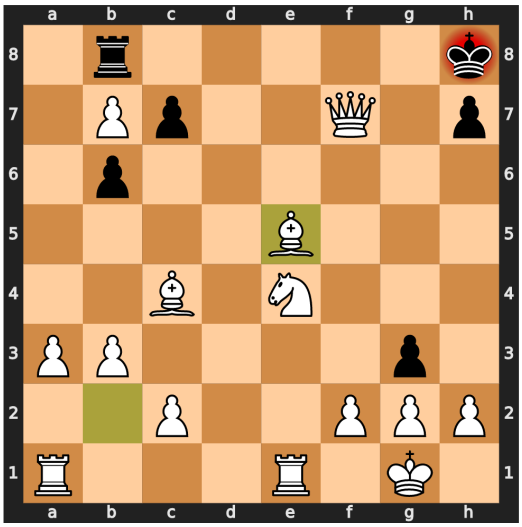
CNAM Paris
Intelligence artificielle avancée

RCP 211

2021 - 2022

semestre 2

Génération de scénarios et renforcement



Par Denis Lemarchand

<u>Professeurs</u>

Clément Rambour

Nicolas Audebert

Nicolas Thome

Sommaire

1 - Introduction

1.1 - Sujet choisi

Vous implémenterez un modèle capable d'échantillonner des objectifs pour un agent qui devra apprendre à les atteindre. Ce projet se décline donc en deux parties. Grâce à un modèle génératif de votre choix (VAE, GAN, Autroregressive model, champs de Markov...) générez des buts à atteindre. Un exemple donnant une certaine complexité dans les scénarios est de reprendre la bibliothèque python-chess, les buts étant alors une configuration spécifique de l'échiquier. Enfin, apprenez à l'agent à réaliser la configuration demandée. Vous pourrez vous inspirer de ce papier et code : Visual Reinforcement Learning with Imagined Goals.

1.2 - Objectifs du projet

J'ai choisi d'explorer le domaine des échecs comme support de ce sujet. Les objectifs que je me suis fixés dès le départ étaient les suivants:

- Comprendre le modèle RIG (Reinforcement Learning with Imagined Goals).
- Faire une courte revue des approches IA dans le domaine des jeux d'échecs.
- Tester différentes approches RL pour tenter de battre un agent de qualité professionnelle tel que "stockfish" et les comparer entre elles.
- Expérimenter les modèles génératifs comme substitut aux simulations de parties réelles pour générer des situations d'apprentissage.
- Implémenter l'approche RIG pour entraîner un agent et tenter de battre le moteur stockfish.
- Renforcer mes compétences avec la programmation Python & framework pour IA.
- Mettre en pratique les concepts vus en cours.

Ces objectifs ont guidé mon exploration de ce domaine et la rédaction de ce rapport.

1.3 - Stratégies pour atteindre les objectifs

J'ai élaboré au préalable quelques idées de stratégies pour atteindre mes objectifs. Les stratégies sont à priori de difficultés croissantes. Je reviendrai dessus dans la conclusion sur ces choix et commenterai si mon exploration m'a conduit à d'autres stratégies.

1. Tester le moteur "stockfish" avec le framework python-chess.
2. Commencer par la génération de labyrinthes ou autre "grid world" avec contraintes.
3. A partir de parties simulées entre deux joueurs "robots", apprendre les configurations du gagnant afin de pouvoir générer des configurations (objectifs) "synthétiques" gagnantes.
4. Reprendre des parties existantes de joueurs professionnels pour apprendre à générer d'autres configurations puis apprendre à un agent d'atteindre ces configurations.

2 - Considérations théoriques

Dans cette partie, je synthétise le contenu de l'article "Visual Reinforcement Learning with Imagined Goals" en mettant l'emphase sur les parties théoriques. Les notations sont celles de l'article et les liens avec les concepts vus en cours ont été fait lorsque cela s'est avéré nécessaire.

2.1 - Introduction

Les algorithmes classiques d'apprentissage par renforcement sont dédiés à l'apprentissage de tâches uniques définies par un schéma de récompenses explicitement décrit. Le but à atteindre est décrit implicitement par le schéma de récompenses.

Les humains apprennent dès leur plus jeune âge à se fixer des objectifs arbitraires pour découvrir le monde qui les entoure. C'est une façon d'apprendre efficiente afin de réaliser des tâches de plus en plus complexes. L'article propose l'approche RIG (Reinforcement learning with Imagined Goals) qui va dans ce sens en définissant un cadre RL permettant d'apprendre une représentation de l'environnement et à atteindre des buts artificiels dans cette représentation. Ainsi en apprenant à atteindre des objectifs aléatoires échantillonnés à partir de la représentation du monde (comprendre ici l'espace latent), la stratégie ainsi conditionnée par les objectifs peut ensuite être utilisée pour atteindre des objectifs spécifiés, cette fois-ci, par l'utilisateur.

L'article a pour cadre d'étude le domaine de la robotique classique avec une représentation du monde par des capteurs sensoriels d'images. Les auteurs notent que dans cet environnement, essentiellement visuel, il devient très utile d'avoir une représentation symbolique du monde afin de fixer des objectifs et une mesure pour les évaluer.

Dans l'article, les auteurs utilisent les termes objectifs, buts ou états à atteindre comme des synonymes. Ils justifient ce choix par souci de généralisation de la notion d'objectif et principalement lorsque l'on a pas de connaissance du domaine comme c'est le cas avec les représentations visuelles des tâches à accomplir.

2.2 - Approche RIG proposée

La méthode d'apprentissage par renforcement avec des objectifs imaginaires (RIG) combine un apprentissage par renforcement, hors politique, c'est à dire que la politique apprise est différente de celle qui est utilisée pour explorer, et conditionnel aux objectifs qui sont des échantillons générés à partir d'un espace latent qui a été formé avec un apprentissage par représentation non supervisé. RIG combine un modèle RL et un modèle génératif. Dans l'article, les auteurs ont utilisé un auto-encodeur variationnel (VAE) pour générer les représentations de l'environnement (l'espace latent) et générer ainsi des configurations à atteindre (les objectifs synthétiques). En outre, l'espace latent permet également de définir une mesure des objectifs (le schéma des récompenses) qui est essentielle pour la couche RL.

La méthode par renforcement appliquée par les auteurs est un modèle Q-learning avec des objectifs d'apprentissage, modèle qui appartient à la classe des GCRL (Goal-conditioned reinforcement learning). Les objectifs aléatoires issus de l'espace latent viennent alors nourrir la mémoire "replay goals" plutôt que de se restreindre aux états observés le long de la trajectoire empruntée. Cette astuce rendrait l'apprentissage plus efficace.

2.3 - Les fondations théoriques de l'approche RIG

La base de l'approche RIG est un apprentissage par renforcement guidé par des objectifs. Les modèles standards RL consistent à maximiser la récompense globale R_0 sous la politique π pour atteindre un seul objectif g telle que :

$$R_0 = E_\pi \left[\sum_{i=0}^{T} \gamma^i r_i \right]$$

Avec $r_i = r(s_i, a_i, s_{i+1})$ la récompense obtenue en choisissant l'action a_i permettant de passer de l'état courant s_i à l'état suivant s_{i+1}. Le facteur de "discount" γ permet quant à lui de pondérer l'importance des récompenses futures. L'apprentissage de tels modèles revient à trouver la politique optimale π, celle qui maximise R_0, pour atteindre l'unique objectif g.

Contrairement aux modèles standards RL, l'approche RL guidée par les buts (GCRL), nous invite à trouver une politique π qui permet d'atteindre un ensemble de buts $g \in G$. Cela consiste à maximiser la récompense globale R sous la politique π et la distribution de G telle que :

$$R = E_g \left[E_\pi \left[R_0 \right] \right]$$

Avec $r_i = r(s_i, a_i, s_{i+1}, g)$ la récompense obtenue spécifiquement pour atteindre le but g.

La méthode Q-learning qui permet d'apprendre une telle politique doit minimiser l'erreur de Bellman suivante :

$$\epsilon(\omega) = \frac{1}{2} \| Q_\omega(s, a, g) - (r + \gamma max_{a'} Q_{\bar{\omega}}(s', a', g)) \|^2$$

Avec $Q_\omega(s, a, g)$ la Q-fonction qui apprend les paramètres ω pour estimer la valeur état-action courante pour le but g en prenant l'action a depuis l'état s et $Q_{\bar{\omega}}(s', a', g)$ la Q-fonction qui renvoie la valeur état-action pour l'état suivant s' avec l'action a'dont les paramètres ω sont fixés. Il s'agit de la stratégie "Fixed Q-target" utilisée pour améliorer la stabilité des modèles de Deep Q-learning.

Le second composant de l'approche RIG est un modèle permettant de générer les buts à atteindre. La manipulation de l'espace latent, qui représente les buts, donc les états à atteindre, n'étant pas requis, l'utilisation d'un auto-encodeur variationnel (VAE) est suffisant. L'encodeur q_ϕ projette un état à atteindre s dans l'espace latent sous la forme d'une variable latente z et le décodeur $p_\phi$réalise l'opération inverse, à partir d'une valeur dans l'espace latent génère un état à atteindre.

La méthode pour apprendre conjointement les paramètres ϕ et ψ est basée sur la maximisation de l'ELBO (evidence lower-bound) :

$\mathcal{L}(\psi, \phi, s^{(i)}) = E_{q_\phi(z|s^{(i)})}\left[log(p_\psi(s^{(i)}|z))\right] - \beta D_{KL}(q_\phi(z|s^{(i)})||p(z))$

Avec D_{KL} la divergence de Kullback-Leibler et β la pondération de l'importance donnée à la structuration de l'espace latent qui se rapproche au plus près de la distribution $p(z)$ choisie.

2.4 - Spécificités de l'approche RIG

La haute dimension de l'espace des observations, je parle ici des images de l'environnement du robot, a contraint les auteurs à utiliser l'espace latent (de dimension bien plus faible) pour représenter les états et donc les buts à atteindre. La politique optimale pour atteindre les buts est donc entièrement définie dans l'espace latent. En définitive, les actions sont exécutées dans l'espace réel du robot mais les réponses de l'environnement (les états atteints) ainsi que les buts fixés sont exprimés dans l'espace latent.

Le β-VAE est tout d'abord entraîné en parcourant l'espace réel du robot avec une politique aléatoire puis l'apprentissage de la politique optimale dans l'espace latent peut alors commencer avec la couche RL guidée par les buts. Les buts sont alors générés par la couche β-VAE. On profite de la phase d'apprentissage de la politique pour compléter l'apprentissage de la couche β-VAE avec les nouveaux états explorés durant cette phase. On voit clairement sur les vidéos, mises à disposition par l'un des auteurs, la phase d'entraînement du β-VAE qui correspond aux soubresauts du robot.

Le choix de la récompense est critique pour la couche RL. Elle est définie comme la distance euclidienne dans l'espace latent entre l'état courant et le but:

$r(s, g) = -||e(s) - e(g)|| = -||z - z_g||$

Avec la fonction encodeur e, on obtient dans l'espace latent les transformées des états et des buts : $z = e(s)$ et $z_g = e(g)$. Techniquement, on utilise alors la moyenne[1] de l'encodeur comme code z.

Travaillant dans l'espace latent, l'approche RIG nécessite un encodeur pour projeter une observation X, dans cet espace, afin d'obtenir le code z correspondant. Ceci exclut donc l'utilisation des modèles GAN pour cette tâche. En effet ils ne permettent pas d'association directe $X \rightarrow z$. Il faudrait implémenter un encodeur pour réaliser cette inversion.

[1] Dans une implémentation classique de VAE, l'encodeur s'appelle : `mu,logvar = encoder(x)`

2.5 - L'algorithme RIG proposé

Algorithm 1 RIG: Reinforcement learning with imagined goals

Require: VAE encoder q_ϕ, VAE decoder p_ψ, policy π_θ, goal-conditioned value function Q_w.

1: Collect $\mathcal{D} = \{s^{(i)}\}$ using exploration policy.
2: Train β-VAE on $\mathcal{D}$ by optimizing (2).
3: Fit prior $p(z)$ to latent encodings $\{\mu_\phi(s^{(i)})\}$.
4: **for** $n = 0, ..., N-1$ episodes **do**
5: Sample latent goal from prior $z_g \sim p(z)$.
6: Sample initial state $s_0 \sim E$.
7: **for** $t = 0, ..., H-1$ steps **do**
8: Get action $a_t = \pi_\theta(e(s_t), z_g) +$ noise.
9: Get next state $s_{t+1} \sim p(\cdot \mid s_t, a_t)$.
10: Store (s_t, a_t, s_{t+1}, z_g) into replay buffer $\mathcal{R}$.
11: Sample transition $(s, a, s', z_g) \sim \mathcal{R}$.
12: Encode $z = e(s), z' = e(s')$.
13: (Probability 0.5) replace z_g with $z'_g \sim p(z)$.
14: Compute new reward $r = -||z' - z_g||$.
15: Minimize (1) using (z, a, z', z_g, r).
16: **end for**
17: **for** $t = 0, ..., H-1$ steps **do**
18: **for** $i = 0, ..., k-1$ steps **do**
19: Sample future state $s_{h_i}, t < h_i \leq H-1$.
20: Store $(s_t, a_t, s_{t+1}, e(s_{h_i}))$ into $\mathcal{R}$.
21: **end for**
22: **end for**
23: Fine-tune β-VAE every K episodes on mixture of $\mathcal{D}$ and $\mathcal{R}$.
24: **end for**

Fig 1 : algorithme RIG en pseudo code

Les lignes 1 à 3 couvrent l'apprentissage du β-VAE, les lignes 7 à 16 couvrent l'apprentissage RL guidée par les buts (ici utilisation d'une approche TD3[2]). A la ligne 9, on utilise l'environnement pour obtenir l'état suivant. A la ligne 23, on relance une phase complémentaire d'apprentissage du β-VAE avec les nouveaux états explorés durant l'apprentissage RL. La double boucle des lignes 17 à 22, permet de nourrir la mémoire "replay goals" à partir des états explorés qui deviennent eux-même des nouveaux buts à atteindre.

3 - Considérations expérimentales

3.1 - Environnement technique

J'ai utilisé l'environnement cloud Google Colab Pro qui est bien adapté au Deep Learning avec la mise à disposition de GPU à la demande. J'ai utilisé la librairie python-chess qui offre un environnement complet de jeu d'échecs interfaçable avec un agent programmable.

La librairie python-chess ne fournit pas les agents alors j'ai utilisé comme agent de référence le célèbre moteur open source stockfish[3]. Cet agent implémente un algorithme minimax sur l'arbre des possibilités de jeu avec un élagage alpha-beta ainsi qu'une base de données des tables de finale.

Le moteur stockfish peut être paramétré notamment sur la profondeur de recherche dans l'arbre[4] ainsi que le nombre de nœuds visités. C'est une façon élégante de lui donner un

[2] Twin Delayed Deep Deterministic Policy Gradients
[3] Lire l'article sur Wikipedia pour plus de détails.
[4] Lire la documentation sur les limites que l'on peut imposer :
https://python-chess.readthedocs.io/en/latest/engine.html#chess.engine.Limit

handicap variable pour tester mes agents. En effet, le moteur stockfish est très puissant et a récemment été détrôné par AlphaZero en 2017. Ce qui, de mon point de vue, est une référence quasi impossible à battre avec les agents que j'ai testé.

Pour tester la librairie python-chess, j'ai utilisé les agents RLC (Reinforcement Learning Chess) implémentés par Arjan Groen permettant de mettre en œuvre des approches déjà vues en cours mais pas nécessairement appliquées dans les TP. Je pense notamment au modèle Monte Carlo Tree Search (MCTS) qui est implémenté par Arjan Groen dans un agent qui combine MCTS pour parcourir l'arbre et TD Deep Q-Learning pour estimer la valeur d'un nœud. Les agents RLC utilisent TensorFlow qui est un framework comparable à Pytorch. Il existe deux modes d'apprentissage des agents RLC. Le mode "capture" dont les récompenses sont calculées en fonction de la valeur des pièces[5] adverses capturées par l'agent. Ce mode apprend à maximiser la somme de toutes les captures obtenues jusqu'à ce que la partie se termine, gagnée ou non. Le mode "réel" dont les récompenses sont calculées comme la somme des captures et du résultat obtenu en fin de partie. Ce mode apprend à maximiser les gains des captures avec un bonus supplémentaire en cas de victoire finale.

3.2 - Expérimentation

Voici les tests et expérimentations que j'ai réalisés:

1. Faire jouer un agent RLC en mode "capture", préalablement entraîné avec la méthode "Reinforce" contre un opposant avec stratégie "aléatoire", à jouer les blancs avec le moteur stockfish en position des noirs.

2. Dans la stratégie "capture", l'agent RLC implémenté par Arjan Groen suppose que l'opposant est un agent "aléatoire" lors de l'apprentissage avec la méthode "Reinforce". Remplacer cet agent par le moteur stockfish. Reproduire le cas 1 et comparer.

3. Dans la stratégie "capture", l'agent RLC implémenté par Arjan Groen ne prend pas en compte, dans la récompense, les victoires mais uniquement les captures. Ajouter une récompense importante pour les victoires. Reproduire le cas 1 et comparer.

[5] La valeur des pièces respecte la valeur de Reinfield. La valeur totale, pour un joueur, au début d'une partie est de 8x1 + 2x5 + 2x3 + 2x3 + 1x9 = 39. Attention en cours de partie, la valeur accumulée des captures peut dépasser 39 car il y a la promotion des pions qui fait partie des règles des échecs.

La courbe d'apprentissage de l'agent RLC (Fig 2.) en mode "capture" entraîné avec la méthode "Reinforce" contre un opposant avec stratégie "aléatoire" semble satisfaisante après 6000 parties (on atteint une capture presque totale des pièces de l'adversaire).

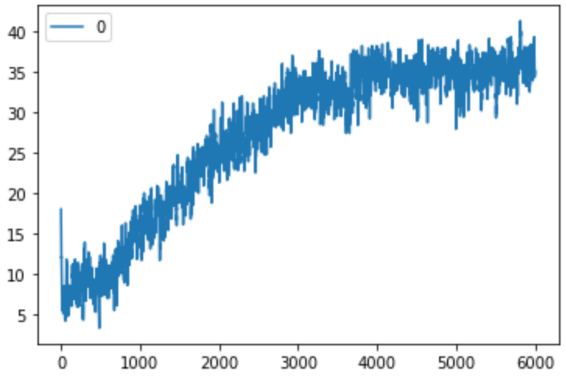


Fig 2 : courbe d'apprentissage agent RLC

On effectue un test de contrôle qualité en faisant jouer un agent RLC Reinforce contre lui-même pour s'assurer que les résultats sur 1000 parties sont cohérents. Les blancs ayant l'avantage et l'agent jouant avec la même stratégie des 2 bords, il serait normal que les noirs ne gagnent jamais. C'est effectivement ce que l'on constate sur la figure 3 ci-dessous.

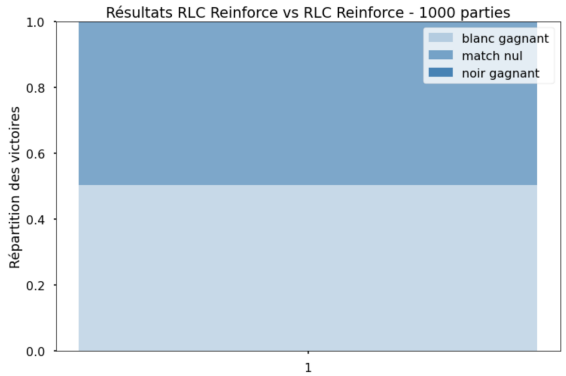


Fig 3 : résultats RLC contre RLC

On effectue un test de contrôle de performance pure (en termes de victoire) en faisant jouer un agent RLC Reinforce contre un agent "aléatoire" sur 1000 parties. On constate sur la figure 4 ci-dessous que l'agent RLC Reinforce est à peine meilleur qu'un agent aléatoire, sachant qu'il a les blancs. On obtient principalement des matchs nuls. Toutefois on ne compte pas en cas de match nul la valeur des captures réalisées par chaque camp.

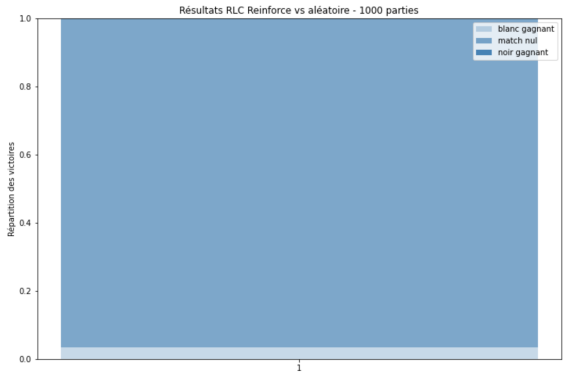


Fig 4 : résultats RLC contre agent aléatoire

En outre, lorsque l'on place l'agent RLC Reinforce en position des noirs, les résultats sont désastreux. On obtient 29% de matchs nuls, 70% de victoire pour l'agent aléatoire et seulement 1% pour l'agent RLC. Ceci peut s'expliquer par l'avantage tactique des blancs détenu par l'agent aléatoire et le fait que RLC semble à peine meilleur qu'un agent aléatoire (Fig 4.).

Les résultats pour le cas 1 (RLC Reinforce contre stockfish), illustrés ci-dessous (Fig 5.), vont me servir de base de comparaison pour la suite des tests. On limite le moteur stockfish à une profondeur de parcours de 4 niveaux (4 coups en avance) pour 4 nœuds (parcours en largeur de 4 stratégies) car il est très efficace.

Avec 10 nœuds, son taux de succès est déjà à 80% de victoires contre 1% pour RLC et 9% de parties nulles. On limite aussi

le moteur en temps de “réflexion” à 0.1 seconde.

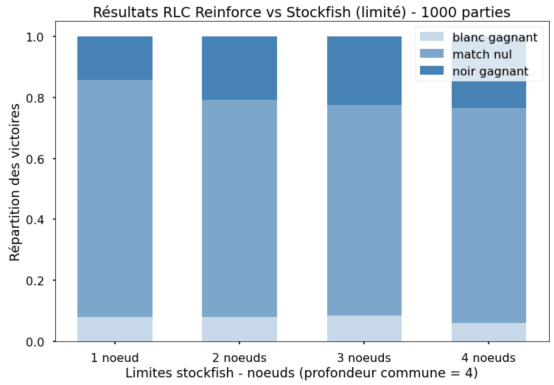


Fig 5 : résultats RLC contre stockfish

La courbe d'apprentissage de l’agent RLC en mode “capture” entraîné avec la méthode “Reinforce” mais cette fois-ci contre le moteur stockfish semble relativement satisfaisante après 6000 parties (Fig 6.). Cependant on obtient une moins bonne capture des pièces de l’adversaire que précédemment avec une quantité identique de parties d’apprentissage.

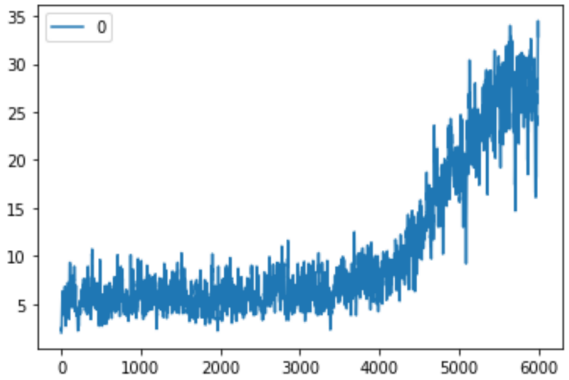


Fig 6 : apprentissage RLC avec stockfish

On rejoue le cas 1 (RLC Reinforce contre stockfish) et on obtient cette fois-ci de moins bons résultats que précédemment tels qu’illustrés sur la figure 7. RLC ne remporte quasiment aucune victoire (moins de 0.5%).

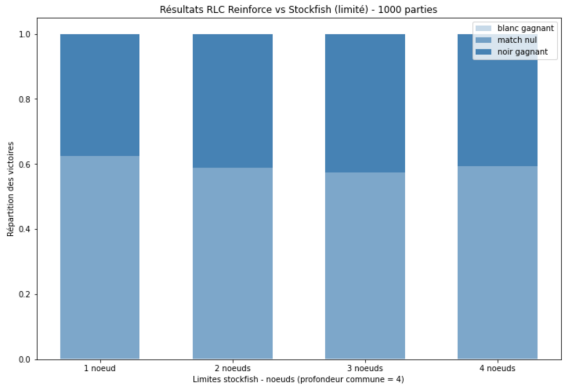


Fig 7 : RLC entraîné et jouant contre stockfish

On relance cette fois-ci l’apprentissage en ajoutant une récompense importante pour les victoires et on augmente le nombre de parties d’apprentissage à 8000 (Fig 8.).

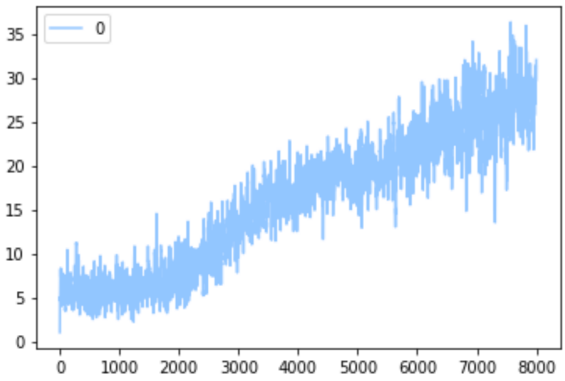


Fig 8 : apprentissage RLC avec récompense accrue pour les victoires

On rejoue le cas 1 (RLC Reinforce contre stockfish) et on obtient cette fois-ci encore de moins bons résultats que précédemment tels qu’illustrés ci-dessous (Fig 9.). RLC ne remporte quasiment aucune victoire (moins de 0.5%) ni de match nul.

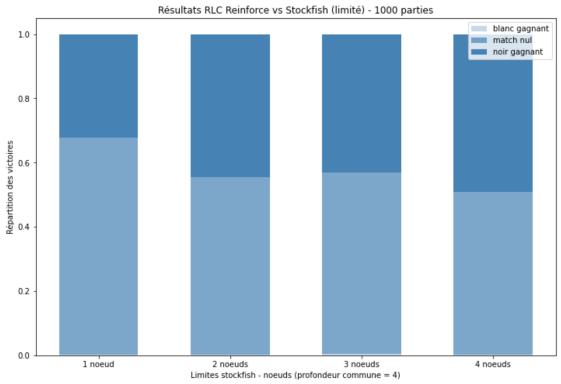


Fig 9 : RLC entraîné avec récompense accrue pour les victoires et jouant contre stockfish

On peut convenir que j'ai atteint une limite avec cette approche d'apprentissage. En effet, l'agent RLC semble plus performant durant une partie d'échec (en termes de victoires) lorsque son apprentissage a été effectué avec un opposant aléatoire et sans récompense pour la victoire. Je vais m'orienter sur une approche RIG et vérifier, dans un contexte identique (même opposant), si elle est significativement plus efficace. C'est le thème principal de ce projet et l'objet du prochain chapitre.

4 - Implémenter une approche RIG pour les échecs

4.1 - La génération des configurations de l'échiquier

Les approches de type GAN sont incontestablement en tête de file pour la génération synthétique d'observations. Les auto-encodeurs, par contre, sont très "mauvais" pour cette tâche car la distribution dans leur espace latent est inconnue. Leur principal emploi réside dans les tâches de réduction de dimension d'espace et de filtre de débruitage de signaux. Entre ces deux approches, les modèles VAE peuvent être à considérer comme modèle génératif lorsque la manipulation sémantique via l'espace latent n'est pas l'objectif principal.

Utiliser un modèle génératif comme substitut aux simulations de parties réelles pour générer des situations d'apprentissage est indispensable pour implémenter une approche RIG. Créer des objectifs à atteindre pourrait prendre beaucoup trop de temps avec des simulations de parties. Il faudrait à chaque fois que nécessaire, faire jouer 2 robots, sauvegarder les configurations obtenues en cours de parties et piocher dans ces configurations de l'échiquier obtenues le long du parcours. Utiliser un générateur de configurations synthétiques à partir d'échantillons aléatoires tirés d'une loi à priori est beaucoup plus simple à mettre en œuvre. C'est ce que je vais expliquer et tester dans la suite de ce paragraphe.

Dans le domaine des échecs, la génération d'une configuration de l'échiquier comme but à atteindre pourrait être considérée comme similaire à une image de dimension 8x8. Ceci permet de faire un rapprochement avec les données manipulées dans l'article (des images), au détail près que la dimension de l'espace des observations reste faible et sans doute manipulable tel que par notre algorithme. De plus, cela m'oriente dès le départ sur une architecture de VAE convolutif.

Je vais donc commencer à implémenter un modèle β-VAE qui va apprendre à générer des configurations d'échiquier plausibles. Le modèle β-VAE est bâti sur la base d'un CNN à 2 couches pour coller le plus possible à la représentation d'un échiquier. L'algorithme que je propose pour l'apprentissage de notre β-VAE reprend une partie de la stratégie vue dans l'article:

1. Configurer une instance de stockfish, une zone mémoire D pour stocker les configurations réelles d'échiquiers.
2. Tant que N configurations n'ont pas été mémorisée:
    a. Initialiser une nouvelle partie.
    b. Tant que la partie n'est pas terminée:
        i. Faire jouer stockfish dans la position des blancs/noirs.
        ii. Stocker l'état de l'échiquier dans D.
3. Séparer D en 2 parties, DA pour l'apprentissage et DT (de taille K = 20% N) pour les tests.

4. Entraîner β-VAE sur DA, en utilisant les techniques habituelles de chargement de batchs aléatoires.
5. Tester le générateur β-VAE:
    a. Pour chaque configuration dans DT:
        i. Passer chaque configuration dans l'encoder-decoder du β-VAE pour évaluer sa qualité de reconstruction. Idéalement, on devrait reconstruire l'entrée identiquement.
        ii. Utiliser la validation de python-chess pour évaluer la qualité de chacune des configurations reconstruites. On répond dans ce cas à la question suivante : même si la reconstruction n'est pas identique à l'entrée, est-elle valide ?
    b. Pour K génération depuis le prior (on génère des codes latents z):
        i. Soumettre chaque génération à la validation de python-chess pour évaluer la qualité de chacune des configurations synthétiques. On répond dans ce cas à la question suivante : est-ce une configuration synthétique valide ?
    c. Sortir les résultats des tests

J'ai choisi une représentation de la configuration d'un échiquier par un tenseur (6, 8, 8) : 6 canaux de taille 8x8. Chaque canal correspond à un type de pièces (dans l'ordre : pion, tour, cavalier, fou, reine, roi). La position d'une pièce est identifiée dans le canal de son type, avec une valeur 1 (blanche) ou -1 (noire) dans la case correspondante du tableau 8x8. Par exemple, le tableau du canal 3 (cavaliers) représente la position des cavaliers de l'échiquier ci-dessous (Fig 10.).

```
tensor([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  1.,  0.,  0.,  1.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0., -1.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0., -1.,  0.,  0.,  0.,  0.,  0.,  0.]])
```

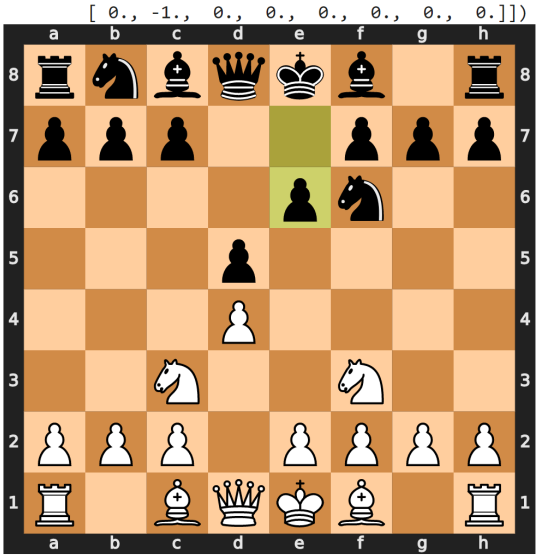


Fig 10 : représentation par un tenseur de la configuration des cavaliers sur l'échiquier

Le défi que je vois est que le β-VAE va traiter des données “ternaires” éparses, un peu comme une succession de 6 matrices creuses composées uniquement de -1, 0, et 1. Je pense que cela aura un impact sur la stabilité et la qualité de l’apprentissage. J’ai bien évidemment normalisé les matrices dont les valeurs possibles des éléments sont 0, 0.5 et 1 mais cela ne change pas la nature du défi présenté.

Le jeu de données utilisé pour l’apprentissage possède une taille de 20000 configurations (espace mémoire DA) et pour les tests une taille de 4000 configurations (espace mémoire DT). J’ai également généré 4000 configurations synthétiques à partir du prior gaussien pour le test b.

Le jeu de données d’apprentissage a été généré à partir de 123 parties jouées entre 2 robots stockfish. Le nombre moyen de coups par partie est de 162. Les blancs ont gagné 39 fois, les noirs 31 et il y a eu 53 matchs nuls. Ce qui représente un jeu relativement équilibré. Je n’ai par contre aucune information sur la distribution des variétés des configurations obtenues.

Les résultats des tests de performance de la génération des configurations avec le générateur β-VAE entraîné sur DA sont rassemblés dans le tableau 1 ci-dessous.

Architecture β-VAE	Paramètres et hyperparamètres[6]	Taux d’échec (test a) 4000 obs.	Taux d’échec (test b), 4000 obs.
CNN 2 couches (6x11) (11x11) Sortie Sigmoid kernel_size=3, stride=1, padding=1	beta = 0.1 batch_size=128 latent_dim=10 epochs=20 lr=0.001	taux de différence=100% (reconstruction à l’identique de l’entrée) taux d’invalidité=75% (configuration reconstruite valide au sens des échecs)	85% (configuration synthétique valide au sens des échecs)
	beta = 1	pas d’impact significatif	73%
	beta = 3	taux d’invalidité=100%	100%
	beta = 0.5	pas d’impact significatif	73%
	beta = 0.3	pas d’impact significatif	73%
	latent_dim=100	pas d’impact significatif	80%
	latent_dim=256	pas d’impact significatif	88%
	batch_size=32	pas d’impact significatif	81%

[6] Le seuil de probabilité pour la conversion entre la sortie Sigmoid et la classe correspondante (-1, 0, ou 1) est fixé à 50% pour tous les tests.

Architecture β-VAE	Paramètres et hyperparamètres[6]	Taux d'échec (test a) 4000 obs.	Taux d'échec (test b), 4000 obs.
	beta = 0.3 latent_dim=100 batch_size=32	taux de différence=100% taux d'invalidité=71%	68%
CNN 2 couches (6x32) (32x6) Sortie Sigmoid kernel_size=3, stride=1, padding=1	beta = 0.3 latent_dim=100 batch_size=32 epochs=20 lr=0.001	taux de différence=100% taux d'invalidité=72%	75%
CNN 2 couches (6x12) (12x24) Sortie Sigmoid kernel_size=3, stride=1, padding=1	beta = 0.3 latent_dim=100 batch_size=32 epochs=20 lr=0.001	pas d'impact significatif	pas d'impact significatif
CNN 2 couches (6x32) (32x32) Sortie Sigmoid kernel_size=5, stride=1, padding=2	beta = 0.3 latent_dim=100 batch_size=32 epochs=20 lr=0.001	pas d'impact significatif	pas d'impact significatif

Table 1 : résultats des tests de performance de la génération des configurations

Le modèle β-VAE implémenté et entraîné sur les configurations réelles de parties entre 2 agents robot n'est pas à la hauteur de mes attentes initiales. Au mieux, j'arrive à générer à partir du "prior" gaussien 25% de configurations plausibles (c'est à dire valide au sens des règles des échecs). Ceci pourrait convenir pour l'algorithme RIG dans la génération des buts. Il faudra les filtrer avant de pouvoir les utiliser comme tels. Néanmoins, une spécificité importante dans RIG est de travailler directement sur les représentations des états courants de l'échiquier et des objectifs dans l'espace latent, ainsi que la mesure des récompenses basée sur une distance entre les codes dans cet espace latent. Or les résultats ci-dessus indiquent que quasiment 100% des reconstructions sont strictement différentes de l'entrée (systématiquement le VAE n'arrive pas à reproduire l'entrée). L'impact pour l'agent RIG que je souhaite implémenter est que la couche β-VAE ne garantit aucunement que l'encodage des entrées est pertinent pour effectuer un apprentissage de la stratégie de jeu (par la couche RL). En effet les objectifs donnés à cette couche sont des états potentiellement "non plausibles". J'ai donc à priori des doutes sur la qualité des objectifs générés.

J'aurais pu croire qu'en augmentant considérablement la taille du jeu de données pour l'apprentissage améliorerait les résultats obtenus mais il n'en fût rien. J'ai essayé avec 80 000 configurations (pour rappel, une configuration est un état de l'échiquier à un instant donné dans l'une des parties jouées entre les deux robots) et les taux d'échec (ie. de succès) restent

sensiblement les mêmes. J'envisagerai alors de revoir la représentation des configurations d'échiquier permettant plus de nuances et de permettre au VAE de reconstruire l'entrée fidèlement. Je pense notamment à utiliser une image de l'échiquier plutôt qu'une grille numérique. Néanmoins, je n'aurais pas de validation automatique des images reconstruites (dans le cas de la grille, c'est la librairie python-chess qui se charge de cette tâche). Il me sera donc difficile de comparer statistiquement les taux d'échec des 2 approches. En revanche, l'approche RIG n'oblige pas à avoir une reconstruction puisque l'apprentissage se fait sur des objectifs et des retours directement dans l'espace latent. Je laisse donc ouverte cette option pour une prochaine suite à ce projet.

Deux autres pistes seraient également à considérer. La première serait d'utiliser un modèle GAN plutôt qu'un β-VAE. Je vois dans cette approche GAN un bon atout pour le problème de validité des objectifs générés. En effet, la qualité de la génération au sens du réalisme (plausible) est au cœur de l'architecture des GAN avec son discriminateur chargé d'évaluer ceci. Néanmoins, le GAN n'ayant pas d'encodeur, il ne sera pas possible de travailler la stratégie RIG dans l'espace latent comme le propose l'article. Cependant, les motivations premières à travailler dans l'espace latent étaient la grande dimension de l'espace d'observation dans le cadre de la robotique pilotée par la vision. Ceci n'est bien évidemment pas le cas pour un échiquier. En dérogeant à l'approche RIG exposée dans l'article, c'est-à-dire à rester dans l'espace des observations, je considère cette avenue comme une option à creuser. La seconde piste consiste à revenir à un modèle plus simple d'auto-encodeur classique (AE). Je laisse donc ouverte cette seconde option.

A ce stade, il me resterait un dernier test à faire pour le β-VAE qui reste malgré tout un candidat pour l'approche RIG. Est-il plus performant qu'une génération purement aléatoire ? Si la génération aléatoire, mais tout de même guidée par les règles des échecs, obtient un score supérieur à 25% de générations valides alors mon β-VAE serait-il à éliminer des candidats ? En réfléchissant à ce qui est mentionné dans l'article, je travaillerais uniquement avec l'encodeur car la reconstruction n'est pas pertinente pour l'approche RIG. Dans ce cas, il conviendrait de maximiser la structuration de l'espace latent pour le contraindre à se rapprocher d'une distribution normale centrée réduite. Finalement générer des buts reviendrait simplement à effectuer des tirages d'une loi normale centrée réduite. Pour ce faire, il convient alors de prendre un β assez élevé (dans l'article, ils suggèrent de prendre $\beta \in [1, 10]$).

Les résultats des tests de performance de la génération des configurations avec le générateur β-VAE entraîné avec un β plus élevé et un seuil de probabilité de 20% pour la conversion entre la sortie Sigmoid et la classe correspondante (-1, 0, ou 1) sont relativement concluants comme présentés ci-dessous dans le tableau 2 :

Architecture β-VAE	Paramètres et hyperparamètres	Taux d'échec (test a) 4000 obs.	Taux d'échec (test b), 4000 obs.
CNN 2 couches (6x16) (16x32) Sortie Sigmoid kernel_size=3, stride=1, padding=1	**beta = 2** batch_size=32 latent_dim=100 epochs=20 lr=0.001 **p=0.2**	taux de différence=100% (reconstruction à l'identique de l'entrée) taux d'invalidité=87% (configuration reconstruite valide au sens des échecs)	**47%** (configuration synthétique valide au sens des échecs)
CNN 2 couches (6x11) (11x11) Sortie Sigmoid kernel_size=3, stride=1, padding=1	**beta = 5** **batch_size=128** latent_dim=100 epochs=20 lr=0.001 **p=0.2**	taux de différence=100% (reconstruction à l'identique de l'entrée) taux d'invalidité=**0%** (configuration reconstruite valide au sens des échecs)	**2.7%** (configuration synthétique valide au sens des échecs)

Table 2 : résultats des tests de performance de la génération des configurations avec bêta plus élevé

Au bout du compte, après plusieurs essais et erreurs, une analyse des résultats et quelques réflexions sur la nature de l'approche RIG, je conclue que le générateur β-VAE avec les bons paramètres s'avère être un candidat potentiel pour les besoins de génération de configurations synthétiques. Il reste un dernier point à valider en ce qui concerne la variété des configurations générées. Lors d'un rapide contrôle visuel sur les 4000 générations effectuées, je retrouve souvent les mêmes configurations ou de faibles variations d'une même configuration. Je pencherais sur un problème au niveau du jeu de données DA.

En effet, en faisant diminuer le paramètre β la diversité des générations augmente en même temps que le taux d'échec. J'ai constaté également que la diversité augmente (et le taux d'échec aussi) si on ne mémorise pas toutes les configurations après chaque coup joué mais en les choisissant aléatoirement après quelques coups (lorsque la partie est un peu avancée pour capturer les ouvertures classiques). Que la diversité augmente lorsque β diminue s'expliquerait selon moi par notre fonction d'optimisation qui prend plus en compte la vraisemblance avec les entrées. En effet, on force un peu plus le VAE à mieux reconstruire les entrées. Un β élevé lui fait porter une attention particulière à structurer la distribution dans l'espace latent et finalement à produire des configurations "moyennes". Aussi, quand il y a plus de diversité, on se retrouve avec des configurations ayant plus de pièces sur l'échiquier et donc il y a plus de chance que la configuration soit invalide (au sens des règles des échecs). Ceci expliquerait la montée du taux d'échec.

En conclusion, il faut choisir entre plus de diversité avec des taux d'échec plus élevés (configurations invalides) ou moins de diversité mais avec des taux de succès plus élevés (configurations valides). Je pencherais pour avoir un peu plus de diversité dans les buts à atteindre quand bien même certains seraient potentiellement inatteignables (car la configuration générée est invalide après une reconstruction qui comporte elle-même une certaine variabilité).

Au sujet des taux d'échec constatés, une réflexion, suite au cours sur l'incertitude des prédictions des modèles, m'a amené à me poser la question de l'introduction d'un système de rejet automatique des générations basé sur la confiance envers les générations produites. A ce stade de mon projet, j'ai pu détecter la validation des configurations synthétiques à postériori grâce à la librairie python-chess qui se charge de cette tâche. Comment pourrait-on s'y prendre pour rejeter une génération synthétique en se basant sur la confiance à priori que le modèle a envers cette génération ? En d'autres termes, comment faire pour diminuer le taux d'échec que je mesure ? Ce point, au regard de ce que j'ai mentionné précédemment, n'est pas primordial pour la suite de mon projet mais il m'interpelle beaucoup. J'imagine que l'on pourrait utiliser une approche similaire à ConfidNet pour apprendre à donner un niveau de confiance à chaque position sur l'échiquier et décider alors de placer ou non la pièce. Quelques articles[7] tournent autour de ce sujet et je les ai mentionnés dans les références. Je n'ai pas investigué ces pistes plus loin par manque de temps mais je les note pour d'éventuelles recherches ultérieures.

4.2 - Apprentissage RL guidé par les buts

L'implémentation RIG, proposée dans l'article [1], met en oeuvre l'algorithme RL TD3[8] qui est une variante de l'algorithme Actor-Critic avec 2 paires de critique, un "experience replay buffer", une portion d'exploration stochastique lors de l'apprentissage et une mise à jour retardée de la politique. Je pensais donc m'appuyer sur cet algorithme, en particulier sur une implémentation fournie via un article de Donal Byrne[9], pour réaliser le cœur de mon algorithme RIG.

L'implémentation TD3 de Donal Byrne, comme la plupart des agents RL que j'ai rencontrés dans mes lectures, s'appuie sur l'environnement gym de open AI. Pour minimiser les éventuels changements à apporter aux codes des agents que je compte utiliser, j'ai décidé de créer un environnement pour le jeu d'échecs respectant l'interface gym. Il faut alors décrire l'espace des actions et des observations. Dans notre cas, les actions sont des déplacements de pièces[10] et les observations sont les configurations de l'échiquier[11]. Il faut ensuite implémenter la méthode "step" qui correspond à effectuer un déplacement sur l'échiquier et récupérer la réponse de l'adversaire sous la forme de la nouvelle configuration de l'échiquier ainsi que la

[7] Articles : Coupled VAE: Improved Accuracy and Robustness of a Variational Autoencoder [2], Variational Auto-Encoder: not all failures are equal [3]
[8] Article : Addressing Function Approximation Error in Actor-Critic Methods [4]
[9] https://towardsdatascience.com/td3-learning-to-run-with-ai-40dfc512f93, consulté le 26 mai 2022.
[10] Se reporter à la documentation python-chess et la classe "Move"
[11] Se reporter à la documentation python-chess et la classe "Board", ainsi que la représentation interne que j'ai proposée, c'est-à-dire un tenseur (6, 8, 8), pour représenter un échiquier.

récompense obtenue. Le calcul de la récompense comporte quelques subtilités relatives à notre approche RIG dont les récompenses sont calculées dans l'espace latent et non l'espace réel. En outre, les méthodes qui permettent d'initialiser l'environnement (positionnement initial des pièces sur l'échiquier pour débuter une nouvelle partie) et d'afficher l'échiquier dans sa configuration actuelle sont à implémenter également.

Il existe également une implémentation TD3 offerte par la fameuse librairie stable-baselines[12] qui semble être intéressante. En effet, celle-ci est déjà compatible avec l'interface d'environnement Gym, offre la possibilité de paramétrer tous les aspects de TD3 (couches, type de politique, …), ainsi que d'ajouter des éléments dans le replay-buffer. La partie cruciale de RIG étant d'utiliser les représentations des observations, cela reste donc l'enjeu à régler pour utiliser TD3 stable-baselines. C'est une piste à explorer pour ceux qui veulent bâtir sur une librairie de la communauté IA plutôt que de réinventer systématiquement la roue. Je n'ai pas été jusque là et j'ai préféré reprendre et modifier une implémentation existante. Cependant j'ai tout de même testé l'utilisation de TD3 stable-baselines et réalisé son apprentissage standard pour valider le bon fonctionnement de mon implémentation de l'environnement du jeu d'échecs[13]. Ceci m'a amené à utiliser un outil[14] fourni par stable-baselines pour valider mon implémentation. Il ressort de l'analyse faite par l'outil que les observations (configurations de l'échiquier) auraient avantage à être représentées sous la forme aplatie d'un vecteur 1d de taille 6x8x8 plutôt qu'un tenseur (6,8,8). C'est ce que j'ai mis en place dans la suite de mon développement.

Après quelques lectures[15], je me suis aperçu que TD3 n'était pas approprié pour les espaces d'actions discrets, ce qui est notre cas présentement. Je décidais donc de me rabattre sur une implémentation RIG basée sur A2C ou tout autre algorithme compatible avec la nature discrète de mon espace d'actions tel que DQN. Un point qui a attiré mon attention dans la conception de l'environnement est que faire si l'action proposée par l'agent n'est pas possible (un déplacement de pièce qui n'a pas de sens) ? J'ai pensé à avoir 2 modes, l'un qui abandonne carrément la partie, le second qui laisse l'opposant jouer (l'agent apprenant passe son tour) et l'environnement retourne une mauvaise récompense (éventuellement négative). Le second mode est à mon avis plus souple et plus propice à l'apprentissage car on laisse la partie se dérouler et on continue à explorer les configurations de jeux en alimentant "l'experience replay".

J'ai repris comme base de code le TP numéro 3 du bloc RL du cours RCP 211. Il s'agit d'une implémentation DQN avec expérience replay. J'ai commencé par adapter légèrement le code pour qu'il puisse fonctionner avec mon environnement Gym "ChessForRigEnv". J'ai ensuite lancé l'apprentissage sans implanter les spécificités RIG et j'ai fait jouer l'agent DQN contre un agent "aléatoire". Les résultats, illustrés ci-dessous (Fig. 11), sont, sans surprise,

[12] TD3 stable-baselines : https://stable-baselines.readthedocs.io/en/master/modules/td3.html#
[13] Classe nommée ChessForRigEnv
[14] Voir la documentation : https://stable-baselines.readthedocs.io/en/master/common/env_checker.html
[15] Entre autre cette lecture : https://dl.acm.org/doi/fullHtml/10.1145/3508546.3508598

aussi mauvais que le test que j'avais fait précédemment (agent RLC Reinforce contre un agent "aléatoire"). C'est tout à fait normal puisque la conclusion à laquelle j'étais arrivée avec cet algorithme RL standard (Reinforce) doit être similaire avec un autre algorithme standard (DQN). On constate sur le graphique des récompenses obtenues durant la phase d'apprentissage sur 2000 parties que l'agent n'arrive pas à trouver une politique satisfaisante et la récompense moyenne se situe autour de 1. La première partie, où l'on voit un pic à 50 et plus, correspond seulement à la phase de remplissage de l'expérience replay avec une politique "greedy" plus importante. La médiocrité des récompenses vient du fait que la politique propose systématiquement un déplacement de pièces qui n'a pas de sens. Cela laisse donc le champ libre à l'adversaire pour capturer ses pièces et gagner la partie simplement par domination numérique. Je rappelle que l'adversaire dans ce cas est un simple agent aléatoire (mais qui produit des déplacements dans les règles de l'art).

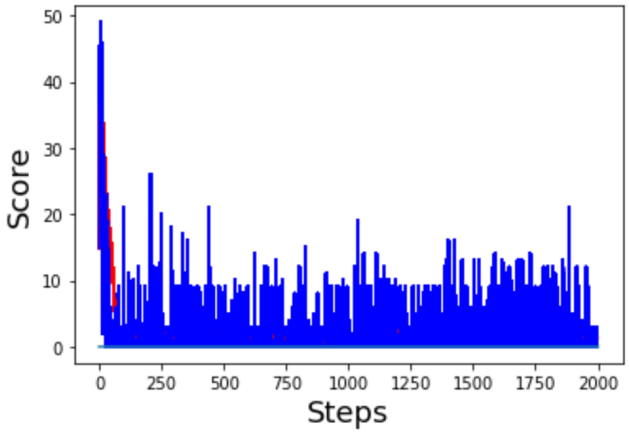


Fig 11 : courbe d'apprentissage agent DQN

L'implantation des mécanismes RIG dans mon agent DQN consiste à modifier l'implémentation de l'expérience replay pour ajouter la mémorisation des buts latents, à modifier le calcul de la récompense à partir des états latents et des buts latents à atteindre, puis modifier la fonction de valeur (le réseau de neurones qui approxime la Q-fonction) pour prendre en compte les états et les buts latents. Il convient en outre d'ajouter le générateur β-VAE pré-entraîné dans la boucle principale d'apprentissage de l'agent DQN pour encoder les états ainsi qu'un générateur aléatoire N(0,1) pour échantillonner des buts à atteindre. Je souligne à nouveau que les états et les buts latents sont des configurations de l'échiquier représentés dans l'espace latent du VAE.

La modification apportée à la fonction de valeur pour implémenter RIG consiste à passer d'une version classique $Q_\omega(s,a)$ à $Q_\omega(s,a,g)$où le but est intégré dans l'évaluation de la valeur. Afin de réaliser ceci, je me suis inspiré de ce que j'ai trouvé dans les codes liés à l'article et j'ai concaténé $s|q$ en entrée du modèle MLP (Q-fonction) du DQN. Je constate que c'est la même astuce technique utilisée pour effectuer le conditionnement dans les modèles génératifs.

J'ai implémenté une première version d'un agent DQN RIG sans ajouter le bruit (à la ligne 8 de l'algorithme RIG[16]), ni l'ajout de nouveaux buts à atteindre issus des configurations atteintes durant la dernière partie (lignes 17 à 23) et sans le fine-tuning du VAE non plus. C'était plus pour avoir une première validation du fonctionnement de mon code. Malheureusement, on constate à nouveau sur le graphique des récompenses obtenues durant la phase d'apprentissage sur 2000 parties (Fig. 12) que l'agent n'arrive toujours pas à trouver une politique satisfaisante et la récompense moyenne se situe cette fois-ci autour de 1.5. La raison semble toujours être la même : les déplacements proposés ne respectent pas les règles des échecs. Je serais tenté de mettre un contrôle à ce niveau mais je crains de pervertir l'esprit de l'apprentissage par renforcement où ce serait à l'agent de découvrir les bons déplacements.

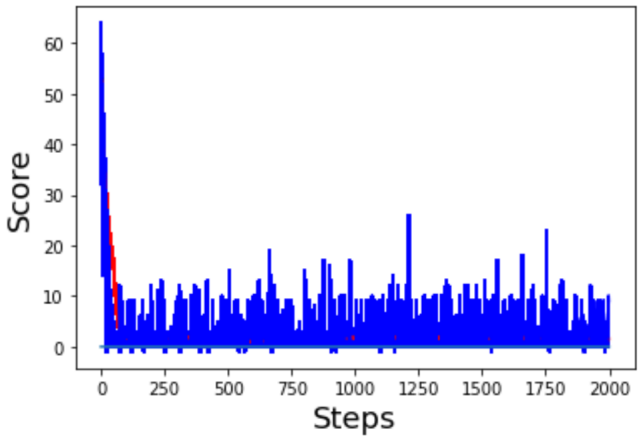


Fig 12 : courbe d'apprentissage agent DQN RIG version 1

Enfin pour tester l'agent DQN RIG en mode production, après l'entraînement, j'ai fixé comme objectif une configuration gagnante pour les blancs que j'avais obtenue lors de mes tests avec le moteur stockfish contre lui-même. J'ai ensuite encodé cette configuration avec le VAE et je l'ai fournie à l'agent en mode évaluation. C'est donc très simple d'utiliser un agent RIG en production : il suffit de lui donner des objectifs à atteindre sous la forme d'un code latent.

Après de nombreux tests et d'adaptations de code, je conclue que mon approche DQN RIG ne permet pas d'apporter de substantiels gains aux résultats attendus. Je note que le DQN classique montrait déjà la voie sans issue que j'essayais malgré tout de continuer à prendre. Je pense que je me suis laissé enfermé dans mes convictions. C'est le paradoxe des coûts irrécupérables : après tous les efforts engagés dans une voie, il est très difficile, pour un humain, de sortir de cette voie malgré l'évidence des faits montrant la non pertinence de poursuivre et de changer de voie. Je soupçonne que la représentation des configurations de l'échiquier est pour beaucoup dans ces échecs : page 14, j'écrivais que "je pense que cela aura un impact sur la stabilité et la qualité de l'apprentissage". Je ne m'étais pas trompé sur ce point.

[16] Voir la Fig 1. chapitre 2.5

5 - Conclusion

Premièrement, je vais revenir sur mes objectifs initiaux et les commenter. Concernant la compréhension du modèle RIG, je pense avoir atteint cet objectif tant sur l'aspect théorique que sur l'aspect pratique en produisant, à partir de codes existants, un agent RIG. J'ai pu également, sans la documenter dans ce rapport, faire une courte revue des approches IA dans le domaine des jeux d'échecs. Ce qui m'a permis de comprendre assez rapidement que mon ambition de battre stockfish allait rapidement être vaine. J'ai eu l'occasion par contre de pouvoir tester différentes approches RL pour jouer contre un agent de qualité professionnelle tel que "stockfish" et les comparer entre elles. J'ai atteint également mon objectif d'expérimenter les modèles génératifs comme substitut aux simulations de parties réelles pour générer des situations d'apprentissage. Cela m'a valu plusieurs interrogations et nouvelles pistes à explorer mentionnées dans le chapitre 4.1.

J'ai implémenté l'approche RIG, entraîné un agent afin de tenter de battre le moteur stockfish et j'ai confirmé mon échec en évoquant la raison principale : le paradoxe des coûts irrécupérables. Je retiens que le choix de la représentation des configurations ainsi que la modélisation d'un problème sont déterminants. Je n'aurai pas eu l'occasion de vérifier si l'approche RIG fonctionne mieux que d'autres.

D'une manière générale, ce projet m'a surtout permis, outre de renforcer mes compétences avec la programmation Python et quelques librairies pour IA, de mettre en pratique la plupart des concepts vus en cours. Une difficulté technique que j'ai rencontrée, et qui semble être partagée par tous les praticiens du domaine, est le temps que les apprentissages prennent pour s'exécuter. Sans doute, les codes ne sont pas optimaux , avec notamment des aller-retours entre GPU et CPU, mais c'est vraiment long et laborieux pour valider des résultats. J'ai pu profiter pourtant d'un compte Google Colab Pro pour effectuer mon projet. Enfin, la part de programmation est non négligeable dans tout projet IA, même si les frameworks sont très pratiques. Je n'ose même pas imaginer les efforts considérables qu'ont dû déployer les pionniers du domaine.

Enfin, je remercie le corps professoral du cours RCP 211 de m'avoir permis de plonger dans cet univers passionnant du RL et des modèles génératifs.

6 - Codes sources du projet

L'ensemble des codes que j'ai produit, adapté ou copié pour la production des résultats présentés dans ce rapport sont disponibles en ligne à cet endroit :

https://github.com/delemarchand2020/DeepLearning/blob/main/Projet_G%C3%A9n%C3%A9ration_de_sc%C3%A9narios_et_renforcement_Denis_Lemarchand.ipynb

7 - Références utilisées dans le cadre de ce projet

Articles scientifiques

[1] Ashvin Nair, Vitchyr Pong, Murtaza Dalal, Shikhar Bahl, Steven Lin, and Sergey Levine. Visual reinforcement learning with imagined goals. arXiv preprint arXiv:1807.04742, 2018.

[2] Cao, S.; Li, J.; Nelson, K.P.; Kon, M.A. Coupled VAE: Improved Accuracy and Robustness of a Variational Autoencoder. Entropy 2022, 24, 423. https://doi.org/10.3390/e24030423

[3] Victor Berger, Michèle Sebag. Variational Auto-Encoder: not all failures are equal. 2020. ffhal02497248f

[4] Scott Fujimoto, Herke van Hoof, David Meger : Addressing Function Approximation Error in Actor-Critic Methods. arXiv:1802.09477v3, 2018

[5] Guillaume Matheron, Nicolas Perrin, Olivier Sigaud : The problem with DDPG: understanding failures in deterministic environments with sparse rewards. arXiv:1911.11679v1, 2020

Apprentissage par renforcement et modèles génératifs

Tous les supports et TP du cours RCP 211 seconde session 2021/2022:
https://cedric.cnam.fr/vertigo/cours/RCP211/

Le jeu d'échec

Documents, librairies et moteurs pour comprendre et simuler des parties d'échec:
https://fr.wikipedia.org/wiki/Pi%C3%A8ce_(%C3%A9checs)
https://www.chess.com/terms/chess-piece-value#Chesspiecevals
https://python-chess.readthedocs.io/en/latest/
https://pypi.org/project/stockfish/
https://fr.wikipedia.org/wiki/Stockfish_(programme_d'%C3%A9checs)

Librairie Reinforcement Learning Chess

Librairie d'agents RL avec le framework python-chess:
https://github.com/arjangroen/RLC
https://www.kaggle.com/code/arjanso/reinforcement-learning-chess-4-policy-gradients/notebook

Création d'un robot de jeu d'échecs en DRL:
https://towardsdatascience.com/hacking-chess-with-decision-making-deep-reinforcement-learning-173ed32cf503

Implémentation RIG

Code :
https://github.com/vitchyr/rlkit

https://github.com/rail-berkeley/rlkit
https://github.com/vitchyr/multiworld

Vidéos :
https://sites.google.com/site/visualrlwithimaginedgoals/

Implémentation TD3

Code et explications pour implémenter TD3 :
https://towardsdatascience.com/td3-learning-to-run-with-ai-40dfc512f93

TD3 prêt à utiliser (librairie stable-baselines):
https://stable-baselines.readthedocs.io/en/master/modules/td3.html

Environnement Gym :
https://www.gymlibrary.ml/

Exemple de création d'un environnement Gym pour un contexte de trading :
https://towardsdatascience.com/creating-a-custom-openai-gym-environment-for-stock-trading-be532be3910e