moins importante, la dame ayant la plus forte et les pions la plus faible) : c'est cette évaluation dite de «dénombrément» que l'on utilise de manière initiale pour notre intelligence artificielle.

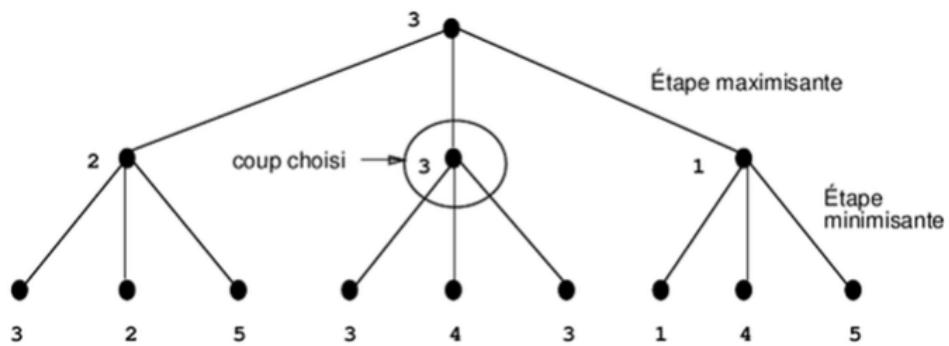
Le gain absolu de cette évaluation pour chaque pièce est répertorié dans le tableau suivant :

Roi	Reine	Tour	Fou	Cavalier	Pion
0	9.5	5	3.5	3	1

Ainsi, de façon théorique, si l'on veut déterminer le meilleur mouvement possible en prenant en compte les n prochains tours, l'algorithme construit tout d'abord un arbre de profondeur n de toutes les configurations possibles (chaque nouveau noeud descendant correspond à un mouvement possible à partir de la configuration actuelle du noeud). On note ainsi que les noeuds de profondeur paire appelés «noeuds max» sont ceux correspondant aux mouvements du premier joueur (qui cherche à maximiser son gain), alors que les noeuds de profondeur impaire appelés «noeuds min» sont ceux correspondants aux mouvements du deuxième joueur (qui cherche à minimiser les gains du premier). Les noeuds de profondeur n contiennent alors le gain d'évaluation du plateau pour chaque configuration.

L'algorithme remonte alors l'information de gain aux noeuds parents en considérant qu'il faut maximiser le gain possible aux noeuds max et le minimiser aux noeuds min. Cela revient à considérer le meilleur mouvement possible du premier joueur sachant que le deuxième joueur joue également le meilleur mouvement possible de son côté. Cela détermine alors le prochain coup à choisir pour le premier joueur et permet de représenter de manière cohérente l'intelligence humaine d'un jeu d'échec.

Un exemple de parcours est décrit dans l'exemple suivant, minimalisté car ne prenant en compte que 3 mouvements possibles à chaque étape :



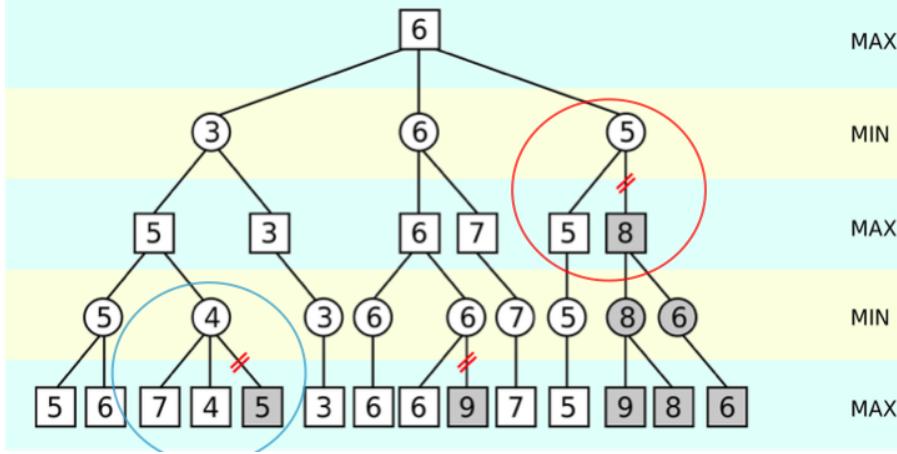
En pratique, faire un arbre de profondeur supérieur à 3 serait irréaliste, car incrémenter la profondeur d'un arbre correspondrait en moyenne à multiplier par 35 son nombre de feuilles.

### 2.4.2 L'élagage alpha-bêta

L'élagage alpha-bêta est une technique d'optimisation de l'algorithme minimax permettant d'éviter le parcours de branches inutiles.

Cela consiste brièvement à conserver et actualiser une borne inférieure  $\alpha$  et une borne supérieure  $\beta$ . Si le gain du noeud descendant n'appartient pas à cet intervalle, alors on ne parcourt pas cette branche de l'arbre.

Un exemple d'un tel parcours d'arbre est représenté dans l'exemple suivant :



Dans le cercle bleu, on voit que le noeud min concerné parcourt tout d'abord la branche de gain 7. Puisque c'est un noeud min, l'intervalle de valeurs pertinentes pour elles est donc  $[-\infty ; 7]$ . Après avoir parcouru le noeud de gain 4, ce dernier devient  $[-\infty , 4]$  : il est donc inutile de parcourir le noeud de gain 5.

De même, dans le cercle rouge, après avoir parcouru le noeud de gain 5, son intervalle de pertinence est  $[-\infty ; 5]$  : inutile de parcourir le reste. L'élagage alpha-bêta permet alors d'éviter le parcours d'une partie non-négligeable de l'arbre.

## 2.5 Réseau de neurones artificiels

### 2.5.1 Développement et historique des réseaux de neurones artificiels

En 1997, le champion du monde d'échecs Garry Kasparov a été battu pour la première fois par DeepBlue, un algorithme développé par IBM utilisant une technique de force brute calculant tous les coups possibles pour choisir le meilleur. Cette victoire de la machine sur l'homme a marqué les esprits, mais jusqu'à

récemment, certains jeux comme le Go, en raison d'un nombre très grand de possibilités de coups, n'ont pu être maîtrisés par des algorithmes, nécessitant une nouvelle approche.

En mars 2016, le champion du monde de Go Lee Sedol a été battu par AlphaGo, développé par Google : cette avancée a été élue parmi les révolutions de l'année par la revue américaine Science. AlphaGo est le produit d'une branche émergente de l'informatique : l'apprentissage automatique (« deep learning » en anglais) Les ambitions de l'apprentissage automatique sont nombreuses : diagnostics médicaux, synthèse vocale, prise de décision. Cette branche répond notamment aux problèmes d'analyse qui ne pouvaient pas être traités par les algorithmes usuels pour qui la quantité et la complexité des données étaient un obstacle. Au contraire les techniques d'apprentissage automatique, avec une logique différente des programmes "classiques", tirent profit de l'abondance des données. L'exemple sur lequel nous avons travaillé est la reconnaissance de chiffres. À l'heure actuelle cette thématique est bien connue, les leaders du domaine travaillent sur des objets beaucoup plus complexes comme des photographies de paysages.

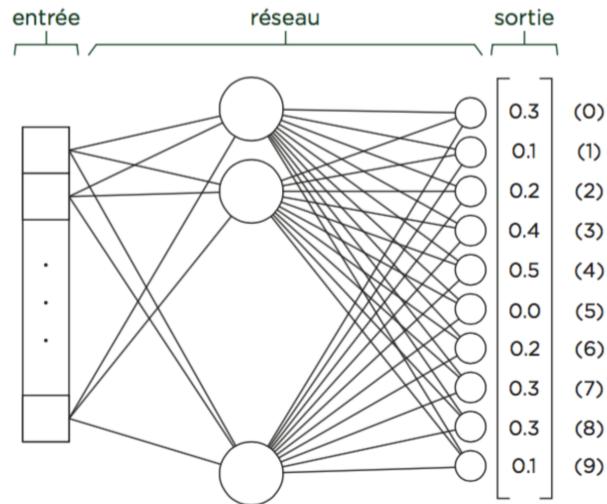
Il existe deux principaux types d'apprentissages automatiques : l'apprentissage supervisé, où l'on possède des données déjà étiquetées c'est à dire des exemples dont on connaît le résultat, et l'apprentissage non-supervisé, où l'on ne connaît pas à l'avance les résultats des exemples fournis. En travaillant avec une base de données déjà étiquetées, notre étude s'inscrit dans l'apprentissage supervisé, que nous avons choisi d'implémenter à l'aide d'une technique majeure de l'apprentissage automatique : un réseau de neurones.

Le concept de neurone formel apparaît avant le développement de l'informatique. En 1943 Mac Culloch et Pitts développent un modèle mathématique proche des neurones biologiques qui ne connaissent que deux états : activé ou non. Les neurones formels sont interconnectés : ils reçoivent des entrées provenant d'autres neurones, ou directement de la donnée à traiter (exemple : valeur d'un pixel pour une image) qui détermineront alors s'il s'active ou non. De plus, on affecte un poids à chaque liaison, ce qui permet de rendre compte de son importance dans le réseau.

C'est ainsi qu'en 1958 Rosenblatt, en tentant de reproduire le processus rétinien, assemble les neurones en une couche d'entrée et une couche de sortie : c'est l'origine du perceptron (réseau de neurones à couches successives). L'idée est abandonnée dans les années 70 : faute de puissance de calcul et de connaissance théorique suffisante, le modèle du réseau de neurone stagne. Il faudra attendre le début des années 90 pour voir les réseaux de neurones revenir sur la scène de la recherche en intelligence artificielle. Notamment grâce à la technique d'optimisation de Hopkins (1982) appelée "rétropropagation du gradient". On peut ainsi résumer la conception d'un réseau de neurones en trois temps :

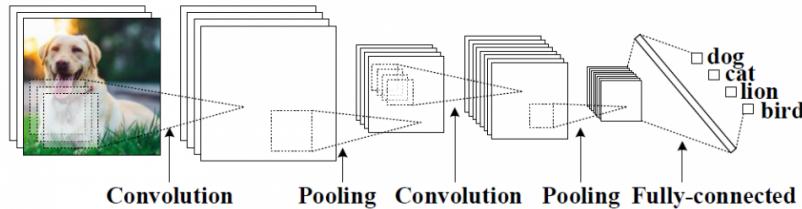
- L'initialisation du réseau avec le choix d'une architecture (nombre de couches et de neurones) et des poids initiaux.
- L'apprentissage : en s'appuyant sur un échantillon des données étudiées, on modifie pas à pas, en appliquant la méthode d'Hopkins, les poids des connexions entre neurones pour se rapprocher du maximum de réussite (le but étant de minimiser l'erreur commise).
- Enfin, on teste le réseau en lui donnant le reste des données qu'il rencontre alors pour la première fois

Voici comment est schématisé un réseau de neurones artificiel, où l'on retrouve les différentes parties expliquées précédemment :



Intéressons nous à la couche de sortie du réseau de neurones représenté. Il s'agit d'un exemple classique, celui de la reconnaissance de chiffres manuscrits par un réseau de neurones (rendu possible par la base de données MNIST). Dans cette colonne de sortie, on a pour chaque indice la probabilité que l'image donnée en entrée soit classifiée comme un 1, un 2, un 3, etc. Ainsi, la classification finale de l'image est donnée par l'indice de probabilité maximale.

Une autre technique importante est le réseau de neurone convolutionnel, inventé en 1998 par Yann Lecun et Yoshua Bengio. Voici un schéma de fonctionnement d'une couche de convolution :



Il s'agit ici de classer une image d'animal dans une des catégories chien, chat, lion ou oiseau. Pour réaliser cette classification, l'oeil et le cerveau humain chercheraient à discriminer les différentes images en repérant des caractéristiques distinctes (appelées "features"). Par exemple si l'on voit des ailes, il est très probable qu'il s'agisse d'une image d'oiseau. Nous cherchons à imiter ce procédé de reconnaissance de motifs. Pour cela, nous utilisons l'opération mathématique de convolution (utilisée également largement en traitement de signal) entre les données d'entrées et une matrice spéciale appelée filtre. C'est le résultat de cette opération qui détermine si un motif est présent ou non dans l'image. Ensuite vient une opération de "pooling" permettant de réduire la dimension du résultat. À l'issue des deux opérations de convolution, on obtient des données des tailles réduites dont les caractéristiques ont été extraites. Ceci permet au réseau de neurone qui suit ("fully-connected") de classifier encore plus efficacement les images. Cette technique de convolution est très utilisée dans la reconnaissance d'image, la reconnaissance vocale et le traitement automatique du langage naturel ("NLP").

### 2.5.2 Intérêt des réseaux de neurones artificiels dans notre projet

Nous avons vu précédemment que la technique minimax était efficace pour développer une intelligence artificielle, mais elle repose sur un aspect fondamental du jeu d'échecs : la fonction d'évaluation. C'est cette fonction qui attribue un score à une position donnée du tableau, permettant de savoir quantitativement à quel point la position est favorable aux noirs ou aux blancs. Cette évaluation repose en général sur des heuristiques qui ont été formulées par des humains ("le roi noir est protégé par des pions", "tour et roi blanc ont roqué"...). Plus la fonction d'évaluation est efficace, plus la technique minimax fonctionne. Les meilleures intelligences artificielles comme Stockfish sont fondées sur des milliers d'heuristiques développées par les plus grands maîtres d'échecs.

On sait que les réseaux de neurones artificiels sont capables de reconnaître des motifs (via la technique de convolution par exemple) sur un jeu de données pour en extraire des informations. Notre idée est alors d'utiliser cette technique d'apprentissage automatique supervisé pour évaluer les positions.

### 2.5.3 Construction du jeu de données

#### 2.5.3.1. Téléchargement et nettoyage des données

Comme dans toute technique d'apprentissage automatique, il est vital d'avoir des données structurées et de qualité. Les données sont divisées en deux bases : la première contient les différentes parties avec la liste de tous les coups, et la deuxième contient les évaluations données par Stockfish (meilleure intelligence artificielle à ce jour) pour chaque coups de la première base de données. L'idée est donc d'entraîner notre réseau de neurones sur des données étiquetées par Stockfish pour obtenir à la sortie une intelligence artificielle beaucoup plus légère car elle ne contient que le réseau entraîné, sans la base de données (c'est à dire la structure du réseau et les matrices des poids).

Nous utilisons une base de données extraite du site ChessBase contenant 40 000 parties, soit environ 4 million de coups (positions de plateau). Chaque coup a été analysé par Stockfish, et l'évaluation correspondante est stockée dans la deuxième base de données (celle des étiquettes ou "labels"). L'apprentissage étant supervisé, il faut diviser les données en deux : une partie pour l'entraînement du réseau, une autre pour le tester sur des données qu'il n'a jamais vu. Nous avons choisi une base d'entraînement de 3 millions de parties et une base de test de 1 million de parties.

Les parties téléchargées sont au format PGN (Portable Game Format), un standard de codage de parties d'échecs. Voici un aperçu du contenu d'un des fichiers que nous utilisons :

```
[Event "?"]
[Site "?"]
[Date "2017.12.14"]
[Round "1"]
[White "Stockfish 101217 64 BMI2"]
[Black "Stockfish 101217 64 BMI2"]
[Result "0-1"]
[ECO "E10"]
[Opening "Queen's pawn game"]
[PlyCount "120"]
[TimeControl "1+0.1"]

1. d4 {book} Nf6 {book} 2. c4 {book} e6 {book} 3. Nf3
{book} d5 {book}
4. Nc3 {book} dxc4 {book} 5. Qd4+ {book} c6 {book} 6.
0xc4 {book} b5 {book}
7. Qb3 {book} Nbd7 {book} 8. Bg5 {book} Qa5 {book} 9.
e3 {+0.57/13 0.30s}
a6 {+0.02/14 0.33s} 10. Be2 {-0.05/14 0.26s} c5
{-0.01/13 0.075s}
11. 0-0 {-0.01/12 0.056s} Bb7 {+0.03/13 0.22s} 12.
Rfd1 {+0.15/13 0.19s}
h6 {-0.05/13 0.21s} 13. Bf4 {+0.19/14 0.27s} Be7
{+0.02/14 0.26s}
14. a3 {+0.16/13 0.11s} Rc8 {+0.01/13 0.13s} 15. Ne5
{-0.20/14 0.33s}
Nx5 {+0.24/13 0.16s} 16. dx5 {-0.10/12 0.043s} Nd7
{+0.45/11 0.031s}
17. Qc2 {-0.41/15 0.16s} Qc7 {+1.03/12 0.063s} 18.
Ne4 {-0.47/14 0.054s}
Bxe4 {+0.44/17 0.32s} 19. 0xe4 {-0.56/15 0.11s} c4
{+0.46/14 0.037s}
20. Rab1 {-0.49/14 0.088s} Nc5 {+0.54/14 0.091s} 21.
Qf3 {-0.59/14 0.14s}
0-0 {+0.64/13 0.036s} 22. Qg3 {-0.78/14 0.10s} Kh8
{+0.49/15 0.14s}
```

Ici nous avons les données correspondants à une partie. Dans le premier tableau se trouvent les différents coups dans la notation algébrique longue, et le

second contient leurs évaluations.

Ce qui nous intéresse est le mouvement (par exemple, "d4" ou "Nf6"), et son évaluation, en "pawns". Ces données, inexploitables en l'état, doivent être nettoyées. Il faut d'abord convertir les notations de mouvements, ici en notation algébrique abrégée, en notation algébrique longue ("Long Algebraic Notation"), plus pratique à utiliser par la suite. Nous utilisons pour la conversion l'outil PGN-extract. Avec cet outil, nous éliminons également toutes les parties qui ne finissent pas en échec et mat. Il reste maintenant à enlever les en-têtes, et à extraire uniquement les mouvements et leurs évaluations. Ceci est particulièrement pénible car les ouvertures n'ont pas d'évaluation (ce qui est logique car cela n'a pas de sens aux échecs), et le fichier brut contient de nombreuses particularités comme des sauts de lignes ou des commentaires additionnels à chaque fin de partie. Pour plus de lisibilité, les scripts de nettoyage des données sont dans un notebook Jupyter nommé "PGN Explorer.ipynb". Voici à quoi ressemblent les données nettoyées :

On a ici toutes les données pour une partie. Le premier tableau contient les mouvements (ce sont des chaînes de caractère) et le deuxième contient les évaluations associées à chacun de ces mouvements (ce sont des flottants). Les ouvertures sont toutes classiques donc elles ont été évaluées à 0 (aucun avantage), et on remarque bien qu'en fin de partie un joueur se démarque avec une évaluation à 100 (certain de faire échec et mat).

### 2.5.3.2 Formatage des données pour le réseau de neurones

En ce qui concerne les positions à évaluer, nous avons choisi de représenter comme précédemment les plateaux de jeux par un tableau complet de taille 64 contenant les positions, les couleurs et les valeurs de chaque case. Donner les notations algébriques n'aurait aucun sens car elles ne sont pas porteuse de sens seules, un plateau donné est la succession de plusieurs coups et non d'un seul (sauf le premier!). Pour cela nous avons fait un script qui simule une partie, et qui à chaque déplacement génère le plateau correspondant et l'intègre dans une liste. Ainsi, les données en entrées sont constituées d'une liste de n configuration de plateau, une configuration de plateau étant une liste de taille 64 contenant des flottants. Voici un exemple montrant les deux premiers éléments du "dataset" :

```

import numpy as np
tab = np.load("newGamesData.npy")
print(tab[0:2])

[[-5.   0.  -3.5  0.   0.  -3.5  0.  -5.  -1.   0.   0.  -3.   0.  -1.  -1.
 -1.   0.   0.  -1.   0.  -1.  -3.   0.   0.  -9.  -1.   0.   0.   0.   0.
 3.5   0.   0.   0.   0.   1.   0.   0.   0.   0.   0.   9.   3.   0.   1.
 3.   0.   0.   1.   1.   0.   0.   0.   1.   1.   1.   5.   0.   0.   0.
 0.   3.5  0.   5.  ],
[-5.   0.  -3.5  0.   0.  -3.5  0.  -5.  0.   0.   0.  -3.   0.  -1.  -1.
 -1.  -1.   0.  -1.   0.  -1.  -3.   0.   0.  -9.  -1.   0.   0.   0.   0.
 3.5   0.   0.   0.   0.   1.   0.   0.   0.   0.   0.   9.   3.   0.   1.
 3.   0.   0.   1.   1.   0.   0.   0.   1.   1.   1.   5.   0.   0.   0.
 0.   3.5  0.   5.  ]]

```

Il faut maintenant formater les labels correctement. Nous les divisons en 6 classes en fonction de la valeur de l'évaluation, donnée en pawns. La classe 1 correspond aux coups très défavorables au joueur, la classe 6 correspond aux meilleurs coups. La valeur de la classe n'étant d'aucune importance pour le réseau de neurone, nous allons utiliser l'encodage one-hot. C'est à dire, plutôt que d'étiqueter une position comme étant n, on l'étiquette avec une liste de n valeurs, avec des 0 partout sauf à la n-ième position où l'on écrit 1. Ceci permet d'utiliser des classes sans donner de poids à leurs dénominations respectives. Voici un exemple des labels ainsi obtenus :

```

import numpy as np
tab = np.load("newOneHotEncoded.npy")
print(tab[20:23])

[[0 1 0 0 0 0]
 [0 0 1 0 0 0]
 [0 0 1 0 0 0]]

```

Ainsi, le premier coup de cet échantillon appartient à la classe 2, et les autres à la classe 3. Ce ne sont donc pas des coups favorables.

Après l'étape finale du one-hot encoding, les labels que nous utiliserons dans le réseau de neurones sont constitués d'une liste de n listes (autant que de positions retenues) contenant chacune 6 entiers.

De même, les scripts de formatage des données sont dans un notebook Jupyter appelé "Data Builder for Neural Network.ipynb"

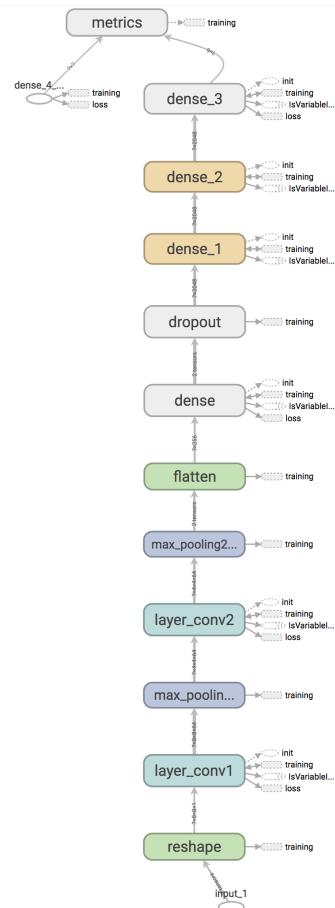
#### 2.5.4 Mise en place du réseau de neurones

Nous avons choisi la librairie Tensorflow, développée par Google, avec Keras, développée par François Chollet, pour mettre en pratique notre réseau de neurones. Ce sont toutes les deux des librairies open-source.

Dans Tensorflow, on crée d'abord un graphe computationnel statique (on ne

peut plus le modifier pendant l'exécution, contrairement à d'autres librairies comme PyTorch). Ce graphe contient des récepteurs de données et des opérations (comme des couches de neurones). Les données coulent dans le graphe et les calculs ne sont réalisés qu'au moment de l'exécution du graphe. L'avantage d'utiliser un tel graphe est de pouvoir répartir les calculs sur différentes machines afin de paralléliser le calcul et gagner du temps. En particulier lorsque l'on utilise des réseaux de neurones, l'utilisation d'un GPU plutôt qu'un CPU (même très bon) est primordiale pour ne pas perdre de temps lors de l'apprentissage. En effet les milliers de coeurs des GPU permettent de paralléliser beaucoup d'opérations (comme le produit matriciel entre les données d'entrée et les matrices de poids), alors qu'un CPU exécute séquentiellement ces opérations avec moins de coeurs. Pour ce projet nous avons donc utilisé un GPU NVIDIA GeForce GTX 1080 Ti prêté par notre tutrice Mme Brunet que nous remercions.

Voici le graphe computationnel de notre réseau, obtenu avec Tensorboard :



Les données ("input") entrent dans le graphe par le bas et sont redimensionnées (par le noeud "reshape") avant la convolution. S'ensuit deux couches de convolution séparées à chaque fois par une étape de "max pooling" (réduction de dimension). Les données sont ensuite remises dans le format précédent où elles sont "aplaties" (étape "flatten"), puis sont servies aux quatre couches de neurones (dont le nom commence par "dense"). Les trois premières sont constituées de 2048 neurones, et entre celles-ci nous avons ajouté un "dropout". Cette opération permet de réduire l'overfitting en parcourant la couche "dense" de manière stochastique (chaque neurone a une probabilité 0,8 d'être utilisé). La dernière couche de taille 6 permet de réduire la dimension afin d'obtenir en sortie une colonne de 6 éléments correspondant aux probabilités d'être dans chaque classe. Le dernier noeud appelé "metrics" calcule des informations utiles comme la précision.

L'utilisation de Tensorflow devient plus complexe et rigide lorsque l'on a de nombreuses couches de neurones ou de convolution, car il faut calculer à la main les tailles de celles-ci. La librairie Keras apporte une surcouche de plus haut niveau et permet de le faire automatiquement et d'avoir un code plus court et plus lisible.

## 2.6 Évaluation par la méthode des k plus proches voisins

La méthode des K plus proches voisins est une méthode d'apprentissage supervisé. Elle se base sur l'utilisation d'une base de donnée (base de donnée "entraînement") qui va lui servir à apprendre à déterminer la qualité d'un coup. Le but de cette méthode est donc, pour chaque nouvelle situation, de déterminer les situations similaires qu'il connaît déjà, et parmi celles-ci, déterminer qu'elle est la plus avantageuse. Dans notre cas, pour déterminer quelles sont les situations similaires, l'algorithme va calculer la distance euclidienne entre l'organisation de plateau donné et toutes les autres organisations de plateau qu'il a déjà rencontré. Logiquement, celles dont les distances avec l'organisation rencontrée seront les plus petites seront désignées comme les situations "voisines". Le "K" est un entier qui permet de dire à l'algorithme combien de K-voisins il doit considérer. Pour trouver ce K, il n'existe aucune méthode théorique, la seule façon de procéder est de tester pour chaque K puis de ne garder que le K pour lequel la précision est la plus grande. Concrètement, nous donnons à l'algorithme une base de donnée "entraînement" dans laquelle les coups sont étiquetés et une base de test avec laquelle l'algorithme va s'entraîner et apprendre ensuite à discerner les bons des mauvais coups.

Comme précédemment, la qualité d'un coup est en fait la "mesure" de l'avantage qu'il donne à celui qui l'effectue sur son adversaire. Ainsi nous allons continuer à utiliser les deux bases de données précédentes (OneHotEncoded and GamesData). Nous utilisons la librairie python libre de droit Scikit-learn spécialisée dans les algorithmes de classification.

Nous allons donc définir une méthode qui prend en argument deux entiers a et b qui seront les bornes de l'intervalle sur lequel on va chercher le K optimal. Ensuite nous importons les librairies nécessaires : numpy, matplotlib et Scikit-learn.

```
fonctionKNN_Method(a, b) :
importnumpy
importmatplotlib
importsklearn
```

Par la suite nous allons charger les 2 bases de données qui nous seront utiles : GamesData qui recense tous les coups (organisations de plateau consécutives) et OneHotEncoded qui contient les notes correspondantes allant de 0 à 6. La première chose est d'indiquer à l'algorithme quels sont les labels et quelles sont les données. Les données seront donc la totalité de la base de donnée GamesData et les labels seront en fait la position du 1 dans la base de données OneHotEncoded car comme les notes ont été «one-hot encoded», la position du "1" dans la liste donne en fait la note attribuée. La fonction argmax de Numpy permet de nous donner la position.

Maintenant que nous avons signalé à l'algorithme quelles données prendre en compte, nous allons, à partir de ces deux bases de données créer la base d'entraînement et la base de test. Pour cela, nous utilisons la fonction train\_test\_split de la bibliothèque Scikit-learn. Elle prend en argument la base de données contenant les données ( X ), celles contenant les labels ( y ) et enfin le pourcentage de la base que nous allons utiliser (un nombre entre 0 et 1) nous prendrons ici 30% soit 0,3. On a donc :

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3)
```

Maintenant, nous allons calculer les taux de précision avec les K compris dans les bornes que nous avons fixé. Nous rentrons dans une boucle for dont l'indice ( i )va prendre des valeurs entières allant de a à b. Une fois dans la boucle nous allons utiliser les fonctions fournies par Scikit-learn. En premier lieu, nous allons créer une classe de KNeighboorsClassifier de sklearn. Cette classe prend uniquement en argument la valeur du K, ici notre indice. L'étape d'après consiste en la normalisation de X\_train et y\_train. Cette normalisation est nécessaire en machine learning pour que les algorithmes fonctionnent correctement. Nous utiliserons pour ce faire la fonction fit de sklearn qui prend en argument un élément de la classe précédente et les bases de données et labels d'entraînement (X\_train et y\_train). On obtient les lignes suivantes :

```
KNeighboorsClassifier(i)
knn.fit(X_train,y_train)
```

Il ne reste maintenant plus qu'à effectuer la prédiction à l'aide de la fonction predict de sklearn.

Celle-ci prend en paramètre knn ainsi que la base de test X\_test. Maintenant nous calculons la précision à l'aide la fonction score de sklearn qui prend en argument knn, X\_test et y\_test. Enfin nous générerons le rapport final grâce à la fonction classification\_report prenant en argument la base de test et la base de prédiction (y\_test et y\_pred). Si la précision est la meilleure par rapport à celles effectuées précédemment, on la conserve, ainsi que son rapport final.

```
y_pred = knn.predict(X_test)
knn.score(X_test, y_test)
classification_report(y_test, y_pred)
(if knn.score is best)
    score = knn.score
    bestReport = classification_report
```

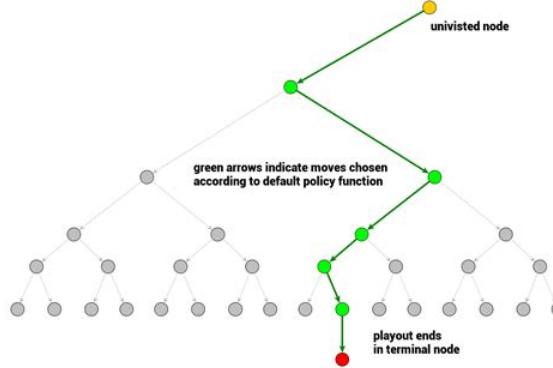
## 2.7 Méthode de recherche arborescente de Monte-Carlo

Les deux méthodes précédentes ont donc permis d'obtenir une fonction d'évaluation alternative à la méthode de dénombrement utile pour l'algorithme minimax. La méthode de Monte-Carlo possède une approche bien différente de celle de minimax, et répond à un de ses défauts qui est la nécessité de créer l'arbre complet des possibles, qui peut être excessivement grand. Comme son nom l'indique (le nom Monte-Carlo faisant référence à des processus aléatoires), cette méthode cherche à simuler un grand nombre de fois des parties et tente de prédire le mouvement le plus prometteur par rapport aux résultats trouvés.

L'un des premiers concepts de cette méthode est la recherche, qui consiste à trouver le noeud où effectuer une simulation de parties. Basiquement, il faudrait effectuer au moins une simulation pour chacun des premiers mouvements possibles à partir de la configuration initiale. Ainsi, tant que tous les premiers descendants de la racine n'ont pas été visités (ie que l'on y a effectué une simulation), la prochaine simulation sera nécessairement faite sur un noeud non-visité.

La simulation consiste en la succession aléatoire de mouvements jusqu'à arriver à un noeud terminal, ie dans le cas du jeu d'échecs, de tomber sur une configuration de plateau où il y a échec et mat. Ainsi, on aboutit à un certain résultat de simulation dépendant du joueur ayant gagné la partie, et après que le premier joueur ait joué un tel coup (ie choisi un tel premier noeud descendant). Pour nous, ce sera simplement 1 si le premier joueur gagne la partie, 0 si la partie est nulle, et -1 s'il la perd.

Un schéma de ce concept est le suivant :



Il faut ensuite rétro-propager cette information du noeud de départ jusqu'à la racine. En comptabilisant le nombre de fois où ce noeud  $\nu$  a été sur le chemin de propagation  $N(\nu)$ , ainsi que le nombre de succès ayant résulté d'une simulation à partir de ce noeud  $Q(\nu)$ , on peut obtenir le ratio de victoires obtenues à partir de ce noeud.

Mais que faire après que tous les premiers noeuds aient été visités au moins une fois ? Notre intuition nous pousse à vouloir explorer les noeuds ayant un haut ratio de réussite, mais aussi ceux qui ont été peu visités, à qui l'on n'aurait pas donné de nouvelles chances. C'est pour cela que l'on définit la fonction UCT (pour Upper Confidence Bound applied to Trees) définie de la manière suivante :

$$UCT(\nu_i, \nu) = \frac{Q(\nu_i)}{N(\nu_i)} + c\sqrt{\frac{\log N(\nu)}{N(\nu_i)}}$$

où  $\nu$  est le noeud courant et  $\nu_i$  un noeud descendant

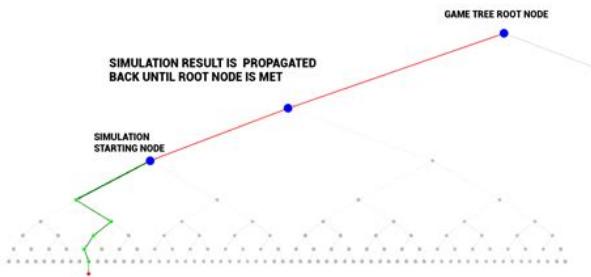
La première composante est appelée la composante d'exploitation, car elle exploite les résultats des simulations. La deuxième composante est appelée composante d'exploration, car elle favorise l'exploration des noeuds ayant été relativement peu visités.

Ainsi, pour définir la fonction permettant de choisir le prochain noeud où effectuer une simulation, on utilise l'algorithme suivant que l'on lance à la racine de l'arbre :

Tant que tous les descendants du noeud courant ont déjà été visités :

Le nouveau noeud courant est le noeud maximisant la fonction UCT  
On retourne un descendant non-visité du noeud courant

On peut résumer ce processus par le schéma suivant :



Mais alors quand est sensé ce terminer ce processus ? On remarque que ce processus n'est pas clairement défini dans le temps. En théorie, plus ce processus dure longtemps, plus les ratios de victoires et le nombre de visites s'affinent et se rapprochent statistiquement de leur valeur théorique, d'après la théorie des probabilités. Ainsi, c'est à nous de fixer le temps de calcul effectué pour un tel processus pour que ce dernier soit suffisamment significatif pour avoir une valeur exploitable mais également cohérent avec le temps d'attente humain d'un vrai joueur.

A la fin d'un tel processus, le descendant le plus prometteur est finalement le noeud qui a été visité le plus de fois (le noeud  $\nu_i$  où  $N(\nu_i)$  est le plus grand)

On peut ainsi résumer un tel programme par le code suivant :

```
def monte_carlo_tree_search(root,tmax):
    debut=time.time()
    fin = time.time()
    while (fin - debut) <= tmax:
        leaf=traverse(root)
        simulation_result=rollout(leaf)
        back_propagate(leaf,simulation_result)
        fin = time.time()
        echechs.scan(leaf.parent.name[0])
    return best_child(root)
```

où traverse est la fonction définie par le pseudo-code précédent, la fonction rollout est ici la fonction permettant de dérouler une partie aléatoire et d'en retourner le résultat, la fonction back\_propagate est la fonction de rétro-propagation du résultat, et enfin la fonction best\_child est la fonction retournant le noeud descendant le plus visité.

## 3 Tests et Analyses

### 3.1 Tests Unitaires

Afin d'effectuer les tests unitaires sur le code echeecs.py, nous avons décidé d'utiliser la librairie unittest de Python.

Cependant, nous nous sommes vite rendus compte que les tests que nous devions effectuer pouvaient être très fastidieux. En effet, tester de nombreuses configurations d'échec et d'échecs et mat par exemple, sachant qu'il faut répertorier de nombreux paramètres dans des dictionnaires et listes appropriés, devient vite irréalisable.

C'est pourquoi nous avons choisi de concentrer nos tests unitaires sur la fonction qui détermine les valeurs accessibles par la pièce située à une certaine position et qui est finalement à la base de notre moteur de jeu.

Ainsi, pour ce début de code, on teste si un pion et un cavalier situé aux coordonnées (6,6) a bien comme valeurs accessibles la liste de coordonnées stockées dans expected.

```
import echeecsTest
import unittest

class TestEcheecs(unittest.TestCase):

    def testValeursAccesiblesPions(self):
        echeecsTest.plateau[6][6]=1
        expected=[(5,6)]
        actual=echeecsTest.valeurs_accessibles(6,6)
        self.assertEqual(expected,actual,"failed test for pion")

    def testValeursAccesiblesCavaliers(self):
        echeecsTest.plateau[6][6]=2
        expected=[(5,4),(4,5),(4,7),(5,8),(7,4),(7,8),(8,5),(8,7)]
        actual=echeecsTest.valeurs_accessibles(6,6)
        expected.sort()
        actual.sort()
        self.assertEqual(expected,actual,"failed test for cavalier")
```

Grâce à la commande suivante, on enregistre nos résultats dans le fichier testResults.txt :

```
if __name__ == '__main__':
    log_file = 'testResults.txt'
    f = open(log_file, "w")
    runner = unittest.TextTestRunner(f)
    unittest.main(testRunner=runner)
    f.close()
```

Les résultats sont alors corrects :

```
.....
-----
Ran 6 tests in 0.002s
OK
```

## 3.2 Test et Analyse de nos programmes

**NB :** tous les tests de performance ont été effectués sur un MacBook Air, 8Go de RAM, processeur IntelCorei5, fréquence 1.6GHz.

### 3.2.1 Moteur de jeu et interface graphique

Dans son état actuel, l'intégration du moteur de jeu à l'interface graphique fonctionne : les mouvements des pièces sont corrects, la prise des pièces également et notre interface ne propose que des mouvements empêchant la mise en échec.

Nous n'avons cependant pas eu le temps d'intégrer à l'interface graphique le roque et la promotion de pièces, même si ces méthodes existent actuellement au sein du moteur de jeu.

L'utilisation importante des méthodes du moteur de jeu ont permis de corriger un très grand nombre d'erreur. Cependant, l'impossibilité d'effectuer des tests unitaires sur un nombre important de ses fonctions, ainsi que des erreurs possibles liées à l'intégration du moteur à l'interface graphique, empêchent de nous prononcer quant à l'exactitude totale de ces programmes.

### 3.2.2 L'algorithme minimax et élagage alpha-bêta

Comme expliqué précédemment, cet algorithme doit tout d'abord créer un arbre de parcours de l'ensemble des configurations possibles du plateau après 3 coups successifs. En moyenne, un joueur est capable de jouer environ 35 coups différents à chaque tour. Ainsi, en moyenne, notre arbre comportera  $35^3 = 42875$  feuilles.

Il doit ensuite parcourir cet arbre de façon optimisée grâce à l'algorithme alpha-bêta.

L'ensemble de ces processus a un temps de calcul non-négligeable, sachant que l'évaluation des plusieurs dizaines de milliers de feuilles finales est variable selon les méthodes.

Les différents temps de calcul d'évaluation (en secondes) pour chaque méthode sont référencés dans le tableau suivant :

Dénombrement	Réseau de neurones	K plus proches voisins
4.053e-06	0.09524	0.002647

Ainsi, pour la méthode de dénombrement, la création de l'arbre de profondeur 3 ainsi que son temps de parcours sont, pour un mouvement initial :

```
Temps de création de l'arbre (méthode dénombrement) :  
8.907292127609253  
Temps de parcours dans l'arbre  
0.6717550754547119
```

De même, pour la méthode du réseau de neurones :

```
Temps de création de l'arbre (Méthode du réseau de neurones):  
98.67834281921387  
Temps de parcours dans l'arbre  
0.5220258235931396
```

Et celle des k plus proches voisins :

```
Temps de création de l'arbre (Méthode knn) :  
38.21303606033325  
Temps de parcours dans l'arbre  
0.557995080947876
```

On remarque ainsi que les temps de calcul des évaluations influence grandement le temps de calcul d'un mouvement pour l'intelligence artificielle.

De plus, on remarque que le temps de parcours n'est pas influencé par le type d'évaluation et est très faible devant le temps de création de l'arbre : cela valide ainsi notre algorithme alpha-bêta.

Ces derniers calculs ont été effectués pour le premier mouvement de l'intelligence artificielle, où les mouvements sont au départ restreints. Si l'on avance un peu plus dans la partie, les pièces acquièrent plus de liberté ce qui augmente le nombre de possibilités et donc le temps de création de l'arbre et son parcours.

Après 4 coups :

```
Temps de création de l'arbre (méthode dénombrement) :  
17.53955316543579  
Temps de parcours dans l'arbre  
2.7331528663635254
```

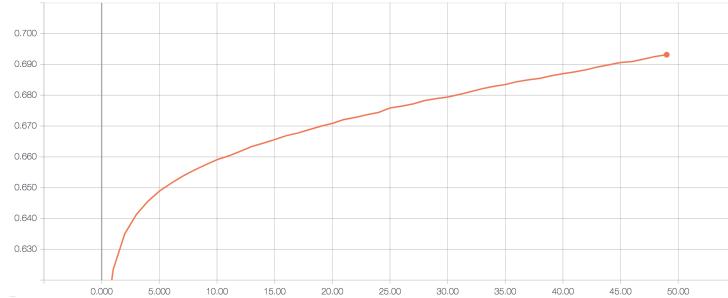
Ainsi, en conclusion, on peut dire que le temps de calcul de l'algorithme peut rendre le jeu presque inutilisable de par sa lenteur (près de 100 secondes pour le réseau de neurones, près de 40 pour la méthode des k plus proches voisins). Sa complexité est principalement due au temps de création de l'arbre. Celle-ci varie en fonction du temps d'évaluation d'un plateau, mais sa lenteur relative peut être expliquée par la non-optimalité de notre code : on est amenés à copier de très nombreuses fois le plateau de jeu, et les fonctions de notre moteur de jeu sont probablement peu optimisées.

### 3.2.3 Le réseau de neurones artificiel

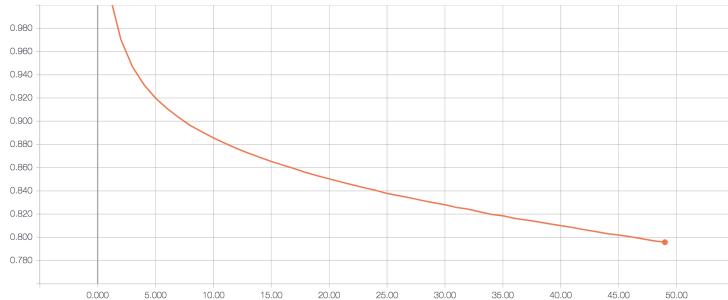
De nombreux paramètres sont à établir dans un réseau de neurones. Il s'agit du nombre de couches à utiliser, du nombre de neurones dans chaque couche, de la technique d'optimisation (qui permet de minimiser la fonction de coût), la fonction de coût, du nombre d'itérations (nombre de fois que l'on passe dans toute la base d'entraînement), de la taille du batch (échantillon que l'on prend à chaque tour dans le réseau de neurone pour recalculer les nouveaux poids et biais), de la learning rate (qui peut-être vue intuitivement comme la vitesse à laquelle on cherche un minimum de la fonction de coût).

Les paramètres structurels sont à intuiter en cherchant dans les revues scientifiques les problèmes qui se rapprochent le plus du nôtre. Pour certains, il y a même un consensus dans la communauté scientifique (souvent amené à changer). Par exemple pour notre type de problème, l'optimiseur d'Adam est a priori le meilleur donc c'est lui que nous utilisons. Il en est de même pour le choix de quatre couches dont trois de 2048 neurones (un nombre en puissance de deux est plus efficace lorsque l'on utilise un GPU). De même pour la fonction de coût qui est classique, l'entropie croisée. Pour les autres paramètres, appelés « hyperparamètres », leur valeur doit être choisie empiriquement. Nous n'avons pas eu le temps ni les ressources d'utiliser une méthode « grid search » permettant de croiser le plus d'essai afin d'optimiser les hyperparamètres. Nous avons cependant pu tester certains paramètres à la main, en voici quelques-uns.

Le premier paramètre est le nombre d'itérations sur la base de données d'entraînement. Voici un graphe montrant notre précision (nombre de classification correctes sur la base d'entraînement) en fonction du nombre d'itérations (au total, 50) :



On remarque qu'à chaque itération notre réseau est meilleur, ce qui est logique car à chaque fois le coût est minimisé comme le montre le graphique similaire suivant (coût en fonction du nombre d'itérations) :



Cependant, il ne faut pas non plus choisir un nombre d'itération trop grand. On peut en théorie choisir un nombre très grand qui permettrait d'avoir 100% de précision, mais ce serait sur la base d'entraînement. Et il est assez clair qu'un tel réseau risque d'être mauvais sur un jeu de donnée nouveau (comme celui de test), car il est devenu trop spécifique : ceci s'appelle l'overfitting. On cherche au contraire à avoir un réseau de neurones capable de généraliser à des données nouvelles. Nous avons conclu que 200 itérations étaient optimales.

Concernant la learning rate, on la fixe en général à  $10^{-3}$  pour l'optimiseur d'Adam. Nous avons tout de même souhaité comparer des learning rate voisines pour en être certain :

learning rate	accuracy on test dataset
$10^{-3}$	0.59
$10^{-4}$	0.62
$10^{-5}$	0.61
$10^{-6}$	0.50

On remarque qu'une learning rate de  $10^{-4}$  maximise la précision (avec les mêmes paramètres que précédemment)

Concernant la taille du batch, il faut savoir qu'avec un bon GPU, un batch de plusieurs milliers d'échantillons ne pose pas de problème, c'est même plus rapide qu'un batch plus petit. Sa valeur si elle est suffisante ne change pas significativement la précision. Nous avons remarqué qu'un batch de 2048 permet un entraînement plutôt rapide sur le GPU (30s par itérations) et qu'un batch plus petit ne changeait pas la précision. Nous avons donc choisi un batch de taille 2048.

Ainsi, avec les paramètres précédents, nous avons réussi à obtenir une précision de 68% (pourcentage de classifications correctes sur une base de test de 1 million de positions de plateau). Nous ne sommes pas satisfaits de ce taux. Nous sommes en particulier bloqués par la précision sur la base d'apprentissage qui atteint une limite à 70% (n'augmente plus significativement avec le nombre d'itérations). Nous avons essayé d'autres paramètres sans succès, et avons également essayé de changer le nombres de classes, et avons refait un nouveau dataset, sans succès. Il est probable que le problème vienne de la modélisation d'un plateau de jeu, qui n'est pas assez parlante pour le réseau de neurone. Nous en reparlerons dans la partie perspectives.

### 3.2.4 Les K plus proches voisins

Avec les K plus proches voisins, nous avons 80% de classifications correctes, ce qui est meilleur que le réseau de neurones actuel. Cependant, contrairement au réseau de neurones, ce pourcentage est difficilement améliorable et n'est pas encore satisfaisant.

### 3.2.5 La méthode de recherche arborescente de Monte-Carlo

La méthode de recherche arborescente de Monte-Carlo est une méthode élégante par sa théorie, mais extrêmement difficile à mettre en place de par les capacités de calcul requises. Elle a permis de faire grandement avancer les performances des algorithmes pour de nombreux jeux, et notamment le jeu du go réputé insoluble. Elle a été la structure utilisée pour battre Lee Sedol, le champion mondial de go en 2016, mais cela a nécessité un véritable tour de force : pas moins de 76 processeurs très performants travaillaient en parallèle, pour un temps de calcul de chaque coup d'environ 2 à 12 minutes.

Sur notre processeur utilisé, le temps de calcul pour une simple simulation, ie le déroulement aléatoire d'une partie jusqu'à arriver à un noeud terminal, peut prendre jusqu'à plusieurs dizaines de secondes. En effet, la profondeur d'un tel noeud terminal peut aisément dépasser 200, et l'algorithme peut même se perdre dans une partie nulle où toutes les pièces sont mangées et où seuls les deux rois bougent, rendant la simulation presque infinie. Effectuer des calculs statistiques sur de telles simulations devient alors impossibles car il faudrait plusieurs minutes simplement pour tester les premiers noeuds descendants de la racine.

Ainsi, pour essayer de revenir à des temps de calcul raisonnables, on va

plus simuler des parties nécessairement jusqu'à un état terminal, mais jusqu'à un stade avancé de maximum 30 coups. À ce stade, un résultat sera alors considéré comme positif si l'évaluation de cette configuration de plateau finale dépasse un certain seuil. Tous les différents types d'évaluation développés dans la première partie sont donc utilisables. Cela permet de réduire considérablement le temps de calcul d'une simulation, et d'optimiser le nombre de simulations possibles.

En contrôlant la durée d'exécution du programme, on peut déterminer le nombre de simulations possibles pour différentes durées :

— Pour 20 secondes :

```
Nombre de simulations en 20s:  
14  
Moyenne temps de simulation :  
1.4857134478432792
```

— Pour 100 secondes :

```
Nombre de simulations en 100s:  
70  
Moyenne temps de simulation :  
1.403047844341823
```

Un temps raisonnable d'attente pour une intelligence artificielle est d'environ 20 secondes. Or, le programme n'est capable d'effectuer que 14 simulations pendant cette durée : cela ne permet même pas de visiter les premiers descendants de la racine (ie les différents coups jouables par l'intelligence artificielle).

Pour 100 secondes, le nombre de simulations est encore trop faible pour en tirer de vraies conclusions statistiques.

### 3.3 Perspectives et conclusion

#### 3.3.1 Améliorations possibles pour la méthode de Monte Carlo

On peut alors déceler les contraintes matérielles de cette méthode. Pour que celle-ci soit techniquement possible, il faut nécessairement pouvoir utiliser des processeurs plus performants. De plus, l'exploration de l'arbre est adapté à la parallélisation. En effet, chaque simulation de parties est indépendante et il paraît tout à fait envisageable de pouvoir les effectuer de manière parallèle et de rassembler les résultats communs à la fin. La parallélisation de cette méthode est donc nécessaire pour pouvoir avoir des résultats corrects car dans le cas présent et pour des durées avoisinant la centaine de secondes, celle-ci semble encore jouer au hasard.

### 3.3.2 Améliorations possibles pour le réseau de neurones

Comme nous l'avons vu, il s'agit principalement d'un problème de conception d'un plateau de jeu. Tel que nous l'avons réalisé, il peut ne pas être assez expressif pour le réseau de neurones. Plutôt que de présenter une liste de taille 64, nous devrions donner une liste contenant pour chaque pièce son existence, sa valeur et ses coordonnées par exemple. Ceci permettrait au réseau de mieux gérer les positions des pièces sur le plateau (l'étape de convolution n'étant apparemment pas du tout suffisante ici). Il semble également que l'hypothèse de la non nécessité d'heuristiques humaines soit fausse : pour les ouvertures, il pourrait être utile de donner des indications au réseau.

L'idéal serait en réalité d'utiliser des techniques d'apprentissage par renforcement pour que le réseau puisse apprendre à jouer sans données. Cette idée a été notamment exploitée avec succès dans cette *publication*

### 3.3.3 Conclusion générale

Les performances de nos intelligences artificielles sont finalement assez décevantes. En effet, elles n'arrivent pas à jouer de manière réellement humaine et l'intérêt de ses coups est parfois difficile à percevoir. Le compromis trouvé pour classifier la base de données (nombre de classes différentes pour optimiser le taux de réussite) ne parvient pas à restituer la finesse de l'évaluation de StockFish. Ainsi l'intérêt de l'algorithme minimax devient restreint car trop de parties différentes sont évaluées de la même façon. De plus, le temps de calcul parfois excessif empêche de pouvoir envisager un match sérieux contre nos IA.

Même si nos résultats de classifications par le réseau de neurones sont décevants, nous avons appris énormément de choses. Nous avons pu mettre en application les techniques de gestion de projet informatique comme Git, les tests unitaires, la génération de documentation et le travail en équipe. Concernant l'approche technique, nous avons appris qu'il est essentiel d'essayer plusieurs techniques de classifications, et que celles que l'on pense le plus efficace ne sont pas toujours les plus récentes et complexes (même si cette approche est en réalité largement améliorable). Ce projet a été très demandeur en temps en raison de nombreux concepts nouveaux à apprendre, et ce que nous avons appris ici nous sera très utile pour nos expériences futures. Enfin, nous souhaitons remercier tout particulièrement notre tutrice Elisabeth Brunet pour son soutien, ses explications et sa bonne humeur.

## 4 Manuel utilisateur

### 4.1 Modules utilisés

Python est un langage interprété, il est donc nécessaire de posséder Python 3 avec certains modules afin de lancer le programme :

- numpy
- PyQt5
- anytree
- math
- Tensorflow
- ScikitLearn de Scipy

Tous ces modules, à l'exception de anytree (que nous fournissons dans le livrable) sont disponibles dans la distribution Anaconda. Notons que pour les tests unitaires nous avons utilisé le module Unitest, pour les exécuter il est donc aussi nécessaire d'avoir ce module.

Pour utiliser les scripts faisant appel aux méthodes des K plus proches voisins et du réseau de neurones, il faut respectivement posséder les jeux de données pour les K plus proches voisins et le modèle entraîné pour les réseaux de neurones. Ces fichiers sont appelés "newGamesData.npy", "newOneHotEncoded.npy" et "model". Ils sont disponibles sur notre dépôt GitHub, et sont à placer dans le dossier "src" situé au même endroit que les scripts. Nous avons été obligé de procéder ainsi à cause de leur taille conséquente (il s'agit en plus de fichiers où les données ont été largement réduites à cause des limitations de GitHub).

### 4.2 Commandes d'utilisation

Pour jouer contre un autre joueur, ou contre nos intelligences artificielles, il faut lancer les commandes suivantes :

#### 1. `gui.py`

Il s'agit de l'interface graphique sans intelligence artificielle. C'est le script qu'il faut lancer si l'on veut joueur à deux sans intelligence artificielle, via la commande (UNIX) :

```
$ python3 gui.py
```

## **2. gui\_IA\_denombrement.py**

Il s'agit de l'interface graphique avec intelligence artificielle utilisant l'évaluation par dénombrement. C'est le script qu'il faut lancer si l'on veut joueur seul contre l'intelligence artificielle, via la commande (UNIX) :

```
$ python3 gui_IA_denombrement.py
```

## **3. gui\_IA\_nn.py**

Il s'agit de l'interface graphique avec intelligence artificielle utilisant l'évaluation par le réseau de neurone. C'est le script qu'il faut lancer si l'on veut joueur seul contre l'intelligence artificielle, via la commande (UNIX) :

```
$ python3 gui_IA_nn.py
```

## **4. gui\_IA\_knn.py**

Il s'agit de l'interface graphique avec intelligence artificielle utilisant l'évaluation par la méthode des k plus proches voisins. C'est le script qu'il faut lancer si l'on veut joueur seul contre l'intelligence artificielle, via la commande (UNIX) :

```
$ python3 gui_IA_knn.py
```

## **5. gui\_IA\_mcts.py**

Il s'agit de l'interface graphique avec intelligence artificielle utilisant la méthode de recherche arborescente de Monte-Carlo. C'est le script qu'il faut lancer si l'on veut joueur seul contre l'intelligence artificielle, via la commande (UNIX) :

```
$ python3 gui_IA_mcts.py
```

Malheureusement, la version finale déposée ne fonctionne plus au bout de deux coups pour des raisons encore inconnues, mais qui seront probablement réglées lors de la soutenance.

### **4.3 Documentation**

Le code des différents modules est commenté progressivement, avec en plus une docstring et une documentation HTML générée par l'outil Sphinx. Cette documentation des classes et fonctions utilisées se trouvent dans le dossier Documentation. Il suffit d'ouvrir dans un navigateur le fichier "index.html".