Apprentissage par renforcement du jeu de Hex

1 Résumé

Dans ce document, je vous présente une approche possible pour développer une IA pour le jeu de Hex en utilisant l'apprentissage par renforcement. Je vous propose d'utiliser la version la plus simple des méthodes de type "actor-critic". La stratégie π ainsi que la fonction valeur $\hat{v}$ seront paramétrées à l'aide de réseaux de neurones "simples" de type MLP et éventuellement RNN ou CNN (tous ces termes sont expliqués dans le document).

Pour des informations générales sur le jeu de Hex, vous pouvez lire la page wikipédia directement : `https://fr.wikipedia.org/wiki/Hex`.

Dans la section 2, je vous propose un plan de développement pour le projet.

Dans la section 3, je vous présente brièvement le modèle "canonique" d'un problème d'apprentissage par renforcement. Ce modèle est appelé "Markov Decision Process" (MDP). Je fais le lien entre le modèle général et l'exemple qui nous intéresse, à savoir le jeu de Hex.

Dans la section 4, je vous présente l'algorithme actor-critic. Je me contente de décrire les différents paramètres et variables qui interviennent dans l'algorithme, mais je n'explique pas les arguments "théoriques" qui expliquent pourquoi la méthode fonctionne. Si cela vous intéresse, voir le livre de Barto et Sutton [1], en particulier le chapitre 13.

Dans la section 5, je présente très brièvement les réseaux de neurones et je propose une façon de modéliser la stratégie π et la fonction valeur $\hat{v}$ pour le jeu de Hex. J'indique des pistes pour l'implémentation, notamment les bibliothèques possibles (Keras).

2 Plan de développement

L'objectif de ce projet de programmation en Python est d'implémenter une IA pour le jeu de Hex et de concevoir une interface permettant à un joueur humain de jouer contre l'IA. Je vous propose de planifier un point régulier d'un quart d'heure chaque semaine, ou éventuellement toutes les deux semaines selon le nombre d'heures d'autonomie dont vous disposez. On peut diviser le projet en trois grandes étapes :

1. Implémentation du jeu et de l'IHM
2. *[Optionnel]* Première IA "simple" : algo de type minmax, alpha-bêta, MCTS etc.
3. Seconde IA : apprentissage par renforcement [1].

On pourra tenter l'étape 2 si l'étape 1 est finie suffisamment rapidement, ou si l'apprentissage par renforcement ne fonctionne pas bien et qu'il nous reste du temps en Janvier. Les autres sections de ce document ne traitent que d'apprentissage par renforcement (étape 3).

Voici quelques idées/"cahier des charges" pour l'étape 1 (qu'on pourra revoir à la hausse ou à la baisse selon la vitesse de développement et les difficultés rencontrées).

- Jouer à deux joueurs humains.
- Jouer contre l'IA.
- Quand la condition de victoire est vérifiée (deux bords sont reliés), afficher la couleur gagnante. On ne doit plus pouvoir poser de pion supplémentaire. Question : comment vérifier efficacement la condition de victoire ?

- Un bouton pour effacer le plateau et lancer une nouvelle partie.
- Si les cases hexagonales sont trop difficiles à gérer, on pourra commencer par des cases carrées.
- Changer la taille du plateau (sinon, on jouera sur le 13×13 ou 11×11).
- Pouvoir naviguer parmi les coups déjà joués.
- Implémenter la règle du gâteau avec un bouton pour choisir si elle est activée ou pas.
- Un bouton pour afficher/sauvegarder les coups joués. Un bouton pour charger une liste de coups enregistrés dans un fichier.

3 Markov Decision Process

Un MDP est entièrement défini par les éléments suivants :

- Un ensemble d'états $\mathcal{S}$.
- Un ensemble d'actions $\mathcal{A}_s$ pour chaque état $s \in \mathcal{S}$.
- Un ensemble de récompenses $\mathcal{R}_{s,a}$ pour chaque couple d'état-action $(s, a) \in \mathcal{S} \times \mathcal{A}_s$.
- Une fonction de transition $p : \mathcal{S} \times \mathcal{R}_{s,a} \times \mathcal{S} \times \mathcal{A}_s \to [0, 1]$ telle que $p(s', r, s, a)$ est la probabilité d'arriver dans l'état s' et de recevoir la récompense r en partant de l'état s et en faisant l'action a. On peut aussi la noter $p(s', r|s, a)$ et dire qu'il s'agit de la probabilité d'avoir s' et r sachant qu'on avait s et a. Cette notation ne doit pas nous faire oublier que p est simplement une fonction à quatre variables qui est donc parfaitement déterministe.

A partir de la fonction de transition p, on peut obtenir la probabilité $q(s'|s, a)$ d'arriver dans l'état s' sachant qu'on était dans l'état s et qu'on a choisit l'action a. On a $q(s'|s, a) = \sum_{r \in \mathcal{R}_{s,a}} p(s', r|s, a)$.

Une *stratégie* (aussi appelé une *politique*) $\pi : \mathcal{A}_s \times \mathcal{S} \to [0, 1]$ est une fonction telle que $\pi(a, s)$ soit la probabilité de choisir l'action a sachant qu'on est dans l'état s. On peut aussi la noter $\pi(a|s)$.

Supposons que nous disposons d'une stratégie π. Partant d'un état initial s_0, nous tirons une action A_0 selon la loi de probabilité définie par $\pi(.|s_0)$. Remarquez que je note en majuscule les variables aléatoires alors que les valeurs déterministes sont notées en minuscule, autrement dit A_0 est une variable aléatoire alors que s_0 est déterministe (dans le cas général, l'état initial peut aussi être tiré aléatoirement selon une distribution, mais dans notre cas l'état initial sera toujours le plateau vide). Étant donné notre état s_0 et notre action A_0, le MDP nous fournit une récompense R_1 et l'état suivant S_1 d'après la loi de probabilité définie par $p(., .|s_0, A_0)$. Ensuite on utilise la loi $\pi(.|S_1)$ pour obtenir une action A_1, puis la loi $p(., .|S_1, A_1)$ nous donne une récompense R_2 et un état S_2, et ainsi de suite ...

Dans le cas d'un jeu comme Hex, il existe un sous-ensemble *d'états finaux* noté $\mathcal{S}_F \subset \mathcal{S}$. Plus précisément, dans le cas du Hex, les deux états finaux sont *victoire* et *défaite* (les parties nulles sont impossibles). Une fois qu'on arrive dans un état final, le MDP s'arrête.

La séquence d'états, actions, récompenses, états suivants etc., que l'on peut noter

$$s_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_t, A_t, R_{t+1}, S_{t+1}, \ldots, S_{T_F-1}, A_{T_F-1}, R_{T_F}, S_{T_F}$$

entre un état initial et un état final est appelé un *épisode*.

On appelle *gain* la somme pondérée des récompenses sur un épisode

$$G = \sum_{t=1}^{T_F} \gamma^t R_t$$

où $\gamma \in [0, 1]$ est un *taux d'actualisation* permettant de donner plus d'importance aux récompenses immédiates et moins aux récompenses plus éloignées dans le temps. Comme on fonctionne par épisode ici, on pourra prendre $\gamma = 1$. Notez que, comme les R_t sont des variables aléatoires, G est aussi une variable aléatoire.

On note $v_\pi(s) = \mathrm{E}_\pi[G|s_0 = s]$ l'espérance du gain lorsque les actions sont choisies avec la stratégie π et lorsque l'état initial est s. v_π est la *fonction valeur pour la stratégie* π. Il s'agit de notre critère, autrement dit notre but est de trouver une stratégie π_* telle que

$$\pi_* = \arg\max_\pi v_\pi(s).$$

où s est notre état initial (le plateau vide). Notez qu'en général la stratégie π_* n'est pas unique.

En pratique, la fonction de transition p du MDP est inconnue, ou l'espérance est trop coûteuse à évaluer, donc on ne peut pas calculer $v_\pi(s)$ directement. Il existe une très grande quantité de méthodes permettant d'adresser ce problème en fournissant un algorithme qui converge vers $v_{\pi_*}(s)$ ou directement vers π_* (ou a minima qui converge vers une stratégie suffisamment bonne). On verra un exemple d'une telle méthode dans la section 4.

Dans notre cas, le MDP sera défini ainsi :

- Un état $s \in \mathcal{S}$ sera une *disposition du plateau*, c'est à dire la position de l'ensemble des pions.
- Les actions $\mathcal{A}_s$ sont l'ensemble des cases vides dans l'état actuel s, c'est à dire l'ensemble des cases où on peut jouer.
- Pour les récompenses, le plus simple est de donner une récompense nulle à chaque transition qui n'est pas finale. On donne ensuite une récompense positive si on atteint l'état *victoire* (par exemple $+1$) et négative si on atteint l'état *défaite* (par exemple -1). Si cela ne marche pas bien, on pourra réfléchir à d'autres façons de distribuer les récompenses.
- Étant donné l'état actuel et l'action choisie par notre stratégie, l'état suivant sera la position du plateau *après la réponse de l'adversaire* (sauf si c'est un état final). La fonction de transition dépend donc de la stratégie de l'adversaire. En pratique, on entraînera l'IA en la faisant jouer contre elle-même donc la stratégie de l'adversaire sera la même que la stratégie que l'on est en train d'apprendre.

4 Actor-Critic

La méthode *actor-critic* est une méthode de type *gradient*. L'idée est de *paramétrer* l'ensemble des stratégies que l'on est susceptible d'apprendre. On suppose donc que l'on dispose d'une *famille paramétrée de stratégies* $\pi(a|s,\theta)$ où θ est un vecteur de paramètres. Autrement dit, pour chaque θ_0, on a une stratégie différente $\pi(a|s,\theta_0)$. Comme on va devoir calculer le gradient par rapport à θ, il faut que $\pi(a|s,\theta)$ soit différentiable par rapport à θ. On peut paramétrer les stratégies de plein de manières différentes. Dans notre cas la stratégie sera donnée par un réseau de neurones et $\theta \in \mathbb{R}^n$ sera simplement le vecteur des poids du réseau de neurones.

On aura aussi besoin d'une famille paramétrée de fonctions valeurs $\hat{v}(s,w)$ qui aura pour but d'approximer $v_\pi(s)$. Ici aussi, $\hat{v}(s,w)$ doit être différentiable par rapport à w. On utilisera aussi un réseau de neurones pour paramétrer $\hat{v}(s,w)$ avec $w \in \mathbb{R}^m$ le vecteur des poids du réseau de neurones.

La stratégie $\pi(a|s,\theta)$ est appelée l'*acteur* et la fonction valeur $\hat{v}(s,w)$ est appelé le *critique* d'où le nom de l'algorithme.

La méthode actor-critic est présentée dans l'algorithme 1. Il fait appel à plusieurs méthodes et astuces qu'il n'est pas nécessaire de comprendre pour implémenter l'algorithme (notamment, je n'ai pas parlé des traces d'éligibilité). Tout est expliqué dans le livre de Barto et Sutton [1].

Concernant le choix des hyperparamètres, il faudra tester plusieurs valeurs. Pour λ^θ et λ^w, je vous propose de partir de 0.5 puis d'augmenter et diminuer cette valeur par pas de 0.1 (ou 0.25 si vous n'avez plus le temps). Pour les taux d'apprentissage α^θ et α^w, je vous propose de partir de 10^{-6} puis d'augmenter et de diminuer cette valeur en multipliant ou divisant par des puissances de 10 (il se peut que la meilleure valeur soit différente pour α^θ et pour α^w). Inutile d'aller plus haut que 10^{-1} ou plus bas que 10^{-14}. Pour γ, on prendra $\gamma = 1$. Pour le nombre d'épisodes N, j'avoue ne pas avoir d'ordre de grandeur précis en tête (donc à chercher) mais peut-être commencer par des valeurs pas trop élevées pour voir comment l'algorithme se comporte (disons 1000 ou 10000 épisodes) puis monter à 10^6 ou 10^7 ou plus si les temps de calcul ne sont pas trop grands.

Les paramètres les plus importants sont α^θ, α^w et N. Dans un premier temps, on pourra fixer les autres à leur valeur par défaut et se concentrer sur ces trois là.

Algorithme 1: Actor-Critic

Entrées :
Une famille paramétrée de stratégies $\pi(a|s,\theta)$
Une famille paramétrée de fonctions valeurs $\hat{v}(s,w)$
Sorties :
les paramètres θ et w optimaux
Hyperparamètres :
Taux de décroissance des traces $\lambda^\theta \in [0,1]$ et $\lambda^w \in [0,1]$
Taux d'apprentissage $\alpha^\theta > 0$ et $\alpha^w > 0$
Taux d'actualisation $\gamma \in [0,1]$ et nombre d'épisodes N

Initialiser les paramètres $\theta \in \mathbb{R}^n$ et $w \in \mathbb{R}^m$ (e.g., initialisations de He ou de Xavier[a])
Pour i **allant de** 1 **à** N:
Initialiser l'état initial : $S = s_0$
Initialiser les vecteurs de trace d'éligibilité : $z^\theta = 0$ et $z^w = 0$ (avec $z \in \mathbb{R}^n$ et $z \in \mathbb{R}^m$)
$I = 1$
Tant que S n'est pas un état final:
$A \sim \pi(.|S,\theta)$
Faire l'action A et observer l'état suivant S' et la récompense R
Si S' n'est pas final:
$\delta = R + \gamma\hat{v}(S',w) - \hat{v}(S,w)$
Sinon:
$\delta = R - \hat{v}(S,w)$
$z^w = \gamma\lambda^w z^w + \nabla\hat{v}(S,w)$
$z^\theta = \gamma\lambda^\theta z^\theta + I\nabla \ln \pi(A|S,\theta)$
$w = w + \alpha^w \delta z^w$
$\theta = \theta + \alpha^\theta \delta z^\theta$
$I = \gamma I$
$S = S'$

[a] https://machinelearningmastery.com/weight-initialization-for-deep-learning-neural-networks/

5 Neural Networks

5.1 Multilayer Perceptron

Un réseau de neurones est un modèle paramétrique fonctionnant par "couches". Je présente ici le réseau de neurones le plus simple appelé Multilayer Perceptron (MLP). Si ce n'est pas clair, on pourra en discuter avec un tableau. Le MLP est composé de p couches ayant toutes la même structure (mais des tailles différentes). La k-ième couche est décrite par l'équation suivante :

$$x_k = \phi(p_k x_{k-1} + b_k),$$

où $p_k \in \mathcal{M}(n_k, n_{k-1})$ est la matrice à n_k lignes et n_{k-1} colonnes des poids de la k-ième couche, $b_k \in \mathbb{R}^k$ est le vecteur des biais de la k-ième couche, $x_k \in \mathbb{R}^{n_k}$ est la sortie de la k-ième couche, $x_{k-1} \in \mathbb{R}^{n_{k-1}}$ est l'entrée de la k-ième couche, ϕ est une fonction d'activation permettant au réseau de neurones d'être non linéaire et donc de pouvoir approximer n'importe quelle fonction (si les couches sont suffisamment larges et nombreuses).

La fonction d'activation la plus utilisée est la fonction ReLU : $\mathbb{R} \to \mathbb{R}$ (Rectified Linear Unit)

$$\mathrm{ReLU}(x) = \max(x, 0),$$

qui est appliquée composante par composante. Pour la dernière couche de la stratégie π (voir plus bas) on utilisera la fonction d'activation softmax notée $\sigma : \mathbb{R}^{n_p} \to \mathbb{R}^{n_p}$

$$\sigma(x) = \left[\frac{\exp(x[1])}{\sum_{j=1}^{n_p} \exp(x[j])} \quad \cdots \quad \frac{\exp(x[i])}{\sum_{j=1}^{n_p} \exp(x[j])} \quad \cdots \quad \frac{\exp(x[n_p])}{\sum_{j=1}^{n_p} \exp(x[j])} \right]^T,$$

où $x[i]$ est la i-ème composante du vecteur $x \in \mathbb{R}^{n_p}$. La fonction softmax permet d'obtenir une distribution de probabilité. En effet chaque composante de $\sigma(x)$ est entre 0 et 1, et la somme des composantes de $\sigma(x)$ est égale à 1. Pour la dernière couche de la fonction valeur $\hat{v}$, on n'utilisera *pas* de fonction d'activation (autrement dit on prendra la fonction d'activation identité, aussi appelée activation linéaire). En effet, la fonction valeur peut prendre n'importe quelle valeur dans $\mathbb{R}$, alors que la fonction ReLU est à valeurs dans $\mathbb{R}^+$.

Étant donné que les poids et les biais jouent des rôles similaires, on va les regrouper dans une seule matrice qu'on appellera aussi matrice des poids de la k-ième couche et que l'on notera ω_k. On a donc $\omega_k = [p_k, b_k]$ où la notation entre crochets signifie que l'on a concaténé p_k et n_k ou autrement dit, on a ajouté la colonne b_k à la matrice p_k. Ainsi, $\omega_k \in \mathcal{M}(n_k, n_{k-1} + 1)$. La k-ième couche est alors décrite par :

$$x_k = \phi(\omega_k [x_{k-1}^T, 1]^T),$$

où la notation $[x_{k-1}^T, 1]^T$ signifie qu'on a ajouté un 1 à la fin du vecteur x_{k-1} d'où $[x_{k-1}^T, 1]^T \in \mathbb{R}^{n_{k-1}+1}$.

Si on note f_k la fonction correspondant à la couche k, i.e., $x_k = f_k(x_{k-1})$, un MLP est simplement la composition des fonctions de toutes ses couches :

$$\text{MLP} = f_p \circ \cdots \circ f_{k+1} \circ f_k \circ \cdots \circ f_1.$$

On a donc $\text{MLP}(x_0) = x_p$. Dans la suite on notera x à la place de x_0 et y à la place de x_p, c'est à dire $\text{MLP}(x) = y$.

Notons ω le vecteur de tous les poids de notre MLP. Autrement dit, on aplatit chaque matrice ω_k puis on les concatène pour obtenir le vecteur ω. Ce vecteur ω correspond aux paramètres θ ou w apparaissant dans la méthode actor-critic de la section 4. Afin de montrer que la sortie du réseau dépend des poids ω, on peut noter $\text{MLP}(x, \omega) = y$. Dans la méthode actor-critic, on a besoin de calculer des gradients $\nabla \text{MLP}(x, \omega)$ par rapport à ω. Cela est fait via l'algorithme de backpropagation qui est déjà implémenté dans toutes les librairies de machine learning (comme TensorFlow, Keras, Pytorch etc.). En pratique vous n'avez qu'à définir votre réseau de neurones via l'API de la bibliothèque puis, toujours via l'API, vous pouvez demander le gradient en n'importe quel point.

Dans un second temps, on pourra éventuellement utiliser des réseaux de neurones plus sophistiqués comme les réseaux de neurones récurrents (RNN) ou les réseaux de neurones convolutifs (CNN).

5.2 Construction de l'acteur et du critique

Pour définir $\hat{v}(s, w)$, on aura simplement $\hat{v}(s, w) = \text{MLP}_{\hat{v}}(s, w)$.

Comment définir $\pi(a|s, \theta)$ à l'aide d'un MLP ? Une première idée serait de construire un réseau prenant en entrée l'état courant (la disposition du plateau) et fournissant en sortie un vecteur de probabilités dont chaque composante correspond à une case du plateau (vide ou pas). A partir de ce vecteur, vous pouvez tirer aléatoirement l'action à jouer. Il est important de tirer aléatoirement l'action, et pas de choisir celle ayant la plus haute probabilité, car il se peut que la stratégie optimale ne soit *pas* déterministe !

Bien sûr, les cases déjà occupées par un pion ne peuvent pas être jouées. Pour éviter de tirer une action injouable, je vous propose de ne garder que les cases jouables est de réappliquer la fonction softmax sur ce sous-ensemble de cases jouables afin d'obtenir une nouvelle distribution de probabilités qui ne portera que sur les actions jouables. Peut-être vous demandez vous pourquoi ne pas créer un réseau ne fournissant que les actions jouables ? Car un réseau de neurones doit avoir un sortie de taille fixe, or le nombre d'actions jouables dépend de l'état courant.

Attention, dans la méthode actor-critic le gradient qui nous intéresse est $\nabla \ln \pi(A|S, \theta)$ par rapport à θ. Une fois l'action A tirée, il faut donc encore composer votre vecteur de sortie avec une projection ne conservant que la composante correspondant à l'action A, puis avec la fonction ln, avant de calculer le gradient !

Pour l'implémentation, je vous propose d'utiliser Keras ou Pytorch, avec une préférence pour Keras qui est plus simple à utiliser que Pytorch. Keras est construit par-dessus TensorFlow (qui est plus dur à utiliser). Je pourrai vous fournir un squelette de code en Keras implémentant la méthode actor-critic. Vous devrez compléter ce code afin de le connecter à votre environnement (qui n'est autre que le jeu que vous aurez codé).

5.3 Technique d'apprentissage

Dans notre cas, l'algorithme va jouer contre lui-même pour apprendre. Plutôt que d'avoir deux modèles (un pour chaque adversaire), je pense que le mieux est d'avoir un unique modèle utilisé (et mis à jour) par les deux adversaires en même temps. Cela va sûrement accélérer l'apprentissage et éviter d'avoir à se demander comment combiner les deux modèles après l'entraînement.

Concrètement, appelons Blanc et Noir les deux adversaires et notons S_t^B l'état du plateau à l'instant t du point de vue de Blanc (i.e., c'est à Blanc de jouer). Blanc utilise le modèle pour choisir une action A_t^B puis il joue cette action. On peut alors déterminer l'état S_t^N du point de vue de Noir (i.e., c'est à Noir de jouer). Noir reçoit aussi sa récompense R_t^N (qui sera égale à 0 si la partie n'est pas finie). Noir utilise R_t^N et S_t^N pour mettre à jour les poids θ et w. Puis Noir utilise le même modèle (qu'il vient de mettre à jour) pour choisir une action A_t^N puis il joue cette action. Blanc peut alors observer son état suivant S_{t+1}^B et recevoir sa récompense R_{t+1}^B. Blanc met alors à jour les poids θ et w. Et ainsi de suite jusqu'à la fin de la partie. Puis on enchaîne sur la partie suivante (épisode suivant) jusqu'à avoir fait N parties.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning*. The MIT Press, 2018.