GLOESENER Maxime – SYED Tyemur

Advanced Machine Learning
Q-Learning pour les finales au jeu d'échecs

Table des matières

Introduction

Dans ce rapport, nous allons expliquer les différentes approches que nous avons pu tester pour implémenter un modèle Q-Learning qui permet de jouer des finales au jeu d'échecs.

Dans un premier temps, nous avons pu tester les deux approches qui étaient proposées dans les consignes, à savoir :

- explorer les options possibles à partir d'un papier de recherche sur le Q-Learning et le code fourni sur un Github ;
- essayer d'implémenter un projet similaire en utilisant Gym-Chess.

Ces deux méthodes n'ont pas donné de résultats concluants et nous expliquerons pourquoi dans la suite du rapport.

Ensuite, comme les deux premières méthodes n'avaient donné aucun résultat, nous avons retrouvé la personne au nom de Antonio Boar (Tonisnakes) qui a fait le papier de recherche pour son projet « Bachelor's Project Thesis – Solving Chess Endgames using Q Learning ». Nous avons retrouvé son Linkedin et nous avons essayé de prendre contact avec lui par différents moyens, en espérant pouvoir obtenir le code correspondant à sa thèse, mais nous n'avons jamais eu de réponse.

Pour implémenter un algorithme de Q-learning, une troisième méthode a été testée en utilisant la librairie Python-Chess. Pour cette approche, nous sommes partis de zéro et nous avons réussi à obtenir des résultats intéressants.

Finalement, étant donné que le Q-learning est limité à des cas où le nombre d'états n'est pas trop grand, nous nous sommes intéressés au Deep Q Learning et nous avons pu montrer l'intérêt d'utiliser des réseaux de neurones pour approximer les Q values.

Les différentes méthodes seront expliquées dans les prochains chapitres, ce qui permettra de mettre en avant les problèmes rencontrés et les solutions trouvées pour répondre à ces problèmes.

Q-learning

Avant de voir en détails les approches utilisées, voyons à quoi correspond la méthode Q-learning.

C'est un algorithme d'apprentissage par renforcement qui permet à un agent de décider quelles actions prendre dans un environnement en fonction des récompenses qu'il reçoit. Cet algorithme vise à maximiser le profit au long terme. L'algorithme fonctionne en itérant sur beaucoup d'épisodes en permettant à l'agent d'interagir avec l'environnement.

A chaque étape d'un épisode, l'agent choisit une action en fonction de la table de Q values et d'un paramètre epsilon. Ce dernier permet à l'agent de choisir soit faire une action au hasard (phase d'exploration) ou faire la meilleure action à cet instant en fonction de la table de Q values actuelle. Il effectue cette action et reçoit une récompense (qui peut être négative).

L'algorithme itère de cette manière jusqu'à que nous arrivons à un état terminal et un prochain épisode est lancé. La table de Q values est mise à jour en utilisant la formule de Bellman qui relie la valeur Q d'un état à la valeur Q des états suivants et les récompenses reçues.

La mise à jour des Q values suit la formule suivante :

$Q(s,a) \leftarrow Q(s,a) + \alpha * (reward + \gamma * \max\big(Q(nextstate, nextaction)\big) - Q(s,a))$, avec s qui correspond à l'état actuel et a qui est l'action effectuée.

Lors de son entrainement, la table de Q values devient de plus en plus précise et l'agent pourra prendre des décisions de plus en plus efficaces pour maximiser la récompense à long terme.

Approche 1 : Explorer les possibilités à partir d'un projet déjà fait

Pour la première approche, nous nous sommes basés sur ce GitHub : https://github.com/paintception/A-Reinforcement-Learning-Approach-for-Solving-Chess-Endgames. Ce GitHub est celui qui était proposé dans les consignes et il permet de travailler avec des finales de jeu d'échecs à 3 pièces.

Nous avons passé pas mal de temps à comprendre le fonctionnement de leur projet, nous avons réussi à le lancer et nous avons pu observer le déroulement des parties. En étudiant le projet, nous nous sommes rapidement rendu compte que celui-ci était assez grand et composé de plusieurs parties. En effet, c'est un projet qui a été réalisé par 3 personnes durant une période de 1 mois si on se base sur les commit sur leur GitHub. Leur projet contient de nombreuses perspectives d'amélioration dans le code qui n'ont jamais été terminées. Au final, comme les perspectives n'ont jamais été implémentées, leur code ne fonctionne que pour des parties où nous avons une tour blanche, un roi blanc et un roi noir. Le roi noir étant considéré comme un joueur expérimenté d'échecs selon ce qui est expliqué dans les explications du projet.

Le code suivant dans **BaseParams.py** permet d'initialiser l'échiquier avec tous les états possibles ayant les trois pièces mentionnées :

```
def get_all_params(self):
    params = []
    for wk_r in range(0,8):
        for wk_c in range(0,8):
            for wr_r in range(-1,8):
                for wr_c in range(-1,8):
                    for bk_r in range(0,8):
                        for bk_c in range(0,8):
                                    for white_plays in range(0,2):
                                        if(wr_r == -1 and wr_c != -1 or wr_r!=-1 and wr_c == -1) :
                                            continue
                                        params.append((wk_r, wk_c, wr_r, wr_c, bk_r, bk_c,white_plays))
    return params
```

Figure 1 – Initialisation des états

Cela génère environ 250000 possibilités. Ces états sont vérifiés et stockés dans un fichier .bson. Après cela il faut lancer le code **QLearning.py** qui permettra d'entrainer le modèle et d'obtenir une Q Table qui sera utilisée par le fichier **Play.py.**

Dans ce dernier fichier, on fait jouer une configuration donnée et on affiche si le modèle a réussi à gagner et en combien de coups.

Nous avons observé les résultats de plusieurs parties et après plusieurs essais, le modèle n'a jamais réussi à gagner une partie.

Nous avons donc décidé d'observer les jeux. En observant ces parties jouées lors de l'entrainement, on peut vite se rendre compte que le joueur qui joue les noirs fait des coups au hasard. L'agent contre qui s'entraîne le modèle n'est donc pas un joueur d'échecs professionnel comme dit dans le « readme » du GitHub qui insinue que le joueur noir est un joueur expérimenté. Ce joueur n'est donc pas du tout fort. Cela explique les mauvaises performances du modèle. C'est assez intuitif de comprendre pourquoi s'entrainer contre de l'aléatoire ne peut pas donner de bons résultats mais nous expliquerons plus en détails lors l'approche 3 pourquoi cela est important de jouer contre un bon

joueur et pourquoi de ce fait, le Q learning n'est pas utilisé en pratique pour des environnements avec plusieurs agents qui jouent l'un contre l'autre.

Il y a également des traces d'implémentation d'autres pièces comme le fou. Nous savons qu'il est impossible de mettre en échec et mat un roi en utilisant seulement un fou et un roi. Cette modélisation nécessite donc au moins 4 pièces en jeu pour le joueur blanc. Pour gérer ces cas, il faut ajouter une pièce en plus. Il faut donc changer la grande boucle for mentionné sur la figure 1 pour pouvoir faire cela. On peut envisager d'étendre le projet à des cas plus complexes, notamment des cas où nous avons plus de 3 pièces en jeu il faut :

- ajouter le comportement des nouvelles pièces qu'on veut gérer ;
- ajouter la génération de tous les nouveaux états possibles.

Ce sont les conditions nécessaires auxquelles nous avons pensé directement. Nous avons donc implémenté la classe **Pawn** dans **Pieces.py**. Un pion peut devenir une reine. Donc nous avons implémenté **Queen** aussi. Ces deux classes ont été implémentées en se basant sur les classes des autres pièces. Le code se trouve dans le fichier **Pieces.py**.

Nous ne pouvions pas vérifier le fonctionnement des pièces avant d'avoir lancé la génération des états possibles. C'est là que nous avons compris que la modélisation utilisée par les créateurs du projet ne fonctionnera pas pour nous. En ajoutant une pièce en plus (un pion noir) dans le jeu, la génération des états demande beaucoup plus de RAM. On passe de 100MB à plus de 10GB. Ce fichier est beaucoup trop grand lors de la génération pour pouvoir le générer et sauvegarder. De plus, il faudra faire la Q Table pour tous ces états après. Cette approche a donc été abandonnée.

Approche 2 : Gym-Chess

La deuxième approche proposée était d'utiliser Gym-Chess. Après quelques recherches rapides, nous nous sommes rendu compte que Gym-Chess est inutilisable dans l'état actuel des choses. La documentation pour cet API est très limitée. Nous avons seulement le site https://pypi.org/project/gym-chess/ qui contient quelques informations sur la librairie mais la documentation est assez mal expliquée. Après avoir utilisé les méthodes et les fonctions proposées, nous avons vite compris que cette API était très limitée. Nous ne pouvions même pas définir des environnements de jeu d'échecs nous-mêmes.

En bref, après avoir testé cette API, nous nous sommes rapidement rendu compte qu'elle ne pouvait pas répondre à nos besoins pour le projet. Les environnements sont prédéfinis. Il est impossible de définir nos propres environnements pour travailler avec seulement un certain nombre de pièces. Nous avons commencé à chercher une alternative et nous l'avons trouvé : Python-Chess.

Approche 3 : Python Chess

Pourquoi Python-Chess ?

La librairie Python-Chess permet de faire pratiquement tout ce qu'on pourrait imaginer faire avec un environnement du jeu d'échecs. La documentation est également très complète. Nous avons accès à toutes les fonctions nécessaires pour réaliser ce projet. Nous pouvons générer facilement et rapidement tous les états possibles et pour chaque état, grâce à python chess, nous pouvons connaître les coups légaux, ce qui nous permet donc de définir les actions associées à chaque état.

Par exemple, pour la définition d'un échiquier, nous pouvons utiliser le code **FEN** (Forsyth–Edwards Notation) qui nous permet de générer un échiquier ayant un état particulier.

Pour le code 'R2K4/8/8/8/8/8/8/3k4', il nous suffit de définir chess.Board('R2K4/8/8/8/8/8/8/3k4') et cela nous permet d'avoir le board suivant:

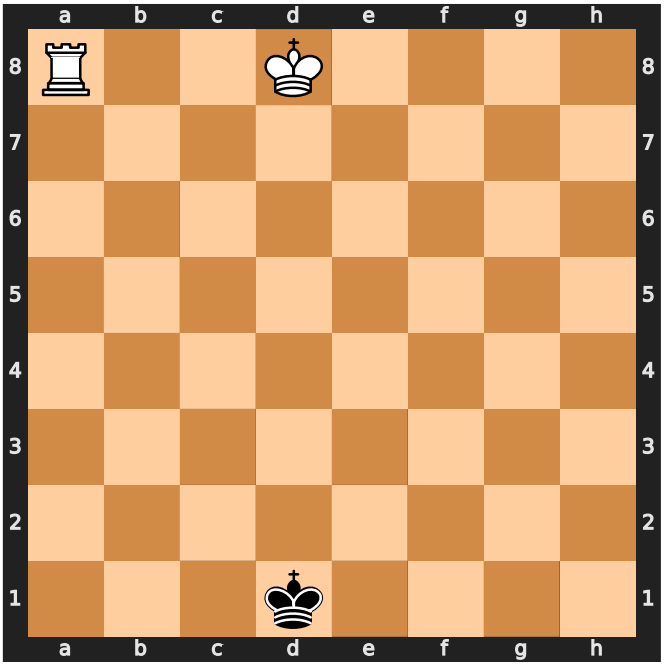

Figure 2 - Board 'R2K4/8/8/8/8/8/8/3k4'

En plus de code FEN, il est aussi possible de générer un état à partir d'un board vide et d'y ajouter les pièces qui nous intéressent, pour générer cela :

1. Nous pouvons définir le code `fen='8/8/8/8/8/8/8/8'` qui correspond à un échiquier vide. Nous utilisons ensuite la méthode `b = chess.Board(fen)` pour définir un échiquier b sans pièce.
2. Pour remplir l'échiquier, il suffit après d'utiliser une autre méthode appelée set_piece_at() qui permet d'ajouter des pièces sur un échiquier. Par exemple, l'ajout d'un roi noir à la case 45 se fait avec :
   `b.set_piece_at(chess.Square(45),chess.Piece(chess.KING,chess.BLACK))`

C'est cette deuxième méthode que nous avons pu utiliser pour générer tous les états possibles de l'échiquier avec un roi contre un roi + une tour, un roi contre deux tours et un roi contre une dame et un roi.

Le fichier **pychess.ipynb** correspond au fichier qui définit en premier temps tous les états possibles. Ensuite, nous définissons une Q Table et nous l'entrainons grâce à stockfish (un engine de chess).

Définition des états

Les états possibles que nous définissons dans la Q Table correspondent uniquement à tous les états où c'est aux blancs de jouer. Nous ne devons donc pas définir les états quand les noirs jouent car ce sont des états intermédiaires pendant lesquels nous ne pouvons rien faire. C'est stockfish qui joue pendant ces états. Nous arrivons ensuite dans un état où c'est de nouveau aux blancs de jouer.

Pour chaque état, il faut définir les actions possibles. Ce sont les coups possibles pour les blancs. Chaque coup a une récompense qui sera mise à 0 au début, avant l'entrainement. Pour générer les actions pour chaque état, nous nous basons sur la fonction legal_moves() de python-chess qui nous permet, pour un état, de connaître tous les coups possibles légaux.

Pour générer les états possibles, nous utilisons la méthode expliquée juste avant et pour chaque état généré, nous le vérifions avec les fonctions board.is_valid() et board.is_insufficient_material() pour s'assurer que l'état est valide et qu'il reste assez de matériel pour continuer le jeu.

La génération des états possibles licites pour le cas un roi contre un roi et une tour, nous avons déjà 178780 états possibles. Nous prenons en compte seulement les cas où le joueur qui joue les blancs peut jouer donc par exemple, nous ne prenons pas en compte les cas où la partie est terminée (échec et mat, stalemate…). Cette valeur est logique car si on prend 64*63*62 on a 249984 et le nombre d'états obtenus dans notre cas, est de façon logique, une valeur amortie de cela car on prend en compte que les cas possibles.

Pour la situation avec un roi contre un roi et deux tours, nous avons 9029468 états possibles.

Joueur noir – pourquoi Stockfish ?

Q-learning est un algorithme conçu pour des environnement à agent unique. Pour adapter cet algorithme à des environnements multi-agents, comme dans notre cas, il faut considérer les autres agents comme faisant partie de l'environnement et les modéliser. Dans beaucoup de projets utilisant le Q learning, l'agent est modélisé de façon aléatoire. Cela pose un problème car le modèle va s'entrainer contre un agent qui joue de façon aléatoire et sera donc très mauvais contre des bons joueurs. Dans le cadre de notre projet, nous avons pu utiliser Stockfish pour modéliser le comportement d'un bon joueur (environ 3600 elo avec la version utilisée).

Stockfish est un engine de chess gratuit et disponible sur beaucoup de plateformes. Nous l'utilisons pour faire les coups du joueur qui joue les noirs. Nous lui donnons un temps défini pour calculer son coup. Il fait ce coup et c'est à nous de jouer de nouveau.

Entrainement

Pour que le modèle soit performant, il faudrait normalement entrainer sur chaque état plusieurs fois. Nous avons choisi à chaque itération de prendre un état au hasard parmi les états possibles et jouer tant que nous n'arrivons pas à un état où le jeu ne peut plus continuer. Le jeu ne peut plus continuer s'il n'y a pas assez de pièces pour mettre en échec et mat ou alors on a mis en échec et mat. Les conditions de match nul après répétition de mêmes coups ou la règle de 50 coups sont également prises en compte.

Pour un épisode, tant que la partie n'est pas finie, on boucle et on fait :

- Choix d'un état
- Récupération de l'action faite, soit on prend le meilleur coup possible en fonction de la table de q values, soit on fait un coup aléatoire pour explorer
- Récupération du reward pour l'action qui a été choisie
- On fait jouer stockfish pour avoir le prochain état
- Mise à jour de la q value en suivant la formule de Bellman

Nous faisons beaucoup d'épisodes, ce qui nous permet de compléter la table de Q values.

Il faut noter qu'il est important de définir intelligemment les différents rewards. Par exemple, au début du projet, nous avions donné un reward lorsque l'action réalisée permettait de mettre en échec le roi noir. Le problème de cela est que du coup, lorsqu'on jouait contre le bot, il avait tendance à suicider sa tour pour mettre en échec mais ne prenait pas en compte le fait qu'on pouvait simplement lui prendre sa tour avec le roi noir. Nous avons donc dû adapter les rewards en fonction des observations que nous avons pu faire.

A nous (vous) de jouer

Après avoir entraîné le modèle, nous avons stocké les états possibles et les valeurs des Q tables dans des fichiers. Nous avons implémenté 3 cas possibles mais il est extrêmement simple de générer des finales supplémentaires, il suffit lors de l'initialisation des états, d'initialiser les états voulus supplémentaires.

Les cas possibles sont :

- Roi contre roi + tour
- Roi contre roi + dame
- Roi contre roi + 2 tours

Nous avons créé un fichier **playchess.ipynb** qui peut être utilisé pour jouer contre les bot entrainés. Le bot va toujours jouer les blancs. Nous n'avons le droit qu'à un roi dans ce cas.

A chaque coup, nous affichons l'état du jeu et les différents coups possibles. On peut introduire les coups à jouer grâce la console. Après chaque coup introduit, le bot va faire son coup et vous pouvez de nouveau faire un coup.

Lors de chaque coup, l'échiquier sera affiché comme ceci :

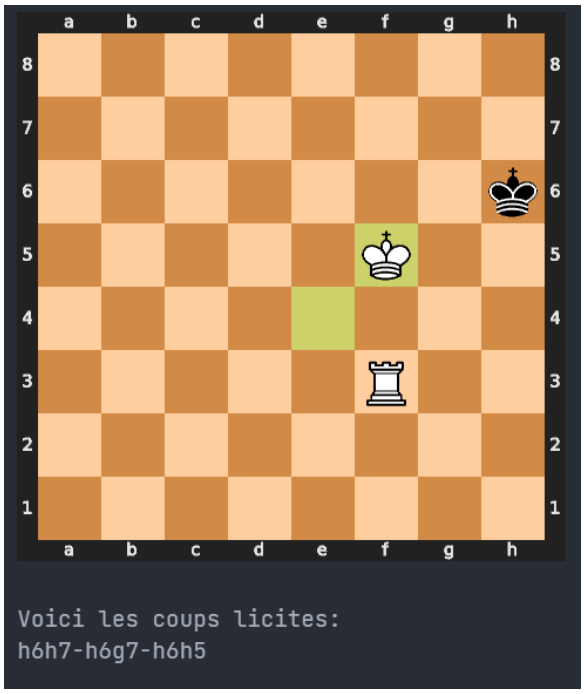

Figure 3 Interface

Les coups permis sont aussi affichés juste en-dessous de l'échiquier. Il suffit de rentrer un des coups permis dans la console qui s'ouvre, le bot jouera alors le coup pour les blancs et vous pourrez jouer la prochaine position pour les noirs.

Pour jouer ce type de partie, soit vous pouvez définir un état à partir d'un code FEN, soit vous pouvez choisir un état aléatoire parmi tous les états possibles. Nous avons essayé de commenter un maximum notre code pour faciliter la compréhension.

Cas avec 4 pièces en jeu (2 tours blanches, 1 roi blanc et 1 roi noir)

Nous avons également implémenté un fichier **4pychess.ipynb** où nous avons géré le cas avec 4 pièces (un roi contre un roi et deux tours). Nous devons ajouter des boucles en plus lors d'initialisation pour ajouter tous les états possibles en plus. Le nombre d'états licites dans ce cas est environ 10^7.

L'initialisation des états possibles prend une dizaine de minutes. Il ne faut pas changer le reste du code. Tout est pris en compte automatiquement. Il faut seulement changer le nom des fichiers à utiliser pour save/load les Q Tables et les boards.

Conclusion de l'approche

L'utilisation de Python-Chess est une amélioration énorme par rapport l'approche 1 où nous avons exploré un projet déjà fait.

Premièrement, le code run beaucoup plus vite. Nous arrivons même à générer tous les états possibles avec 4 pièces (2 tours blanches, un roi blanc et un roi noir sur le board) en moins de 10 minutes. La création de la table Q prend également beaucoup moins de temps. On passe d'une dizaine de minutes à quelques secondes pour le cas avec 3 pièces. Nous arrivons à la générer pour 4 pièces également.

Deuxièmement, nous pouvons définir très facilement, juste en modifiant le nombre d'épisodes, la qualité de notre « bot ». Si on le fait jouer pendant des jours et des jours, on est certain qu'il peut arriver à des coups optimaux après un certain temps. Le temps d'entrainement n'est fonction que du

nombre d'épisodes et de la « force » de Stockfish. Il est discutable que Stockfish puisse trouver des coups meilleurs si on lui donne plus de temps pour calculer les coups. Dans notre cas, nous donnons à Stockfish un temps de 0.1s pour calculer les coups. En lui donnant plus de temps, la qualité des coups de Stockfish pourrait être améliorée et donc, cela pourrait améliorer notre agent mais cela prendrait beaucoup plus de temps.

Enfin, nous pouvons jouer nous-même contre le bot entrainé et le tester. On pourrait envisager d'implémenter une interface qui pourrait permettre d'interagir avec le board et choisir les coups directement dessus à la place de devoir les écrire.

Cependant, nous pouvons quand même se rendre compte des limites de cette approche. En passant de 4 pièces à 5, le nombre d'états possibles monte de 15 millions à presque 1 milliard. Il est impossible de générer cela avec un PC habituel. Il faut dans ce cas envisager d'autres approches, par exemple Deep Q-learning.

De plus, pour que le bot puisse avoir des bonnes performances, il faudrait pouvoir l'entrainer très longtemps en explorant plusieurs fois chaque état. Le problème est que pour 1 000 000 d'épisodes, cela prend environ 14 heures. Avec 1 000 000 d'épisodes, nous explorons 1 000 000 d'états, or avec 3 pièces on a déjà 178780 états initiaux et avec 4 pièces, on a 9029468 états initiaux. On peut remarquer que rien que pour explorer au moins une fois chaque état initial pour 4 pièces, il faut déjà 126 heures. Pour avoir un bot performant, il ne suffit pas d'explorer une fois un état mais plusieurs fois donc cela prend vite énormément de temps pour entrainer un modèle.

Pour nos différents bots, ils ont été entraînés durant une dizaine d'heures chacun. Les Q tables et les états initiaux générés ont été stocké pour éviter de devoir relancer l'entrainement. De plus, ce qui peut aussi être fait, est de repartir des valeurs de la Q table et de relancer un entrainement.

Un autre point négatif pour Q-learning est qu'il faut avoir des agents adversaires pour les environnements multi-agents. Il faut donc les considérer comme faisant partie de l'environnement et les implémenter. Il faut également que ces agents soient très forts. Nous avons eu la chance de retrouver Stockfish qui est gratuit et disponible. Mais dans un cas différent, nous devrions peut-être modéliser tous les autres agents que nous n'avons pas.

Deep Q-learning

Qu'est-ce que c'est ?

C'est une variante de l'algorithme Q-learning. Le Deep Q-learning utilise un réseau de neurones profonds pour approximer la fonction Q à la place de la stocker dans une table Q. On peut donc arriver à gérer des environnements avec un nombre beaucoup plus grand d'états et d'actions.

Dans Deep Q-learning, on entraine le réseau qui prend en entrée l'état courant et donne en sortie l'estimation de la valeur Q pour chaque action possible. L'agent choisit l'action qui maxime la valeur Q. La fonction Q est mise à jour en utilisant la même formule de Bellman qu'expliquée au début du rapport mais en utilisant les sorties du réseau de neurones.

Utilisation de DQL pour le jeu des échecs

Après avoir implémenté le Q learning pour les finales du jeu d'échecs, nous avons pu nous rendre compte des limites de cette méthode lorsque la taille du problème augmente. Rien que la génération des états et des actions peut prendre des heures (avec plus de 4 pièces) et la taille de la Q table explose et ne peut plus être stockée en mémoire.

Nous nous sommes donc renseignés sur l'utilisation du Deep Q-Learning pour les échecs et nous avons trouvé des cas d'application intéressants. Pour la partie Q-Learning, nous avons tout implémenté from scratch, dans cette partie, nous nous basons sur un travail déjà fait que nous avons simplement adapté à nos besoins, l'objectif est surtout de bien comprendre comment le Deep Q-Learning fonctionne.

Après quelques recherches, nous avons trouvé un cas concret d'application de Deep Q-Learning sur des parties d'échecs. Le fichier permettant l'implémentation du Deep Q-Learning pour le jeu d'échecs est : **Deep_Reinforcement_Learning_Chess.ipynb.** Ce fichier se base le travail d'une personne qui a implémenté l'approche Deep Q-learning pour le jeu d'échecs complet et qui a été adapté pour notre cas. Les parties se jouent du tout début jusqu'à la fin. Dans le notebook, nous avons la possibilité de jouer contre le modèle nous-mêmes, visualiser le modèle jouer contre un adversaire qui fait des coups aléatoirement ou alors de visualiser le modèle jouer contre stockfish.

La librairie utilisée pour modéliser le jeu des échecs est également Python-Chess. L'auteur utilise la version 1.2.0. Il utilise également Keras pour le réseau de neurones.

Ce qui est important de voir est que le modèle n'apprend pas dès le premier coup mais uniquement après le 4ème coup. En regardant le système des récompenses, on peut comprendre la raison pour cela. On attribue une récompense uniquement quand le modèle gagne la partie. Le minimum de coups à faire pour gagner une partie d'échecs est 4 coups (2 pour blancs, et 2 pour noirs). On n'apprend donc qu'à partir du 4ème coup.

On met à jour les poids du modèle tous les 100 coups. Nous pensons que cela est à cause du coût de la mise à jour des poids du modèle.

Le modèle utilisé pour le DQL est le suivant :

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 8, 8, 12)]	0
conv2d (Conv2D)	(None, 4, 4, 64)	3136
conv2d_1 (Conv2D)	(None, 2, 2, 128)	32896
conv2d_2 (Conv2D)	(None, 1, 1, 256)	131328
flatten (Flatten)	(None, 256)	0
dense (Dense)	(None, 4096)	1052672

Figure 4 - Architecture du modèle DQL

L'architecture du modèle est similaire à celle présentée dans le papier de recherche de DeepMind en 2013 lorsqu'ils ont implémenté un algorithme de Deep Q-Learning capable de jouer au jeu Atari.

Le choix de la prochaine action est faite grâce à ce réseau. Il y a quelques méthodes dans la classe du réseau qui permettent notamment de faire le choix d'action. **'predict'** utilise les poids actuels du modèle pour prédire l'action et **'explore'** choisi une action aléatoirement parmi les actions légales.

Pour **'predict'** le modèle prend en entrée l'échiquier représenté par un vecteur 8x8x12. La sortie du modèle est l'approximation de la valeur Q pour chaque coup. Il y a 4096 coups car on modélise absolument tous les coups. Nous avons 8*8 cases et on imagine pouvoir aller à n'importe quelle case.

Donc, (8*8)^2 = 4096. Lorsqu'on choisit l'action avec la plus grande valeur Q, on choisit uniquement parmi les coups licites.

Ce modèle est entrainé et après utilisé pour jouer contre nous (un vrai joueur), un joueur qui joue des coups aléatoirement ou alors contre Stockfish.

Conclusion de Deep Q-learning

Après avoir joué contre le bot entrainé, on remarque ces coups ne sont pas aléatoires. Il nous met en échec quelques fois. Cependant, si on joue sérieusement (1200 elo sur chess.com), il est assez facile de battre le bot. On ne sait pas estimer nous-mêmes l'elo du bot mais on peut voir que Stockfish bat le bot à chaque partie et souvent très vite. Il faut garder en esprit que le bot n'a été entrainé que pendant 100 époques.

Le gros avantage du Deep Q-Learning est le fait que le bot est maintenant capable de jouer des parties entières et pas seulement des finales avec peu de pièces.

Cependant, nous avons remarqué qu'il y avait de nouveau un problème de mémoire et de temps d'entrainement. Le modèle peut occuper une très grande espace mémoire car on stocke l'historique des récompenses, des actions, des états actuels et des états futures.

Il faudrait également entrainer le modèle pendant beaucoup plus d'époques pour arriver à des bons résultats.

Conclusion

En conclusion, lors de ce projet, nous avons pu explorer différentes pistes pour implémenter un algorithme de Q Learning pour le jeu d'échecs. Nous avons pu identifier les limitations du Q-Learning lorsque le nombre d'états augmente (plus de 4 pièces pour les échecs) et la nécessité d'utiliser un réseau de neurones pour approximer les Q values.

Nous avons aussi pu nous rendre compte de la complexité d'entrainer des modèles pour résoudre ce genre de problème. Le temps nécessaire pour entrainer un modèle devient vite très grand, même pour des problèmes de petites tailles (3 ou 4 pièces aux échecs) tant le nombre de possibilités d'états est grand.

Pour aller plus loin, il serait intéressant de laisser tourner plus longtemps l'entrainement des modèles pour que les bots deviennent meilleurs.