le cnam
école d'ingénieur·e·s

Deep Learning

 github/projet

Réalisation : Alexis GUILLTON & Clément MARIE--BRISSON

Superviseur : Philippe CARRRE

2023-03-17

Table des matières

Développement d'un solveur de labyrinthe via du deep learning

1 – Développement d'un jeu

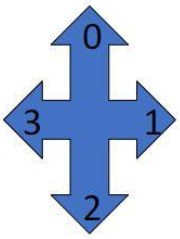
Le jeu ait été développé en utilisant un tableau 2D pour représenter le plateau, chaque élément du tableau correspondant à un emplacement spécifique sur le plateau. Il s'agit d'un moyen simple et efficace de créer un plateau de jeu, en particulier pour les jeux dont la taille et le nombre d'éléments sont fixes.

Le positionnement des éléments sur le plateau est simple, chaque élément étant placé à un endroit spécifique du plateau. Il s'agit d'une approche courante pour les jeux de société, qui fonctionne également pour d'autres jeux comme le Tic-Tac-Toe.

L'interaction avec le jeu se fait via un agent. L'agent peut effectuer un déplacement en sélectionnant un emplacement sur le plateau, et le jeu peut mettre à jour le plateau et vérifier s'il y a un gagnant ou une égalité. L'utilisation de fonctions telles que *application_action()* et *choose_action()* contribue à rendre le code plus organisé et plus facile à comprendre.

Voici une représentation du jeu :

Start			
Dragon Go to Start		Dragon Go to Start	
		Reward = 0	Dragon Reward = -1 Go to Start
	Dragon Go to Start		**Jail** Reward = 1 Go to Start



Les déplacements sont gérés via une variable pouvant avoir 4 valeurs différentes : 0, 1, 2 et 3 qui correspond à une direction différente.

Les cases dragons ont des valeurs spécifiques qui permet de détecter la case, le joueur aura perdu et sera renvoyé au début.

La case Jail (ou fin dans notre cas) correspond à la fin du labyrinthe, si le joueur atteint cette case il aura gagné et sera également renvoyé au début.

2 – Q-learning algorithme

(voir Q_learning.py)

2.1 - Stratégie utilisée pour développer l'apprentissage

La stratégie utilisée pour développer l'apprentissage est l'apprentissage *Q-learning*, qui est un algorithme d'apprentissage par renforcement populaire et efficace. L'algorithme d'apprentissage *Q-learning* fonctionne par la mise à jour itérative d'une fonction action-valeur, mat_q, qui estime le rendement attendu pour chaque paire état-action. L'agent sélectionne une action sur la base d'un compromis exploration-exploitation, en choisissant l'action dont la valeur mat_q est la plus élevée avec une certaine probabilité, ou en choisissant une action aléatoire avec une probabilité plus faible. L'algorithme utilise un facteur d'actualisation pour équilibrer les récompenses immédiates et futures.

Dans notre cas, nous avons fait tourner l'algorithme sur 5 000 parties, chaque partie ayant un nombre de coups maximum de 100. (Pour éviter les boucles infini)

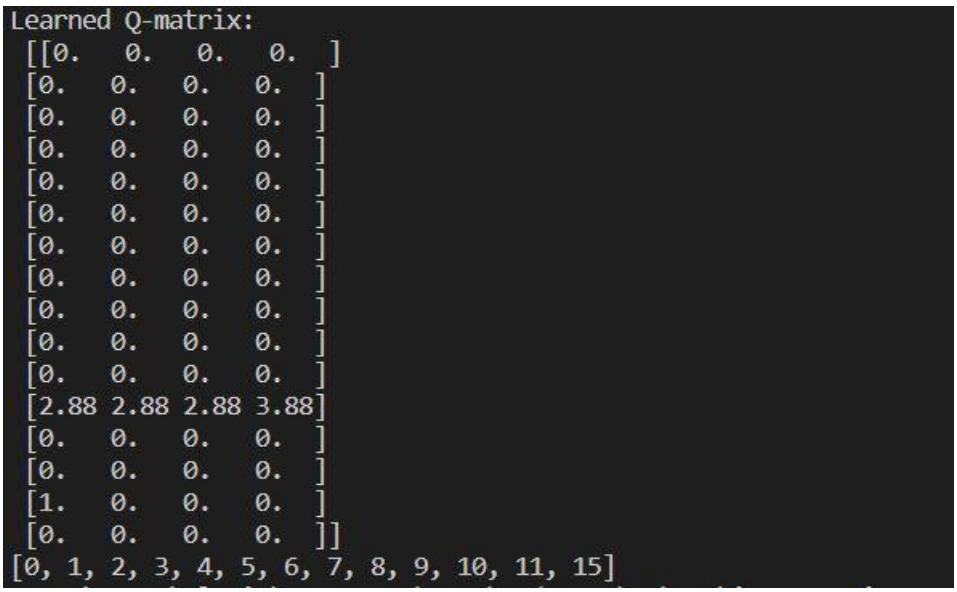
2.2 - Premiers résultats avec des récompenses R=1;0;-1

Pour les récompenses R=1;0;-1, l'agent reçoit une récompense de 1 s'il atteint le but, une récompense de -1 s'il tombe dans un trou, et une récompense de 0 sinon. Après avoir exécuté l'algorithme d'apprentissage pendant un certain nombre d'épisodes, on obtient la matrice mat_q et la politique qui en découlent.

La matrice mat_q est un tableau dans lequel chaque ligne représente un état et chaque colonne une action. Les entrées de la matrice sont les récompenses futures attendues que l'agent recevra s'il entreprend cette action dans cet état. La politique est une correspondance entre les états et les actions, où l'agent sélectionne l'action ayant la valeur mat_q la plus élevée dans chaque état.

L'efficacité de l'algorithme peut être évaluée en examinant la matrice mat_q et la politique qui en résultent. Si la matrice mat_q est précise, l'agent sélectionnera l'action optimale dans chaque état et atteindra l'objectif avec la plus grande probabilité possible.

Voici la matrice et le chemin optimal après apprentissage :



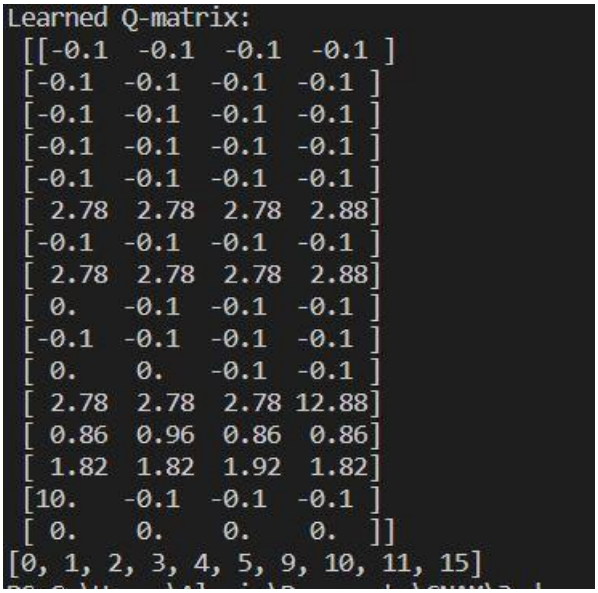
Dans les premiers résultats, la matrice mat_q et la politique sont obtenues après avoir exécuté l'algorithme pendant un certain nombre d'épisodes. La matrice mat_q montre que l'agent a appris les récompenses futures attendues pour chaque paire état-action, et la politique montre que l'agent sélectionne l'action ayant la valeur mat_q la plus élevée dans chaque état. Cependant, l'algorithme semble lent en termes de convergence et la politique qui en résulte n'est pas très efficace. L'agent prend souvent des actions sous-optimales, ce qui réduit ses chances d'atteindre l'objectif.

2.3 - Algorithme évolué avec des récompenses modifiées :

Pour accroître l'efficacité de l'algorithme, les récompenses peuvent être modifiées. Une modification possible consiste à augmenter la récompense pour avoir atteint le but et à diminuer la pénalité pour être tombé dans un trou. Cela encourage l'agent à se concentrer sur l'atteinte de l'objectif plutôt que sur l'évitement des trous.

Par exemple, les récompenses peuvent être modifiées comme suit : R=10;0;-1, l'agent recevant une récompense de 10 s'il atteint l'objectif, une récompense de -1 s'il tombe dans un trou et une récompense de 0 dans le cas contraire. Après avoir exécuté l'algorithme d'apprentissage mat_q avec les récompenses modifiées, on obtient la matrice mat_q et la politique qui en résultent.

Voici la matrice après apprentissage et le chemin optimal :



Les récompenses modifiées conduisent à une politique plus efficace, où l'agent a plus de chances d'atteindre l'objectif. La matrice mat_q montre que les récompenses futures attendues pour atteindre l'objectif sont beaucoup plus élevées que les récompenses futures attendues pour tomber dans un trou. La politique montre que l'agent est plus susceptible de prendre des mesures qui mènent à l'objectif et moins susceptible de prendre des mesures qui mènent à un trou.

Dans l'ensemble, l'algorithme d'apprentissage mat_q est une méthode efficace pour apprendre à jouer à un jeu. En modifiant les récompenses, l'efficacité de l'algorithme peut être améliorée.

3 – Deep Q-learning

(voir DeepQLearning.py)

Dans cette partie on cherche à implémenter un algorithme de Deep Q learning (DQN).

Pour rappel voici un tableau récapitulatif des différences entre le Q-learning, le Deep Q-learning et le Deep Q Network :

	Tabular Q-learning (TQL)	**Deep Q-learning (DQL)**	**Deep Q-network (DQN)**
Is it an RL algorithm?	Yes	Yes	No (unless you use DQN to refer to DQL, which is done often!)
Does it use neural networks?	No. It uses a table.	Yes	No. DQN is the neural network.
Is it a model?	No	No	Yes (but usually not in the RL sense)
Can it deal with continuous state spaces?	No (unless you discretize them)	Yes	Yes (in the sense that it can get real-valued inputs for the states)
Can it deal with continuous action spaces?	Yes (but maybe not a good idea)	Yes (but maybe not a good idea)	Yes (but only the sense that it can produce real-valued outputs for actions).
Does it converge?	Yes	Not necessarily	Not necessarily
Is it an online learning algorithm?	Yes	No, if you use experience replay	No, but it can be used in an online learning setting

C'est une variante de l'algorithme d'apprentissage Q qui utilise des réseaux de neurones profonds pour se rapprocher d'une fonction Q qui représente la récompense escomptée de l'action dans un état donné.

La classe ***DeepQNetwork*** initialise les paramètres DQN tels que les tailles d'état et d'action, un tampon pour stocker l'expérience passée, le taux de mise à jour (gamma), le taux d'exploration (epsilon) et le taux d'apprentissage. Il utilise également l'API Keras pour définir l'architecture du réseau neuronal. Il se compose de deux couches cachées avec des activations ReLU et d'une couche de sortie linéaire. La fonction de perte est l'erreur quadratique moyenne et l'optimiseur est Adam.

La méthode ***'Remember'*** est utilisée pour stocker l'expérience de l'agent dans une mémoire tampon. Le tampon contient l'état actuel, l'action entreprise, la récompense reçue, l'état suivant et un indicateur booléen indiquant si l'épisode est terminé.

La méthode ***'choose_action'*** choisit l'action de l'agent en fonction d'une politique epsilon-greedy qui choisit au hasard une action avec une probabilité epsilon, ou choisit l'action avec la valeur Q la plus élevée prédite par le réseau de neurones.

La méthode ***'iterate'*** met à jour les poids du réseau neuronal en échantillonnant une série d'expériences à partir du tampon et en calculant la valeur Q cible à l'aide de l'équation de Bellman. Nous formons ensuite un réseau de neurones pour minimiser la différence entre la valeur Q prédite et la valeur Q cible en utilisant la perte d'erreur quadratique moyenne.

La fonction '***application_action'*** définit les règles de déplacement de l'agent dans le labyrinthe, comme vérifier si l'agent heurte un mur, atteint l'objectif ou se déplace dans un espace vide. Renvoie le statut suivant, la récompense reçue et un indicateur booléen indiquant si l'épisode est terminé.

La boucle principale du programme s'exécute pendant un nombre fixe d'épisodes et initialise le labyrinthe sous la forme d'un tableau à deux dimensions. Aplatissez le labyrinthe en un vecteur d'état et initialisez le DQN. Il parcourt ensuite un nombre fixe d'étapes dans chaque épisode, sélectionne une action, l'exécute dans l'émulateur et utilise la méthode de lecture pour mettre à jour les poids DQN lorsque le tampon est suffisamment plein.

Les résultats sont inexploitables en l'était puisqu'une erreur semble persister dans la mise en place du réseau, le modèle tourne sans apprendre.

```
1/1 [==============================] - 0s 33ms/step
1/1 [==============================] - 0s 35ms/step
1/1 [==============================] - 0s 35ms/step
1/1 [==============================] - 0s 33ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 37ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 30ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 28ms/step
...
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 85ms/step
1/1 [==============================] - 0s 29ms/step
```

Le modèle tourne à l'infini