Simulation d'une voiture autonome avec DDQN et Pygame

Rapport rédigé par Rachdaoui Rachida ***Filière : ID2***

Encadré par Aziz KHAMJANE ***Date : décembre , 2024***

Résumé (Abstract)

Cet article propose une simulation de conduite autonome utilisant l'apprentissage par renforcement (RL) dans un environnement personnalisé développé avec Pygame. L'algorithme Double Deep Q-Learning (DDQN) est utilisé pour entraîner un agent à naviguer sur des pistes tout en évitant les obstacles. Les résultats montrent une amélioration progressive des performances de l'agent.

Contexte et Objectif

Ce projet pédagogique explore l'utilisation de l'apprentissage par renforcement (RL) pour la conduite autonome. En mettant en œuvre l'algorithme Double Deep Q-Learning (DDQN) dans un simulateur interactif développé avec Pygame, l'objectif est de former un véhicule autonome à naviguer efficacement tout en évitant les obstacles. Cette simulation offre une approche pratique pour comprendre les concepts fondamentaux du RL.

Reinforcement Learning (RL)

L'apprentissage par renforcement (RL) est une technique d'apprentissage automatique où un agent apprend par essais et erreurs dans un environnement interactif, en recevant des récompenses ou des punitions

pour guider ses actions. Contrairement à l'apprentissage supervisé, qui utilise des solutions prédéfinies, le RL vise à maximiser la récompense cumulée en trouvant les actions les plus adaptées.

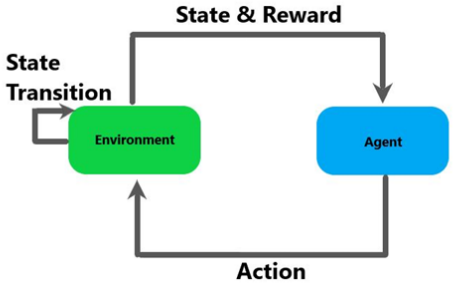

Figure 1. principe de l'apprentissage par renforcement

La Q-Table dans le Q-Learning

Dans le Q-learning classique, la fonction Q est représentée par une Q-table, où chaque ligne correspond à un état s et chaque colonne à une action a. Les valeurs de cette table Q(s,a), estiment la récompense cumulée future obtenue en choisissant l'action a dans l'état s selon une politique optimale.

Exemple de Q-Table

État s	Action a_1	Action a_2	Action a_3
s_1	$Q(s_1, a_1)$	$Q(s_1, a_2)$	$Q(s_1, a_3)$
s_2	$Q(s_2, a_1)$	$Q(s_2, a_2)$	$Q(s_2, a_3)$
s_3	$Q(s_3, a_1)$	$Q(s_3, a_2)$	$Q(s_3, a_3)$

Les mises à jour dans la Q-table s'effectuent selon l'équation de Bellman :

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

Composantes de l'Équation :

- α : Taux d'apprentissage, qui contrôle la vitesse de mise à jour.
- r : Récompense immédiate obtenue après avoir exécuté l'action a.
- γ : Facteur de discount, qui détermine l'importance des récompenses futures.
- max a′Q(s′,a′): Meilleure estimation de la valeur Q pour l'état suivant s′.

DQN

Le Deep Q-Learning (DQN) combine le Q-learning et les réseaux neuronaux. Plutôt que d'utiliser une Q-table, il utilise un réseau neuronal pour approximer la fonction Q(s,a). Ce réseau prend en entrée l'état et génère en sortie les valeurs Q pour chaque action possible. Cette approche est nécessaire lorsque l'espace des états est vaste ou continu, rendant une Q-table impraticable.

Pendant la simulation, toutes les expériences ⟨s,a,r,s′⟩ sont enregistrées dans **une mémoire de replay**. Lors de l'entraînement, des mini-lots sont échantillonnés aléatoirement de cette mémoire plutôt que d'utiliser les transitions les plus récentes. Cette méthode réduit la corrélation entre les exemples consécutifs, ce qui aide à éviter les minima locaux et améliore la stabilité de l'apprentissage. De plus, l'utilisation du replay d'expérience facilite le réglage des hyperparamètres et rapproche l'entraînement du cadre classique de l'apprentissage supervisé.

Double DQN

Dans le Double DQN, au lieu d'utiliser les mêmes poids pour évaluer et sélectionner une action, un deuxième réseau avec des poids distincts est introduit pour évaluer l'action choisie par le réseau DQN original. Cette approche permet de sélectionner l'action qui maximise la valeur Q dans l'état suivant. Ainsi, une politique gloutonne est maintenue pour la sélection des actions, mais l'évaluation de cette politique est effectuée de manière plus précise en utilisant des poids différents.

Limites du Q-Learning et Choix de DDQN

Limites du Q-Learning :

1. Inefficace pour les espaces d'états vastes ou continus (dépend d'une Q-table).
2. Biais de surévaluation des actions optimales (maximisation biaisée).
3. Instabilité due aux mises à jour directes et aux corrélations dans les données d'entraînement.

Pourquoi DDQN ?

1. Réduire le biais grâce à deux réseaux distincts (évaluation et cible).
2. Gère les espaces complexes avec des réseaux neuronaux.
3. Stabilise l'apprentissage via le replay d'expérience et les mises à jour périodiques.

Architecture :

Couche	Type	Taille (Sortie)	Fonction d'Activation
Entrée	Vecteur d'état	`(input_dims,)`	-
Couche cachée 1	Dense	`fc_dims`	ReLU
Couche cachée 2	Dense	`fc_dims`	ReLU
Couche cachée 3	Dense	`fc_dims`	ReLU
Sortie	Dense	`n_actions`	-

Pourquoi ReLu ?

Car elle est simple, rapide et permet une meilleure convergence du réseau. Elle évite le problème du gradient disparu et rend le modèle plus efficace en activant uniquement les neurones avec des entrées positives.

Paramètres :

Hyperparamètre	Description
α (alpha)	Taux d'apprentissage pour la mise à jour des poids du modèle.
γ (gamma)	Facteur de réduction des récompenses futures (importance des récompenses futures).
ε (epsilon)	Probabilité d'explorer de manière aléatoire (exploration vs exploitation).
ε_dec	Taux de décroissance de epsilon après chaque épisode.
ε_min	Valeur minimale de epsilon (fin de l'exploration).
batch_size	Taille des mini-lots utilisés pour l'entraînement.
mem_size	Taille de la mémoire de replay pour stocker les transitions.
replace_target	Fréquence de mise à jour des poids du réseau cible (en nombre d'itérations).
n_actions	Nombre d'actions possibles que l'agent peut prendre.
input_dims	Dimensions des entrées (observation de l'état).

Perte (loss) : La perte est calculée à l'aide de l'erreur quadratique moyenne (**MSE**) entre la valeur Q actuelle et la valeur Q cible, ce qui permet d'ajuster les poids du réseau neuronal.

La **MSE** est choisie comme fonction de perte car :

1. **Adaptée à la régression** : Elle mesure efficacement la différence entre les valeurs Q prédites et cibles.
2. **Punir les grandes erreurs** : Les erreurs importantes sont pénalisées davantage, aidant à une meilleure correction.
3. **Simple et efficace** : Facile à intégrer dans l'optimisation par rétropropagation et assure une convergence stable.
4. **Standard éprouvé** : Largement utilisée et alignée avec l'objectif du Q-Learning.

Mise à jour des modèles dans le DDQN

Modèle d'évaluation (q_eval) : Il est mis à jour à chaque itération d'entraînement en utilisant la rétropropagation pour ajuster les poids en fonction de la différence entre les valeurs Q prédites et les cibles calculées.

Qeval(s,a) =minimize(loss(Qeval(s,a),Qtarget(s,a)))

Modèle cible (q_target) : Il est mis à jour périodiquement pour stabiliser l'entraînement, en copiant les poids du modèle d'évaluation à intervalles réguliers (définis par replace_target).

qtarget.set_weights(qeval.get_weights())

Méthode de Gradient et Mise à Jour de l'Epsilon

La méthode de descente de gradient est utilisée pour optimiser les poids du réseau neuronal en ajustant les paramètres afin de minimiser l'erreur entre les valeurs Q prédites et les valeurs cibles. En parallèle, l'epsilon (taux d'exploration) est mis à jour à chaque itération selon la règle suivante :

$\epsilon = \epsilon \times \epsilon_{dec}$ si $\epsilon > \epsilon_{min}$ sinon $\epsilon = \epsilon_{min}$

Cela permet de diminuer progressivement l'exploration (epsilon) au fur et à mesure que l'agent apprend, favorisant ainsi l'exploitation des connaissances acquises.

Ce processus aide à améliorer l'efficacité de l'apprentissage tout en évitant les oscillations ou les divergences du modèle.

Environnement simulé (pygame):

L'environnement simule une route sur laquelle un véhicule autonome doit naviguer tout en évitant des obstacles.

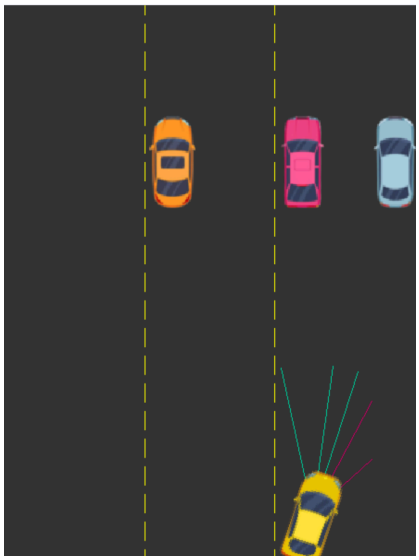

Figure 2. Environnement simulé

Le véhicule utilise un capteur à rayons (*RayCaster*) pour détecter les obstacles.

Observations :

- Position, vitesse et angle du véhicule.
- Détection d'obstacles autour du véhicule.
- Informations sur les collisions (obstacle ou sortie des limites de l'écran).

Actions possibles :

- Tourner à gauche (rotation horaire).
- Avancer (accélération).
- Tourner à droite (rotation antihoraire).

Fonction de récompense

Pénalité pour collision : -10 en cas de collision.

Récompense pour éviter les collisions : +2 après 100 étapes sans dommage.

Récompense pour la vitesse : Bonus proportionnel à la vitesse, avec une pénalité pour l'immobilité.

Récompense pour la distance parcourue : Bonus jusqu'à 5.

Utilisation de Streamlit

Streamlit a été utilisé pour visualiser en temps réel l'évolution des récompenses cumulées par épisode pendant l'entraînement du modèle DDQN, à travers un graphique dynamique.

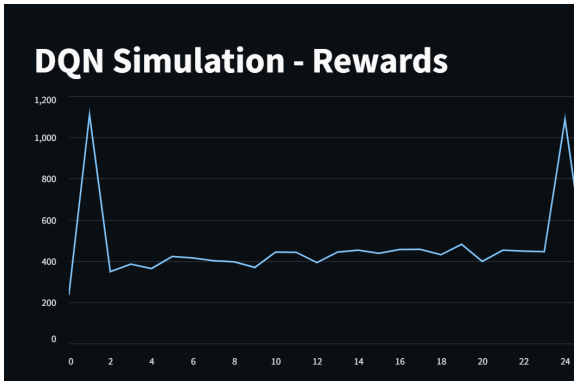

Figure 3. **Visualisation des Récompenses par Épisode(** agent n'est pas encore entraîné **)**

Remarque :

"Le processus d'entraînement du modèle a été limité par des contraintes de temps et les performances de mon ordinateur, ce qui a rendu l'entraînement du modèle complet difficile à réaliser dans les conditions actuelles. Cependant, l'objectif principal de ce projet est de comprendre les principes fondamentaux de l'apprentissage par renforcement (RL), notamment l'application de l'algorithme Double Deep Q-Learning (DDQN).

Une fois que les ressources nécessaires seront disponibles, l'entraînement du modèle pourra être poursuivi pour obtenir des résultats plus complets.

Extensions possibles

Si le temps et les ressources le permettent, plusieurs extensions pourraient être envisagées pour enrichir le projet .

Formation multi-agents : La coordination entre plusieurs véhicules autonomes dans un même environnement, afin d'aborder des problématiques complexes de coopération et de compétition entre agents.

Utilisation de CARLA pour intégrer des capteurs avancés : Exploitation des caméras, radars et lidars disponibles dans l'environnement de simulation CARLA afin de traiter les données d'images et d'améliorer la perception des véhicules autonomes. Cela permettrait de simuler des scénarios plus réalistes et de développer des solutions robustes pour l'analyse d'images et la prise de décision.

Schéma du principe :

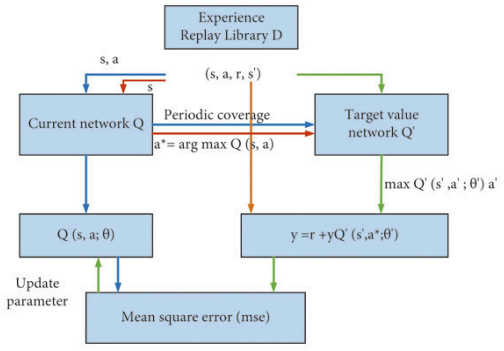

Figure 4. Schéma du principe

Difficultés Rencontrées

Au cours du développement de ce projet, plusieurs défis techniques et logistiques ont été rencontrés, notamment liés aux choix d'environnement de simulation, à l'entraînement du modèle et à la gestion des ressources. Ces difficultés sont détaillées ci-dessous :

1. Problèmes avec les Environnements de Simulation

Essai de divers environnements : J'ai testé plusieurs plateformes de simulation telles que CARLA, Unity, et d'autres. Cependant, leur complexité technique et leurs exigences en termes de ressources ont constitué un obstacle majeur :

- **CARLA :** Nécessite une carte graphique puissante pour exécuter les simulations, ce qui dépasse les capacités de mon appareil.
- **Unity :** Exige des compétences avancées en modélisation 3D pour concevoir l'environnement, ce qui demande un investissement important en temps pour maîtriser l'outil.

Finalement, j'ai opté pour un environnement personnalisé développé avec **Pygame**, qui offre une alternative plus légère et adaptée à mes contraintes matérielles.

2. Limites Matérielles et Entraînement du Modèle

Mon appareil personnel n'étant pas équipé d'un GPU performant, l'entraînement du modèle s'est avéré impossible localement. Pour résoudre ce problème :

- **Utilisation de plateformes distantes :** J'ai utilisé Kaggle et Google Colab pour bénéficier de GPU gratuits pour l'entraînement.
- **Problème d'intégration locale :** La simulation devant s'exécuter localement, j'ai rencontré des difficultés pour relier les données générées par la simulation locale avec les modèles entraînés à distance sur Kaggle ou Colab.

3. Mise en Place d'une Solution Hybride

Pour établir un pont entre la simulation locale et l'entraînement distant, j'ai utilisé **Firebase**, une base de données en temps réel :

- Deux collections ont été créées :
  - Une pour les **observations** collectées par la simulation.
  - Une autre pour les **décisions** prises par le modèle après traitement.
- Cette approche fonctionnait bien initialement, mais a rencontré des limitations dues à l'épuisement du quota gratuit de Firebase, ce qui a interrompu les sessions de simulation.

Références :

- rl-book/en2024 at master · ZhiqingXiao/rl-book
- Q-Learning - GeeksforGeeks
- narenkarthicktp/self-driving-car: A simple simulation of a self driving car in a highway traffic using feed forward neural networks. **(pour le setup de l'environnement)**
- Deep Reinforcement Learning with Double Q-learning