Rapport Advanced Machine Learning

Nicolas Melaerts
253882

December 2024

1 Sujet - RL1 - Q-learning pour les finales au jeu d'échecs - Partie pratique

L'objectif de ce projet est d'utiliser l'algorithme de Q-learning vu au cours pour résoudre des positions simples en finales de jeu d'échecs (roi + dame contre roi, ...).
Il s'agit d'explorer les possibilités et les limites du Q-learning pour des finales simples et de proposer des solutions pour des finales un peu plus compliquées (par exemple pour des finales roi + dame contre roi + pion), en vous basant sur les travaux [`https://fse.studenttheses.ub.rug.nl/28405/` et `github.com/paintception/A-Reinforcement-Learning-Approach-for-Solving-Chess-Endgames`].
Une extension au deep Q-learning est également possible, ainsi qu'une exploration de l'environnement $gym{-}chess$ de OpenAI `https://pypi.org/project/gym-chess/`

2 Introduction

Le Q-learning est une méthode de résolution de problèmes par renforcement, offrant une approche relativement simple mais puissante pour apprendre à partir d'interactions avec un environnement. Cependant, cette méthode présente des limites importantes, en particulier lorsqu'elle est appliquée à des environnements de grande échelle ou à des problèmes complexes comme les finales aux échecs.
Ce projet explore d'abord ces limites à travers l'application du Q-learning classique à des positions simples de jeu d'échecs.
Mais je vais constater que je suis vite limité, donc j'introduis l'approximate Q-learning, une extension basée sur l'utilisation de fonctions d'approximations. Cette approche permet de généraliser l'apprentissage à des environnements plus complexes.
Enfin, j'évalue cette méthode sur des plateaux plus évolués et analyse les résultats obtenus. Ce travail vise non seulement à comprendre les forces et les faiblesses des méthodes de Q-learning mais également à identifier des pistes pour améliorer les algorithmes pour des plateaux relativement simples aux échecs.

3 Définition du Q-learning et présentation de mon implémentation

Le Q-learning est un algorithme d'apprentissage par renforcement permettant à un agent d'apprendre une politique optimale en maximisant une fonction de récompense cumulative. L'agent interagit avec son environnement, ici, le plateau d'échecs, en explorant différentes actions à partir d'états donnés et en mettant à jour une table de Q-valeurs.

3.1 Structure et mise à jour des Q-valeurs

L'agent utilise une table Q notée $Q(s, a)$, où s représente un état (la configuration du plateau d'échecs) et a une action (un coup possible suivant cette configuration de plateau). La mise à jour des valeurs Q suit la règle suivante :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right) \tag{1}$$

où :

- s_t est l'état actuel (configuration actuelle du jeu d'échecs),
- a_t est l'action choisie (coup joué),
- r_{t+1} est la récompense obtenue après l'action a_t,
- s_{t+1} est le nouvel état résultant de a_t,
- γ est le facteur d'actualisation des récompenses futures,
- α est le taux d'apprentissage, qui contrôle la vitesse de convergence.

3.2 Stratégie d'exploration et d'exploitation

Pour choisir une action, l'agent applique une stratégie ϵ-greedy :

$$\text{Action} = \begin{cases} \text{Exploration : action aléatoire} & \text{avec une probabilité } \epsilon, \\ \text{Exploitation : action optimale} & \text{avec une probabilité } 1 - \epsilon. \end{cases}$$

Cette stratégie permet à l'agent d'explorer de nouvelles actions tout en exploitant les informations accumulées dans la table Q.

3.3 Implémentation du Q-learning dans notre projet

Notre agent est implémenté dans la classe `QLearningAgent`. Les principales méthodes sont les suivantes :

1. `get_q_value` Cette méthode retourne la valeur $Q(s, a)$ pour un état et une action donnés. Si la valeur n'existe pas dans la table Q, elle est initialisée à zéro :

```
def get_q_value(self, state, action):
    return self.Q.get(state, {}).get(action, 0)
```

2. `choose_action` L'agent choisit une action en appliquant la stratégie ϵ-greedy. Si l'état n'existe pas encore dans la table Q, toutes les actions possibles sont initialisées à zéro.

```
def choose_action(self, state, possible_actions):
    if np.random.rand() < self.epsilon:
        return random.choice(possible_actions)  # Exploration
    else:
        if state not in self.Q:
            self.Q[state] = {action: 0 for action in
                possible_actions}
        return max(possible_actions, key=lambda action: self.Q[
            state].get(action, 0))
```

3. `update_q` Cette méthode met à jour la table Q selon la règle de mise à jour du Q-learning :

```
def update_q(self, state, action, reward, next_state,
    possible_actions):
    best_next_action = max(possible_actions, key=lambda action:
        self.Q.get(next_state, {}).get(action, 0))
    current_q = self.Q.get(state, {}).get(action, 0)
    future_q = reward + self.gamma * self.Q.get(next_state, {}).get
        (best_next_action, 0)
    self.Q.setdefault(state, {})[action] = current_q + self.alpha *
         (future_q - current_q)
```

4. `train` L'agent s'entraîne pendant un nombre défini d'épisodes. Lors de chaque épisode :

- L'agent joue des coups légaux à partir d'une position initiale.
- Une pénalité est attribuée en cas de répétition de position ou si la règle des 50 coups est atteinte.
- Une récompense positive est donnée si l'agent atteint une position gagnante.
- La table Q est mise à jour après chaque action.

L'exploration diminue progressivement au cours des épisodes grâce à une petite diminution de ϵ :

```
self.epsilon = max(0.01, self.epsilon * 0.995)
```

4 Définition d'une positions simples en finale de jeu d'échecs

4.1 Représentation des états avec la notation FEN

La notation FEN (Forsyth-Edwards Notation) sert a représenter l'état d'une partie d'échecs. Cette notation FEN est une manière concise et standardisée de décrire une position d'échecs, utilisée dans de nombreux programmes et bibliothèques d'échecs, comme `python-chess`. Elle est composée de six champs, séparés par des espaces, qui décrivent entièrement l'état du plateau de jeu. Voici les détails :

1. **Disposition des pièces :** Le premier champ décrit la position des pièces rangée par rangée.
   - Les lettres majuscules (`K, Q, R, B, N, P`) représentent les pièces blanches : Roi, Reine, Tour, Fou, Cavalier, Pion.
   - Les lettres minuscules (`k, q, r, b, n, p`) représentent les pièces noires.
   - Les chiffres indiquent le nombre de cases vides consécutives.
   - Les rangées sont séparées par des barres obliques (`/`).
2. **Trait :** Le deuxième champ indique à qui c'est de jouer : `w` (blancs) ou `b` (noirs).
3. **Disponibilité du roque :** Mouvement unique dans un partie d'échecs qui combine le déplacement du roi et d'une tour mais je n'utilise pas cette fonctionnalité
4. **Case de prise en passant :** Le quatrième champ indique une éventuelle case de prise en passant, ou `-` s'il n'y en a pas. Comme le roque je ne l'utilise pas
5. **Compteur de demi-coups :** Le cinquième champ donne le nombre de demi-coups depuis le dernier coup de pion ou de capture.
6. **Compteur de coups :** Le sixième champ contient le numéro du coup actuel, en commençant à 1.

Exemple : Voici un exemple de chaîne FEN utilisée dans notre projet :

```
1 ChessBoard('4Q3/8/5k2/8/8/8/6K1/8␣w␣-␣-␣0␣1').display()
```

Cette chaîne peut être décomposée comme suit :

- `4Q3/8/5k2/8/8/8/6K1/8` : Configuration du plateau. La dame blanche (`Q`) est en e8, le roi noir (`k`) en f6, et le roi blanc (`K`) en f2.
- `w` : C'est au joueur blanc de jouer.

- `-` : Aucun roque n'est possible.
- `-` : Pas de case de prise en passant disponible.
- `0` : Aucun demi-coup joué depuis la dernière capture ou le dernier mouvement de pion.
- `1` : Premier coup de la partie.

Cette représentation permet de capturer l'état complet de la partie, facilitant l'entraînement de l'agent dans le cadre du Q-learning. J'utilise la librairie python **matplotlib** pour pouvoir afficher un plateau de jeu selon la notation fen pour m'aider à mieux visualisé et débugger le code dans diverses situations. La figure 1 représente le plateau de jeu pris pour exemple pour expliquer la notation fen. Tandis que la figure 2 représente un plateau de jeu complet en début de partie :

Figure 1: Plateau de jeu simple : '4Q3/8/5k2/8/8/8/6K1/8 w - - 0 1'

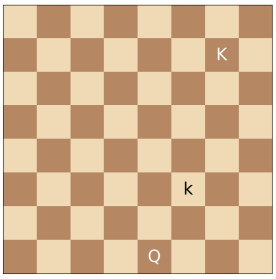
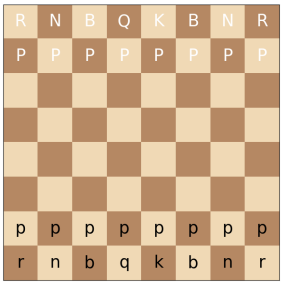


Figure 2: Plateau de jeu complet :
'rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1'



4.2 Qu'est ce qu'un plateau de jeu simple ?

Les finales de jeu d'échecs sont des phases cruciales d'une partie, où le nombre de pièces restantes est réduit et où chaque mouvement peut avoir un impact déterminant sur le résultat de la partie. Parmi les finales les plus simples on trouve la configuration "Roi + Dame contre Roi", où le joueur disposant de la dame et du roi cherche à mater l'adversaire qui ne possède qu'un seul roi.

Dans notre projet, j'ai d'abord concentré mes recherches sur la création de positions simples, en particulier celles composées d'un roi et d'une dame contre un roi.

Je détaillerais plus loin dans le rapport notre générateur de positions FEN respectant cette configuration. Ce générateur va permettre de créer automatiquement ce genre de positions pour nos expérimentations qui serviront à entraîner notre agent à résoudre ces finales simple.

5 Expériences

5.1 Mise en place

Une fois notre algorithme de Q-learning implémenté et notre cadre de jeu simple défini, j'ai cherché à effectuer des expériences afin d'évaluer les performances de l'algorithme. Pour ce faire, j'ai mis en place un générateur de plateaux FEN qui permet de créer des configurations de parties d'échecs sur lesquelles l'agent peut être testé.

Le générateur de plateaux est capable de créer des positions simples, telles que celles composées d'un roi et d'une dame contre un roi, ainsi que d'autres configurations aléatoires tout en respectant certaines conditions. Par exemple, il peut générer des plateaux permettant un mat en un coup, ce qui permet d'explorer différentes situations de fin. Avec ça il va être intéressant de constater le nombre d'épisode que l'algorithme de Q-learning aura besoin pour trouver un échec et mat et ainsi voir ou peut se trouver une limite à cet algorithme de Q-learning.

Le générateur fonctionne à partir de la classe `ChessBoardGenerator`, qui permet de configurer les pièces autorisées sur le plateau, telles que le roi et la dame dans notre cas. Le code permet également de garantir des plateaux valides.

J'ai implémenté une fonction qui enregistre ces positions dans un fichier texte, ce qui permet de les réutiliser ultérieurement pour entraîner et tester notre agent Q-learning sur les mêmes plateau selon différents paramètres d'entraînement ou agents.

5.2 Configuration de différents agents

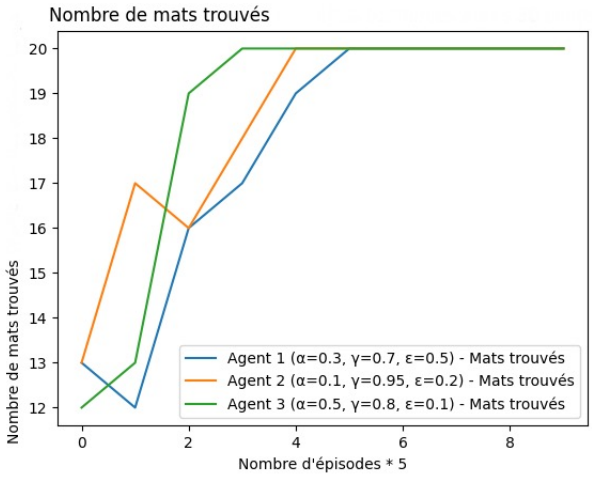
Les agents sont configurés avec des valeurs différentes pour les paramètres d'apprentissage : α (taux d'apprentissage), γ (facteur de discount) et ϵ (taux

d'exploration), permettant ainsi d'explorer diverses stratégies d'apprentissage par renforcement.

- Agent 1 : Avec un taux d'exploration élevé ($\epsilon = 0.5$), un taux d'apprentissage modéré ($\alpha = 0.3$) et un facteur de discount faible ($\gamma = 0.7$), cet agent privilégie l'exploration tout en apprenant à un rythme modéré.
- Agent 2 : Cet agent équilibre exploration et exploitation avec un taux d'exploration modéré ($\epsilon = 0.2$), un taux d'apprentissage faible ($\alpha = 0.1$) et un facteur de discount élevé ($\gamma = 0.95$).
- Agent 3 : Cet agent favorise l'exploitation avec un faible taux d'exploration ($\epsilon = 0.1$), un taux d'apprentissage élevé ($\alpha = 0.5$) et un facteur de discount modéré ($\gamma = 0.8$).

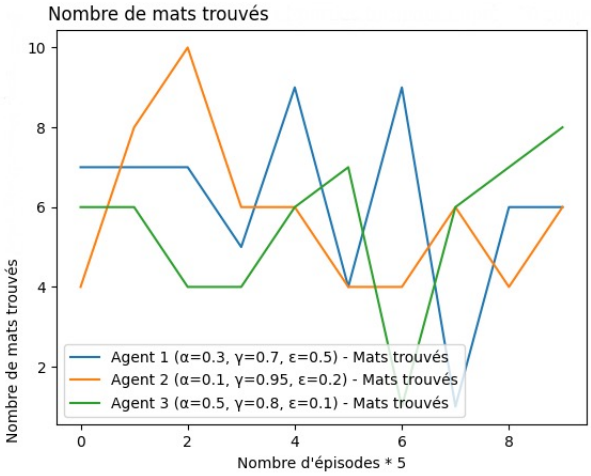
5.3 Un seul coup pour mettre échec et mat

Avec le générateur j'ai générer 20 plateaux simples c'est-à-dire des plateaux ou l'agent peut trouver un coup qui va mettre un échec et mat directement. On constate que dans ce cas le plus simple au bout de 50 épisodes peu importe les hyperparamètres de l'agent celui-ci sera capable de mettre un échec et mat. Le nombre d'épisode commence à 0 et va jusqu'à 100 par pas de 5 ce qui est amplement suffisant pour trouver un échec et mat pour chaque plateau.



5.4 Plateau aléatoire et limite du Q-learning

Maintenant voyons avec des plateaux aléatoires ou il n'y a pas forcément échec et mat en un coups :



Les résultats sont interpellant car pour chaque agent, peu importe les hyperparamètres le nombres d'échec et mat est entre 1 et 10 sur un maximum de 20. En faite, peu importe les politiques des agents ils rencontrent trop souvent des plateaux ou ils tournent en rond et ne parvienne pas à mettre en échec et mat.

Ce phénomène illustre une certaines limites du Q-learning avec des plateau simple. Comme l'espace des états pour les finales d'échecs est vaste, cela rend l'exploration partielle malgré une stratégie ϵ-greedy. Ensuite, le Q-learning utilise une table pour stocker les valeurs $Q(s, a)$, ce qui empêche l'agent de généraliser ses connaissances à des positions similaires. Enfin, les récompenses attribuées uniquement lors de l'échec et mat ou pour des pénalités ne fournissent pas assez de signaux intermédiaires pour guider efficacement l'apprentissage.

C'est pourquoi, les agents peinent à planifier des séquences d'actions complexes, ce qui limite leur performance. Pour améliorer ces résultats, j'ai tenté de mettre en place un approximate Q-learning pour voir si les résultats peuvent devenir meilleurs.

6 Mise en place de l'approximate Q-learning

J'ai adapté la classe `QLearningAgent` pour créé la classe `ApproximateQLearningAgent` et voici les modifications :

1. **Approximation des valeurs** $Q(s, a)$: Au lieu d'utiliser une table pour stocker les valeurs Q pour chaque paire état-action, un modèle de *régression linéaire* est employé pour estimer ces valeurs. Le modèle apprend à partir de caractéristiques extraites des états et des actions.

2. **Extraction de caractéristiques** : Les caractéristiques utilisées incluent la position des rois blancs et noirs, le nombre de pions pour chaque joueur, et

la distance entre les rois. Ces informations permettent de réduire l'espace des états tout en conservant les informations essentielles pour le jeu.

3. **Mise à jour incrémentale du modèle** : À chaque étape, les caractéristiques de l'état-action courant sont ajoutées à un ensemble de données. La cible est calculée avec :

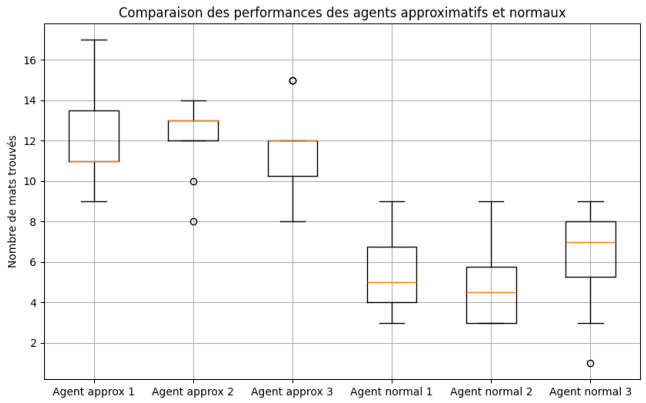
$$Q_{\text{target}} = r + \gamma \max_{a'} Q(s', a')$$

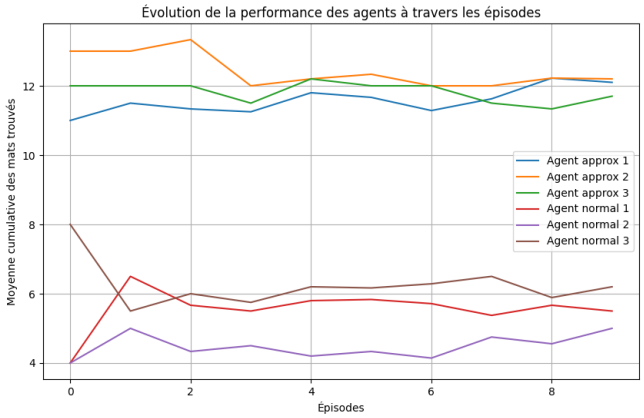
où r est la récompense, γ le facteur de discount, et $Q(s', a')$ la valeur estimée pour le prochain état s'. Le modèle est ensuite réentraîné avec les nouvelles données collectées.

Ces modifications constituent mon approximate Q-learning et maintenant on peut comparer ses performances avec l'agent Q-learning classique

6.1 Analyse des résultats avec l'approximate Q-learning par rapport au Q-learning

L'entraînement est fait sur un ensemble de 20 mêmes plateaux générés aléatoirement, ainsi que sur le même nombre d'entraînement (à savoir 10 entraînement de 10 épisodes).

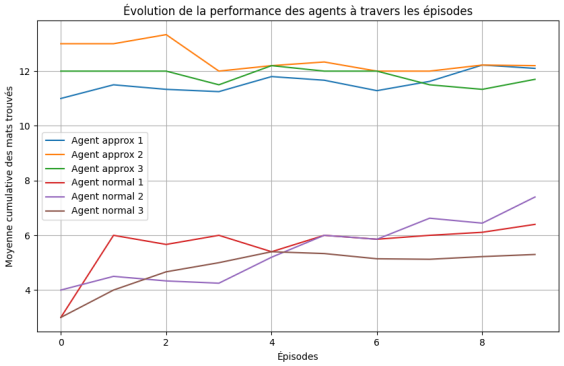




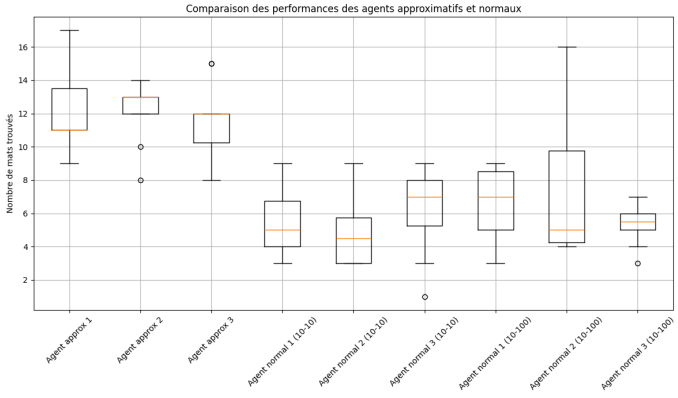
On constate assez clairement que les 3 agents qui performent le mieux sont les agents utilisant le Q-learning approximatif, ils arrivent à trouver entre 8 et 17 échecs et mat sur 20 contrairement aux agents normaux qui sont enter 3 et 9 ce qui est nettement moins bon.

6.2 Analyse des résultats avec l'approximate Q-learning par rapport au Q-learning pour des plateaux moins simple que avec 3 pièces

Pour cette expérience, je garde le roi et la dame qui cette fois ne vont plus affronter juste un seul roi, mais un roi accompagné de 3 pions pour rendre la tâche plus compliqué et ainsi voir la différence de performance entre les agent Q-learning classique et les agents Q-learning approximatif. Les paramêtres sont toujours les mêmes, 20 mêmes plateaux générés aléatoirement (mais avec les 3 pions en plus), ainsi que sur le même nombre d'entraînement (à savoir 10 entraînement de 10 épisodes).



On constate également que les agents Q-learning classique sont nettement moins bon que les agents Q-learning approximatif. Il est intéressant de constater que si on rajoute à ces mesures une augmentation du nombre d'entraînement (10 entraînement de 100 épisodes) il peine aussi à être meilleurs que le Q-learning approximatif même s'il est un peu meilleurs qu'avec moins d'entraînement. Comme le montre le graphiques ci dessous :



7 Pour aller plus loin

Bien que l'approximate Q-learning surpasse le Q-learning classique en raison de sa capacité à généraliser les valeurs Q à partir de caractéristiques extraites, il reste limité pour un jeu tel que les échecs, où l'espace des états est extrêmement vaste. Même si dans mon implémentation on peut encore étudier l'implémentation des features pour améliorer l'approximative Q-learning.
Il serait particulièrement intéressant d'étendre ces recherches en implémentant le *Deep Q-Learning*, qui utilise un réseau neuronal pour approximer les valeurs $Q(s, a)$. Cette méthode pourrait offrir une meilleure capacité de généralisation et des performances accrues sur des environnements aussi complexes que les finales d'échecs ou même des états moins avancer d'une configuration de plateau de jeu.

8 Conclusion

J'ai exploré les limites du Q-learning classique en l'appliquant à des finales d'échecs, avant d'implémenter une approche par Approximate Q-learning pour améliorer ces performance, notamment sur des plateaux plus complexes. Les résultats obtenus sur des plateaux plus évolués montrent une meilleure capacité d'adaptation et d'apprentissage dans des environnements complexes. Ce travail peut être approfondi, notamment en implémentent de meilleures statégies de features extraction pour l'approximate Q-learning et il serait intéressant d'explorer le deep-Q-learning pour le comparer à de l'approximate Q-learning.

9 Références

`https://syzygy-tables.info`
`https://python-chess.readthedocs.io/en/latest/`

10 Annexes

L'ensemble du code utilisé pour le projet se trouve dans le dossier code. Il y a les fichiers **agent_approximate_q_learning.py** et **agent_q_learning.py** avec les implémentations des deux types d'agent. Le fichier **chess_board_generator.py** permet de générer des plateaux selon la notation fen et les mettre dans un fichier, **chess_board_loarder.py** permet de lire ce fichier. Dans **chess_board.py**, il y a l'implémentation du plateau de jeu avec la librairie chess et les méthodes pour agir ou voir l'état du plateau, ainsi que l'afficher. Le fichier **experiences.py** permet de générer l'ensemble des graphiques disponibles dans ce rapport. Et **q_learning_appro_test.py** et **q_learning_test.py** permettent l'implémentation du Q-learning ou de l'approximate Q-learning dans une partie de jeu avec un adversaire qui joue des coups aléatoires.