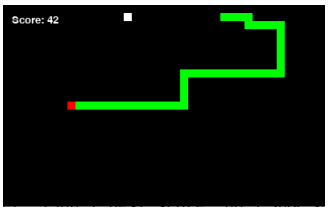
Snake : Faire des *high-scores* grâce à l'apprentissage par renforcement

Bastien Schnitzler *

19 novembre 2021



Sur console, sur téléphone, même sur calculatrice... qui ne connaît pas le jeu *Snake* ? Ce jeu d'arcade sorti en 1976 sous le nom de *Blockade* fait partie de la famille des jeux dont la formulation est très simple mais qui peut tenir en haleine un joueur dans d'interminables parties, car son espace d'états est très vaste. Dans ce projet, on propose d'implémenter ce jeu en Python avec l'aide de la librairie `pygame` [1]. Par ailleurs, on propose d'entraîner une intelligence artificielle sur la base des méthodes d'apprentissage par renforcement pour jouer au jeu.

1 Développement du jeu

1.1 Principe

Dans le jeu *Snake*, le joueur contrôle un serpent composé de blocs contigus qui se déplace dans un monde bidimensionnel avec pour objectif de collecter un maximum de nourriture. Le serpent est toujours en mouvement rectiligne et le joueur contrôle le changement de direction de la tête du serpent à l'aide de quatre entrées : haut, bas, gauche, droite. Lorsque la nourriture est mangée par le serpent, un nouveau point de nourriture apparaît dans la zone de jeu. Le jeu se termine si la tête heurte un des bords de la zone de jeu ou bien le corps du serpent.

*`bastien.schnitzler@recherche.enac.fr`

1. `pypi.org/project/pygame`

1.2 Implémentation attendue

L'implémentation en Python de ce jeu doit permettre de proposer une version jouable interactive du jeu ainsi qu'une version permettant d'entraîner une IA sans interface graphique. Plus généralement une structure de code rigoureuse est attendue pour séparer les fonctions principales du programme.

Une suggestion de fonctions principales serait :

— Créer une configuration de partie et la stocker
— Produire l'état suivant du jeu à partir de l'état précédent et de l'action choisie (moteur du jeu)
— Afficher l'état du jeu
— Enregistrer une partie

La liste précédente n'est qu'une suggestion, d'autres découpes en fonctions élémentaires peuvent convenir.

La gestion de l'interface graphique et de l'interaction avec le joueur se fera à l'aide de la librairie `pygame`.

1.3 Conseils

L'utilisation de la programmation orientée objet est conseillée pour ce projet.

Avant de commencer à écrire du code, un travail de réflexion sur papier peut aider pour mettre au clair la structure du programme. En particulier, le choix de représentation des différents éléments du jeu doit être défini : qu'est-ce qu'une configuration de partie ? Qu'est-ce que la donnée d'une partie ? Sous quelle forme la sauvegarder ? etc. On prendra soin de viser à une efficacité en mémoire de ces représentations.

2 Développement d'une IA

Une fois le jeu fonctionnel, on cherchera à développer une IA jouant au jeu. En particulier, la contrainte temps réel du jeu demande une implémentation peu coûteuse à évaluer, de l'ordre de 10ms de calcul pour choisir une action avant que le jeu ne passe à l'étape temporelle suivante. Cette contrainte pousse à s'intéresser aux méthodes d'apprentissage par renforcement.

2.1 Apprentissage par renforcement

Dans un cadre général, on considère un agent évoluant dans un espace d'état S. Pour un instant donné $t \in \mathbb{N}$ (temps discret dans le problème), l'agent se situe dans un état $s_t \in S$. Il peut effectuer une action $a_t \in A(s_t)$ où $A(s_t)$ désigne l'ensemble des actions possibles dans l'état s_t. La dynamique de l'environnement f permet alors de passer à l'état suivant s_{t+1} et on note $s_{t+1} = f(s_t, a_t)$. L'arrivée dans l'état s_{t+1} occasionne une récompense r_t (qui peut être positive ou négative) et on note ρ la fonction de récompense telle que $r_t = \rho(s_t, a_t)$.

Définition 1. *Une **trajectoire** $\sigma = (s_t, a_t)_{t\in\mathbb{N}}$ est une suite d'états-actions vérifiant :*

$$\forall t,\ s_{t+1} = f(s_t, a_t)$$

La définition précédente formalise l'intuition de chemin parcouru par un agent. L'objectif de l'agent est alors de maximiser les récompenses obtenues le long de sa trajectoire. En apprentissage par renforcement, on introduit très souvent la notion suivante :

Définition 2. *Pour $\gamma \in [0, 1]$ et pour une trajectoire $\sigma = (s_t, a_t)_{t\in\mathbb{N}}$, on note $r_t = \rho(s_t, a_t)$. La **récompense agrégée** d'une trajectoire σ est définie par*

$$R(\sigma) = \sum_{t=0}^{\infty} \gamma^t r_t$$

La définition précédente donne une évaluation de la performance d'une trajectoire. Lorsque les valeurs des récompenses sont bornées (ce qui est souvent le cas), le *facteur d'escompte* γ choisi dans $[0, 1[$ garantit que la récompense agrégée est bien définie (la série converge absolument). En pratique, le facteur d'escompte caractérise aussi l'importante que l'on accorde aux récompenses futures : plus sa valeur est proche de 1, plus le futur compte, et c'est l'inverse quand sa valeur s'approche de 0.

2.2 Q-learning

2.2.1 Principe

Dans les méthodes de Q-learning, on définit une fonction d'utilité état-action qui est une mesure de l'intérêt du choix d'une action dans un état donné. Cette fonction est appelée *fonction Q*. Pour $s \in S$ et $a \in A(S)$, elle est définie par :

$$Q(s, a) = \mathbb{E}[R((s_t, a_t)_{t\in\mathbb{N}}) \mid s_0 = s, a_0 = a]$$

C'est l'espérance de toutes les récompenses agrégées de trajectoires possibles depuis l'état s en choisissant l'action a. Quand l'espace des états S ainsi que l'espace d'actions $A = \cup_{s\in S} A(s)$ sont finis, on parle plutôt de Q-table : on voit la fonction Q comme un élément de $\mathbb{R}^{|S|\times|A|}$. Quand la fonction Q est connue, la stratégie de jeu est simplement la stratégie gloutonne :

$$a_t = \underset{a\in A(s_t)}{\arg\max}\, Q(s_t, a) \tag{1}$$

dans un état donné, on choisit l'action qui maximise l'utilité au sens de la fonction Q.

2.2.2 Apprentissage

L'enjeu est alors d'apprendre la Q-table. Dans le paradigme de l'apprentissage par renforcement, on va faire jouer l'agent au jeu et le laisser accumuler de l'expérience, c'est-à-dire

qu'on le laisse construire sa propre Q-table. Pour définir le comportement de l'agent, il faut définir sa *politique* :

$$\pi \ : \ s \in S \mapsto a \in A(s)$$

c'est-à-dire quel choix l'agent fait lorsqu'il se trouve dans un état donné. Par exemple, (1) définit une politique. Cette politique n'a pas besoin d'être déterministe, on peut avoir recours à un choix au hasard par exemple quand c'est nécessaire.

Une fois la politique de l'agent définie, celui-ci peut être lancé dans une partie et commence à accumuler de la connaissance. Cette accumulation est définie par une formule de mise à jour de la Q-table. Si l'on note Q_t la Q-table à l'instant t, la formule de mise à jour usuelle en Q-learning est la suivante :

$$Q_{t+1}(s_t, a_t) \leftarrow (1-\alpha)Q_t(s_t, a_t) + \alpha(r_t + \gamma \max_{a \in A(s_t)} Q_t(s_{t+1}, a))$$

où $\alpha \in [0, 1]$ est un taux d'apprentissage : la mise à jour est une combinaison convexe de l'information qu'on avait apprise précédemment $Q_t(s_t, a_t)$ et de la meilleure information disponible à cet instant sur la vraie valeur de la Q-table $r_t + \gamma \max_{a \in A(s_t)} Q_t(s_{t+1}, a)$ étant donné la nouvelle observation r_t.

2.2.3 Implémentation

Dans la partie 1, on a développé un jeu avec une structure pensée pour faire jouer un agent. Il s'agit maintenant de définir l'agent, c'est-à-dire sa politique, sa Q-table, sa mise à jour d'information. Comme dans la première partie, la question des représentations est importante : qu'est qu'un agent ? Comment stocker la Q-table ?

La mesure de la performance de l'agent est aussi importante. Il faudra définir tous les outils utiles pour pouvoir produire des graphes montrant la performance de l'agent en fonction du nombre de parties qu'on lui a fait jouer par exemple.

Bonus Si le temps le permet, on pourrait imaginer lancer le jeu en mode "contre l'IA" avec une même partie sur deux écrans : l'un pour le joueur et l'autre pour l'agent.