Entrainement d'une intelligence artificielle sur un jeu

1 Introduction

Le concept de Snake a vu le jour au début des années 1970. L'une des premières versions connues a été développée sous le nom de "Blockade" en 1976 par Gremlin Industries. Cependant, c'est la version intégrée aux téléphones Nokia dans les années 1990 qui a véritablement popularisé le jeu.

Le principe de Snake est simple et addictif :

- **Objectif** : Contrôler un serpent qui se déplace sur l'écran pour manger des objets (souvent représentés par des points ou des fruits) qui apparaissent aléatoirement. Chaque fois que le serpent mange un objet, il grandit ;
- **Déplacement** : Le serpent se déplace continuellement dans une direction (haut, bas, gauche, droite). Le joueur peut changer la direction du serpent, mais ne peut pas le stopper ;
- **Règles** : Le jeu se termine si le serpent touche les bords de l'écran ou s'il se mord la queue.

Notre but est d'entrainer une intelligence artificielle à jouer à ce jeu. On pourrait facilement concevoir un algorithme jouant parfaitement en retranscrivant notre logique sous forme de code, mais le but ici est d'utiliser le machine learning : ce sera à notre algorithme de trouver lui-même les stratégies pour gagner. On utilisera pour cela des réseaux de neurones. Cependant, on se détache du principe de classification que l'on connaît pour se rapprocher du self-learning : on ne dispose plus d'une fonction dérivable à minimiser, et on ne peut donc plus utiliser le principe de descente de gradient pour entrainer notre réseau. On va donc se tourner vers les algorithmes génétiques, et faire évoluer une population de réseaux de neurones afin d'obtenir des agents de plus en plus performants au fil des générations.

2 Basecode

Vous disposez d'un moteur de jeu de Snake, implémenté au sein de la classe Game du fichier snake.py, avec les méthodes suivantes :

- le constructeur permet de déclarer une nouvelle partie au sein d'une grille dont les dimensions sont données en paramètres ;
- la méthode setFruit permet de placer un fruit dans une case vide de la grille, de manière aléatoire ;
- la méthode refresh applique le prochain mouvement du serpent, et gère les cas de fin de partie (collision, cyclage) et d'ingestion du fruit, en mettant à jour les attributs correspondants (nombre de fruits mangés, taille du serpent, ...) ;

- la méthode print permet d'afficher la grille dans la console ;
- la méthode getFeatures traduit la grille sous la forme d'un vecteur d'informations qui sera décrit plus loin.

Un objet de la classe Game sera également décrit par plusieurs attributs :

- grille est une matrice d'entiers, chaque case pouvant être à 0 (vide), 1 (serpent) ou 2 (fruit) ;
- serpent est une liste contenant les positions de chacun des éléments du serpent ;
- direction est un entier pouvant être 0 (haut), 1 (bas), 2 (gauche) ou 3 (droite) ;
- enCours est un booléen à vrai par défaut, et mis à faux lorsque la partie est terminée ;
- steps est un entier contenant le nombre de pas depuis la dernière pomme mangée ;
- score est un entier contenant le nombre de pommes mangées.

Ensuite, vous avez à votre disposition une implémentation de réseaux de neurones en numpy, sans la procédure d'apprentissage qui ici ne servira à rien. Vous disposez également d'une classe SnakeVue permettant la création d'une fenêtre graphique, et l'affichage du jeu via la librairie pygame. Cela permettra de visualiser le comportement d'un agent après apprentissage. Enfin, un fichier main.py servant de lanceur vous est fourni, et suit la logique suivante :

1. importation des différentes librairies ;
2. déclaration de la fonction d'évaluation d'un réseau de neurones, à compléter ;
3. appel de la fonction d'entrainement renvoyant un réseau de neurones ;
4. affichage en boucle du réseau de neurones jouant au Snake.

3 Traduction de l'état de jeu en features

Votre premier travail sera d'implémenter la méthode getFeatures de la classe Game. Le but est de traduire l'état de la partie (une grille de $L \times H$ cases) en un vecteur de taille réduite, mais conservant suffisamment d'information pour prendre une décision. Vous implémentez votre fonction pour qu'elle renvoie un tableau numpy de 8 cases, contenant les informations suivantes :

1. Est-ce qu'il y a un obstacle directement au dessus de la tête du serpent (0 ou 1) ?
2. Est-ce qu'il y a un obstacle directement en dessous de la tête du serpent (0 ou 1) ?
3. Est-ce qu'il y a un obstacle directement à gauche de la tête du serpent (0 ou 1) ?
4. Est-ce qu'il y a un obstacle directement à droite de la tête du serpent (0 ou 1) ?
5. Est-ce que le fruit se trouve au dessus (1), en dessous (-1) ou sur la même ligne (0) que la tête du serpent ?
6. Est-ce que le fruit se trouve à droite (1), à gauche (-1) ou sur la même colonne (0) que la tête du serpent ?
7. Quelle est la direction du serpent (0, 1, 2 ou 3) ?
8. A quelle distance se trouve le bord, compte tenu de la direction actuelle ? On normalisera la distance pour que la valeur soit entre 0 et 1.

Pour la dernière caractéristique, on calcule la distance (en nombre de cases) entre la tête du serpent et le mur vers lequel on se dirige : si on se dirige vers le haut, on prend le mur du haut, ... Et on divise la valeur obtenue soit par la largeur, soit par la hauteur, en fonction de notre direction.

4 Comment faire jouer un réseau de neurones

On travaillera avec des réseaux ayant une couche d'entrée de 8 neurones et une couche de sortie à 4 neurones, ainsi qu'une couche cachée de 24 neurones. La couche d'entrée est définie par le nombre de features (ici 8), et la couche de sortie est définie par le nombre d'actions possibles (ici le nombre de directions). Une partie se déroulera de la manière suivante :

1. on initialise la partie ;
2. tant que la partie n'est pas terminée :
   a) on extrait les features de l'état de la partie ;
   b) on injecte les features dans la couche d'entrée du réseau de neurones, et on récupère sa prédiction ;
   c) la prédiction correspond à la direction à prendre (aller en haut, en bas, à gauche ou à droite), on actualise la direction en cours de la partie en conséquence ;
   d) on actualise la partie.

5 Algorithme génétique

Pour permettre d'entrainer un réseau de neurones sur ce jeu, on va adopter une stratégie évolutionnaire. Les algorithmes génétiques datent des années 60, même si la première publication scientifique à ce sujet paraît en 1975, du chercheur John Holland. Le principe est le suivant : on commence par générer aléatoirement des solutions, appelées individus. Ensuite, en fonction de leur évaluation, on sélectionne un certain nombre de ces individus que l'on organise en couples. Chaque couple génère un ou plusieurs enfants via un **croisement**, chaque enfant généré héritera de caractéristiques des deux parents. Ensuite, chaque enfant est succeptible de muter, des modifications aléatoires peuvent alors apparaître. Enfin, les enfants sont introduits dans la population, et celle-ci est réduite pour atteindre sa taille initiale : on ne garde que les meilleurs individus. On répète le processus de sélection, croisement, mutation et survie un certain nombre d'itérations (appelées également générations), pour renvoyer à la fin le meilleur individu généré.

Votre population est constituée d'individus, chaque individu étant caractérisé par :

1. un réseau de neurones, avec ses poids et ses biais ;
2. un score, déterminé par l'évaluation du réseau de neurones.

Si on suit la logique détaillée plus haut, on peut décomposer l'algorithme génétique de la manière suivante :

1. on commence par créer une liste de P individus, ce qui consiste à créer leur reseau de manière aléatoire, mais en suivant une architecture donnée (nombre et tailles des couches fixés). Ensuite, on évalue chacun des individus pour leur donner une score entre 0 et 1, qui sera à maximiser ;
2. a chaque itération, on suit le processus suivant :
   a) on sélectionne les S meilleurs individus parmi la population et on élimine les $P - S$ pires individus ;
   b) on créé, via croisement et mutation puis évaluation, $P - S$ nouveaux individus qui sont intégrés à la population, qui contient à nouveau P individus ;
3. on renvoie le meilleur individu de la population.

5.1 Fonction d'évaluation

Jusqu'à présent, dans ce cours, on a utilisé la précision ou l'erreur pour l'évaluation de nos méthodes. L'erreur en particulier avait pour avantage d'être une fonction dérivable, nous permettant d'utiliser le processus de la descente de gradient pour améliorer un réseau de neurones. Ceci ne sera pas possible ici, on va donc procéder autrement : pour évaluer un réseau de neurones, on va lui permettre de jouer un certain de parties et on va se baser sur le nombre de pommes mangées et le nombre de pas effectués avant de mourir. Soient N le nombre de parties jouées, H et W respectivement la hauteur et la largeur de la grille, p_i et s_i respectivement le nombre de pommes mangées et le nombre de pas effectués entre la dernière pomme et le gameover lors de la partie i, on définit le score du réseau de neurones par :

$$score = \frac{1}{N \times H \times W \times 1000} \sum_{i=1}^{N} (1000 \times p_i + s_i)$$

Ainsi, le nombre de pommes aura un impact très important sur le score, et le nombre de pas avant de mourir viendra ajouter une petite quantité permettant de valoriser la survie. Votre fonction d'évaluation consistera donc à faire jouer N parties à un réseau de neurones, en mettant à jour, à la fin de chaque partie, le score qui sera renvoyé. Si un individu obtient un score de 1, alors on considérera qu'il est parfait (il mange le maximum de pommes possible dans les parties jouées).

5.2 Croisement

Un couple est défini par deux individus de la population considérés comme étant de bonne qualité. On composera les couples de manière aléatoire : on sélectionne au hasard deux individus parmi les S meilleurs de la population. On va décrire ici une procédure permettant de générer deux enfants à partir de deux parents. Premièrement, on remarque que les deux parents et les deux enfants partagent tous la même architecture : ils ont le même nombre de couches et de neurones. On procédera alors de la manière suivante :

1. on commence par tirer au hasard un nombre entre 0 et 1 : si ce nombre est supérieur au paramètre pc, alors le premier enfant est le clone du premier parent, et le deuxième enfant est le clone du deuxième parent, et on quitte la fonction ;
2. sinon, on suit le processus suivant pour chaque couche.
   a) on tire un nombre α entre 0 et 1 ;
   b) les poids des arcs entrants sur un neurone j de la couche courante sont réglés de la manière suivante :

$$w_{i,j}(c1) = \alpha \times w_{i,j}(p1) + (1 - \alpha) \times w_{i,j}(p2)$$
$$w_{i,j}(c2) = (1 - \alpha) \times w_{i,j}(p1) + \alpha \times w_{i,j}(p2)$$

   c) on utilise le même principe pour régler les biais du neurone j :

$$b_j(c1) = \alpha \times b_j(p1) + (1 - \alpha) \times b_j(p2)$$
$$b_j(c2) = (1 - \alpha) \times b_j(p1) + \alpha \times b_j(p2)$$

Avec $w_{i,j}(c1)$, $w_{i,j}(c2)$, $w_{i,j}(p1)$ et $w_{i,j}(p2)$ représentant les poids d'un arc entrant sur le neurone j, et $b_j(c1)$, $b_j(c2)$, $b_j(p1)$ et $b_j(p2)$ représentant les biais du neurone j, respectivement pour l'enfant $c1$, l'enfant $c2$, le parent $p1$ et le parent $p2$.

5.3 Mutation

Une fois un enfant généré par croisement, on doit intégrer la possibilité d'introduire des perturbations aléatoires. Soit mr le taux de mutation, on procéde, **pour chaque couche**,

1. $pm(biais)$ est la probabilité de mutation des biais, définie par :

$$pm(biais) = \frac{mr}{layerSize}$$

avec $layerSize$ le nombre de neurones dans la couche ;

2. $pm(poids)$ est la probabilité de mutation des poids, définie par :

$$pm(poids) = \frac{mr}{previousLayerSize}$$

avec $previousLayerSize$ le nombre de neurones dans la couche **précédente** ;

3. ensuite, pour chaque neurone, on tire un nombre aléatoire entre 0 et 1 : si ce nombre en inférieur à $pm(biais)$, alors on ajoute (ou on retire) au biais du neurone une petite quantité aléatoire (on pourra utiliser la fonction numpy.random.randn() qui utilise la loi normale) ;
4. puis, pour chaque poids entrant du neurone, on tire un nombre aléatoire entre 0 et 1 : si ce nombre en inférieur à $pm(poids)$, alors on ajoute (ou on retire) au poids concerné une petite quantité aléatoire.

Important : régler la mutation est critique pour la convergence. Dans notre cas, la fonction d'activation utilisée (elu, relu, tanh, ...) peut être plus ou moins sensible au modification des poids et des biais. Il faudra alors être attentif à la quantité à ajouter (ou à enlever) aux paramètres du réseau lors de la mutation, une trop grande perturbation empechera la convergence.

6 Paramétrage et résultats attendus

La démonstration à laquelle vous avez assisté succède à une optimisation utilisant les paramètres suivants :

Paramètre	Notation	Variable	Valeur
taille de la population	P	taillePopulation	400
taille de la sélection	S	tailleSelection	50
probabilité de croisement	pc	pc	0.8
taux de mutation	mr	mr	2.0
architecture	–	arch	(8, 24, 4)
paramètres d'évaluation	–	gameParams	(10, 10, 10)

Avec ce paramétrage, on peut s'attendre, en moyenne, aux résultats suivants :

1. après 100 itérations, le meilleur individu obtient un score compris en 0.2 et 0.5 ;
2. après 200 itérations, le meilleur individu obtient un score compris en 0.4 et 0.6 ;

Après ça, il est difficile de prédire le score atteignable par le meilleur individu, mais on considérera que l'algorithme converge correctement si après 1000 itérations, on obtient un score entre 0.7 et 1.0.

7 Améliorations

Voici quelques idées d'améliorations pour votre algorithme :

- si les algorithmes génétiques permettent l'optimisation d'un grand nombre de problème difficile, ils peuvent, selon le problème traité, prendre un temps conséquent pour générer une solution de bonne qualité. Dans notre cas, l'évaluation d'un individu prend du temps, surtout lors des dernières itérations, car les individus sont plus performants, et jouent donc plus longtemps. Une amélioration évidente est l'utilisation du multi-threading pour l'évaluation des individus, soit au sein même de la fonction d'évaluation (un individu joue plusieurs parties en même temps), soit au niveau au dessus (on évalue plusieurs individus en même temps). On pourra utiliser concurrent.futures pour mettre en place le threading via concurrent.futures.ProcessPoolExecutor.
- plusieurs types de croisements sont possibles, on pourrait par exemple copier les couches paires du premier parent et les couches impaires du deuxième parent pour générer le premier enfant, et inversement pour le deuxième enfant. Utiliser de manière aléatoire plusieurs types de croisement permet d'ajouter de la diversité dans la population et de potentiellement "débloquer" l'algorithme.
- on peut également utiliser des méthodes d'intensification sur les individus intéressants : on prend le meilleur individu de la population que l'on définit comme point de référence, puis, pour un certain nombre d'itération, on génère un grand nombre de clones du point de référence, clones que l'on fait ensuite muter. Si le meilleur de clones est plus performant que le point de référence, alors il devient lui-même le point de référence.
- on peut ajouter une option de réoptimisation : on charge un réseau précédemment entrainé sur une grille de 10 par 10, et on génère une population d'individus copiant ce réseau (avec quelques mutations) pour s'entrainer sur une taille de grille différente. Cela permettrait de générer un réseau de neurones pour un taille de grille particulière sans reprendre l'optimisation de zéro.
- il est également possible d'implémenter un algorithme génétique pour générer des arbres de décision.