CentraleSupélec | université PARIS-SACLAY

Projet 2023

Connect4 agent

Reinforcement Learning

Étudiants :
Deseure Charron Flavien

Enseignants :
Hadiji Hédi

24 avril 2023

I Introduction au problème

I.1 Contexte

L'apprentissage par renforcement est une sous-discipline du machine learning qui se concentre sur l'optimisation des actions d'un agent pour maximiser une certaine récompense dans un environnement donné. Cette approche a montré des résultats prometteurs dans diverses applications, notamment dans les jeux. Dans le domaine des jeux, l'apprentissage par renforcement a été utilisé pour développer des agents capables de rivaliser, voire de surpasser, les compétences humaines. Des exemples notables incluent AlphaGo de DeepMind, qui a battu les meilleurs joueurs mondiaux de Go, ainsi que l'agent OpenAI Five, qui a réussi à vaincre des équipes professionnelles de Dota 2. Ces succès montrent que l'apprentissage par renforcement peut être utilisé pour résoudre des problèmes complexes et créer des agents intelligents capables d'affronter des adversaires humains.

I.2 Formalisation du problème

Le jeu de Connect Four est un jeu de société classique qui représente un problème intéressant pour les algorithmes d'apprentissage par renforcement. Le but du jeu est simple : aligner quatre jetons de sa propre couleur verticalement, horizontalement ou en diagonale sur une grille 6x7. Voici une formalisation plus précise de notre problème :

- **Espace d'états (S)** : L'espace d'états est constitué de toutes les configurations possibles du plateau de jeu. Le plateau de jeu de Connect Four est une grille 2D de dimensions 6x7, où chaque case peut être vide, occupée par un jeton du joueur 1 ou occupée par un jeton du joueur 2. Ainsi, il y a $3^{6\times7} \approx 10^{20}$ états possibles, bien que beaucoup d'entre eux ne soient pas atteignables en raison des règles du jeu.
- **Espace d'actions (A)** : L'espace d'actions représente les actions que les joueurs peuvent effectuer à chaque tour. Dans Connect Four, les joueurs peuvent choisir de placer un jeton dans l'une des 7 colonnes du plateau, à condition que la colonne ne soit pas déjà remplie. Ainsi, l'espace d'actions est un ensemble discret de 7 actions possibles.
- **Règles de transition (T)** : Les règles de transition définissent comment le plateau de jeu évolue en fonction des actions des joueurs. Lorsqu'un joueur choisit une colonne, son jeton est placé dans la case la plus basse disponible de cette colonne. Les joueurs alternent les tours jusqu'à ce que l'un des joueurs gagne ou que le plateau soit rempli, entraînant une égalité.
- **Fonction de récompense (R)** : La fonction de récompense attribue une valeur numérique à chaque transition d'état en fonction de l'action choisie et du résultat. Dans le contexte de Connect Four, la récompense peut être définie comme suit :
  - Si l'action mène à une victoire, attribuer une récompense de 1
  - Si l'action mène à une défaite, attribuer une récompense de -1
  - Si l'action mène à une égalité, attribuer une récompense nulle
  - Si l'action ne mène pas à la fin du jeu, attribuer une récompense nulle
- **Objectif** : L'objectif du jeu pour chaque joueur est de maximiser sa fonction de récompense cumulée en plaçant ses jetons de manière à aligner quatre jetons de sa couleur, soit horizontalement, verticalement ou en diagonale. Les agents d'apprentissage par renforcement sont entraînés pour apprendre des stratégies optimales en explorant l'espace d'états et en adaptant leur comportement en fonction des récompenses reçues.

II Algorithme

Dans ce projet, nous avons développé et évalué plusieurs agents d'apprentissage par renforcement pour le jeu de Connect Four en utilisant différentes techniques telles que les arbres de recherche Monte Carlo (MCTS) et les réseaux de neurones profonds avec Advantage Actor-Critic (A2C). Notre objectif était de créer un agent capable de rivaliser avec des joueurs humains et d'approfondir notre compréhension des défis et des opportunités associés à l'application de l'apprentissage par renforcement aux jeux.

II.1 Design et structure du code

Le code de ce projet est organisé en différentes parties pour faciliter la compréhension et la maintenance. Voici un aperçu des principales composantes du code :

- **Agent de base (base-agent.py)** : Ce fichier contient la classe de base Agent, qui définit l'interface commune pour tous les agents utilisés dans le projet. Les agents spécifiques hériteront de cette classe de base et implémenteront des méthodes supplémentaires. Voici les différents agents qui héritent de cette classe :
  - **Agent MCTS (mcts.py)** : L'agent MCTS implémente l'algorithme Monte Carlo Tree Search (MCTS) pour sélectionner la meilleure action à prendre dans un jeu de Connect Four. Il effectue plusieurs simulations (contrôlées par n-simulations) pour explorer les états possibles du jeu. Il utilise un facteur c-puct pour équilibrer l'exploration et l'exploitation lors de la sélection des actions. L'arbre MCTS est constitué de MCTSNode, qui stockent les informations sur les actions, les récompenses et les visites.
  - **Agent A2C (actor-critic.py)** : L'agent Advantage Actor-Critic utilise une approche d'apprentissage par renforcement appelée "Actor-Critic". Il a deux réseaux neuronaux, l'acteur et le critique. L'acteur est responsable de la sélection des actions, tandis que le critique estime la valeur de l'état actuel.L'agent A2C possède des méthodes pour mettre à jour les réseaux neuronaux, sauvegarder et charger les modèles, et sélectionner la meilleure action en fonction de l'état actuel.
  - **Agent Random (random.py)** : L'agent Random sélectionne une action aléatoire à chaque étape. Il n'apprend pas de ses actions et n'a pas de mémoire ou de stratégie pour s'améliorer au fil du temps. Toutes les méthodes de cette classe sont des opérations simples ou vides, car il n'y a pas de mise à jour ni de mécanisme de sauvegarde et de chargement pour cet agent.
- **Humain (human.py)** : Cette classe représente un joueur humain dans un jeu. Elle est conçue pour permettre à un utilisateur d'interagir avec un environnement de jeu et un agent en fournissant des actions.
- **Environnement de jeu (game.py)** : La classe Game gère le déroulement du jeu et l'interaction entre les agents et l'environnement. Elle prend en charge l'entraînement, l'évaluation et la visualisation des parties de jeu. La classe Game utilise les instances d'agents et d'environnement pour gérer les interactions et les mises à jour des agents. Les agents sont des instances de la classe Agent ou Human.
- **Script principal (main.py)** : Ce fichier est le point d'entrée du projet. Il contient le code pour initialiser l'environnement, créer les agents, et exécuter le jeu en faisant interagir les agents avec l'environnement. Le script principal gère également l'affichage des résultats et la sauvegarde des données d'apprentissage des agents.

III Agents

Nous avons implémentés différents agents afin de comparer leurs performances. Certains sont plus performant en terme de vitesse d'execution alors que d'autres prennent de meilleures décisions pour remporter la partie. Une autre différence est que la méthode A2C necessite un entrainement contrairement à MCTS.

III.1 Advantage Actor-Critic (A2C)

Advantage Actor-Critic (A2C) est un algorithme qui combine deux types d'apprentissage par renforcement :

- Les méthodes basées sur la politique. Elles optimisent une fonction qui définit une correspondance entre chaque état et la meilleure action correspondante.
- Les méthodes basées sur la valeur. Elles optimisent une fonction qui établit la récompense que l'agent peut obtenir s'il commence dans un état donné et agit ensuite conformément à notre politique.

Voici une description des points importants de l'agent A2C et de son fonctionnement :

- **Acteur** : Le réseau de neurones de l'acteur prend en entrée l'état actuel du jeu et retourne une distribution de probabilité sur l'ensemble des actions possibles. Il s'agit d'une politique stochastique qui permet à l'agent de prendre des décisions de manière aléatoire, tout en favorisant les actions les plus prometteuses.
- **Critique** : Le réseau de neurones du critique prend en entrée l'état actuel du jeu et retourne une estimation de la valeur de cet état. La valeur d'un état est définie comme la somme des récompenses attendues à long terme, à partir de cet état, sous la politique actuelle de l'agent.
- **Procédure d'entraînement** : L'agent A2C est entraîné en suivant les étapes suivantes :
  1. Collecte des trajectoires via l'interaction de l'agent avec l'environnement (séquences d'observations, d'actions et de récompenses).
  2. Calcul de la fonction de coût avec la fonction avantage $A(s, a)$. Celle-ci calcule l'avantage relatif d'une action par rapport aux autres actions possibles dans un état. Il s'agit donc de soustraire la valeur moyenne de l'état $V(s)$ de la paire d'actions de l'état $Q(s, a)$:

     $$A(s, a) = Q(s, a) - V(s)$$

     Cependant, cela nécessiterait deux fonctions de valeur $V(s)$ et $Q(s, a)$, ce qui est difficile à implémenter en pratique. Pour résoudre ce problème, nous utilisons l'erreur TD comme estimateur :

     $$A(s, a) = r + \gamma V(s') - V(s)$$

     En ajoutant la fonction avantage au calcul de gradient de la politique, nous obtenons :

     $$\nabla_\theta J(\theta) = \mathbb{E}_\tau[\sum_{t=0}^{T} \nabla_\theta \ log \ \pi_\theta(s_t|a_t) A_{\pi_\theta}(s_t, a_t)]$$

  3. Mise à jour des poids des réseaux de neurones via la fonction de coût et les taux d'apprentissage de chaque réseau.

- **Méthodes** : L'agent A2C dispose des méthodes suivantes pour la prise de décision :
  - **get-best-action** : Cette méthode prend en entrée une observation et retourne l'action ayant la plus haute probabilité, selon la politique actuelle de l'agent.
  - **update** : Cette méthode met à jour les paramètres des réseaux de neurones de l'agent en utilisant l'algorithme de descente de gradient stochastique.
  - **save** : Cette méthode permet de sauvegarder les poids des réseaux de neurones pour l'acteur et et le critique afin de pouvoir les réutiliser ultérieurement.
  - **load** : Cette méthode permet de charger les poids des réseaux de neurones à partir d'un fichier sauvegardé précédemment.

Nous avons utilisé deux architectures pour les réseaux de neurones Actor et Critic : l'une basée sur des réseaux neuronaux convolutifs (ConvNet) et l'autre basé sur des réseaux Dense (Fully Connected Net).

IV Hyperparamètres utilisés

Nous allons vous présenter ici uniquement les hyperparamètres utilisés pour l'entraînement de l'agent A2C :

- **nb_epoch** : Le nombre d'itérations d'apprentissage effectuées sur les données.
- **gamma** : Le facteur de remise, qui détermine l'importance accordée aux récompenses futures par rapport aux récompenses immédiates.
- **optimiser** : Le choix de l'optimiseur pour mettre à jour les poids du réseau.
- **lr_actor** : Le taux d'apprentissage pour mettre à jour les poids du réseau de politiques.
- **lr_critics** : Le taux d'apprentissage pour mettre à jour les poids du réseau d'évaluation de la valeur de l'état.
- **eps_init** : La probabilité initiale d'effectuer une action aléatoire, pour l'exploration.
- **eps_min** : La probabilité minimale d'effectuer une action aléatoire.
- **eps_step** : Le pas de réduction de la probabilité d'effectuer une action aléatoire, pour la recherche par descente de gradient stochastique.

Nous avons choisi les hyperparamètres suivants de manière empirique : **nb_epoch** $= 10^4$, **gamma** $= 0.99$, **optimiser** = Adam, **lr_actor** $= 10^{-5}$, **lr_critics** $= 10^{-5}$, **eps_init** $= 0.5$, **eps_min** $= 10^{-5}$, **eps_step** $= 10^{-3}$