Rapport CART-POLE

By Giuliano Taurone 59333 & Sami El Yaghmouri 60731

Introduction

Dans le monde de l'intelligence artificielle, l'apprentissage par renforcement (RL) se distingue comme une méthodologie puissante qui permet aux agents d'apprendre en interagissant avec leur environnement. En particulier, Deep Q-Learning (DQN) combine l'efficacité des réseaux neuronaux profonds avec la structure des algorithmes Q-Learning, surmontant ainsi les limites des tables de valeurs Q traditionnelles.

Le projet CartPole représente un exemple classique et significatif pour démontrer le potentiel du DQN. L'objectif principal est de maintenir un poteau en équilibre sur un chariot en mouvement grâce à une série d'actions optimales. Cependant, le problème devient rapidement complexe puisque l'espace d'état comprend des dimensions continues, c'est là qu'intervient DQN, qui utilise un réseau neuronal comme fonction de mappage pour estimer les valeurs.

Approche et Méthodologie

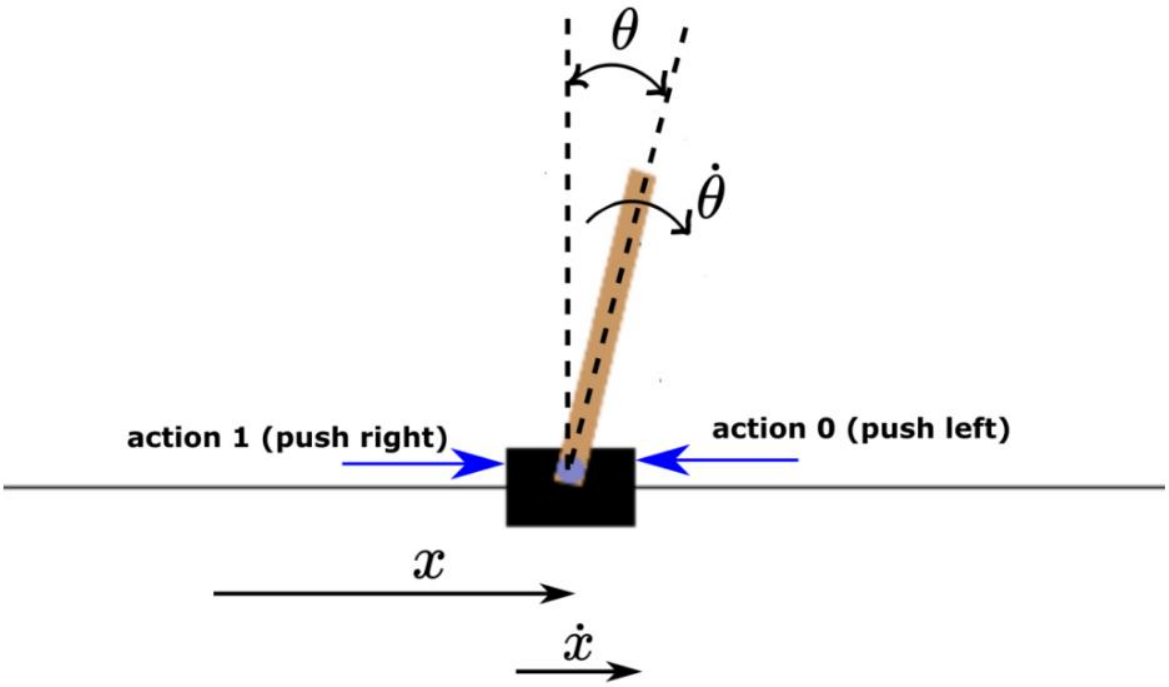
Peut-on modifier l'algorithme Q-learning de manière à ne pas avoir besoin de stocker toutes les valeurs de la fonction d'action pour différents états ? En principe, la réponse est OUI. L'idée est d'estimer une fonction qui produira les valeurs de la fonction d'action pour un état donné. Autrement dit, nous voulons apprendre directement cette fonction ou une Map qui transformera les états d'entrée en valeurs de fonction d'action.

Prenons par exemple l'environnement **Cart Pole OpenAI Gym**. Ce système a quatre états :

- La position du chariot.
- La vitesse du chariot.
- La rotation angulaire du poteau.
- La vitesse angulaire du poteau.

Il y a deux actions disponibles :

1. Pousser le chariot vers la gauche (action 0).
2. Pousser le chariot vers la droite (action 1).



L'objectif de contrôle est de trouver une séquence d'actions optimales qui maintiendront le poteau en position verticale.

Pour cela, nous souhaitons estimer (ou en termes d'apprentissage automatique, apprendre) une fonction qui mappe le vecteur d'état aux fonctions de valeur d'action

$$S = [x, \dot{x}, \theta, \dot{\theta}]^T$$

Pour cet état particulier. Cela peut être représenté comme suit :

$$\begin{bmatrix} Q(S, A = 0) \\ Q(S, A = 1) \end{bmatrix} = F\Big(S = \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix}\Big)$$

Où F est une fonction que nous voulons estimer ou apprendre. Dans cette approche, est modélisé par un réseau neuronal, qui agit comme une fonction de mappage non linéaire capable de produire des valeurs pour chaque action disponible (actions A=0 (pousser le chariot vers la gauche) et A=1 (pousser le chariot vers la droite).

Cette section détaillera la conception, la mise en œuvre et les résultats obtenus en utilisant cette approche, en illustrant comment elle surmonte les limitations des méthodes Q-learning basées sur des tableaux.

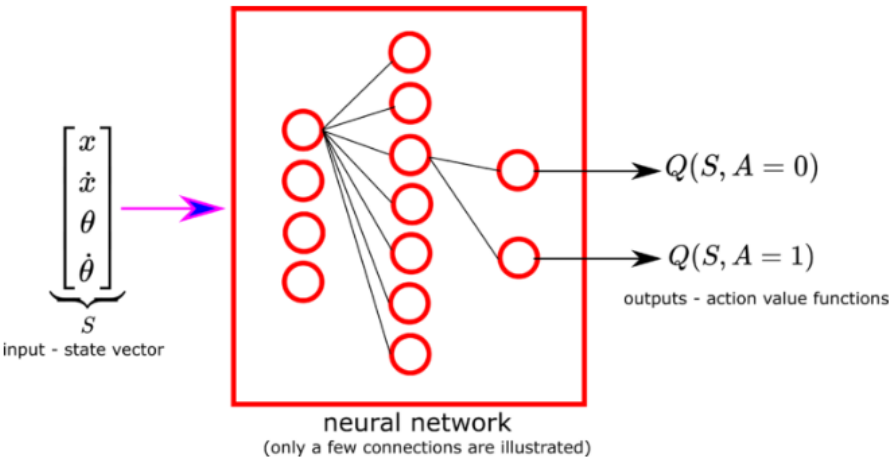


Figure 2: Neural network for approximating the action value functions.

Pour entraîner le réseau, nous devons introduire le concept de mémoire de répétition (replay buffer), également connu sous le nom de « mémoire d'expérience ». Ce concept joue un rôle crucial dans le processus d'apprentissage.

Considérons une séquence d'états et d'actions dans le système CartPole illustré ci-dessous. Prenons un état initial de l'épisode 1, note par S11 Supposons que nous déterminons une action A11 de contrôle et que nous l'appliquons à l'environnement. Nous obtenons alors une récompense R11 et l'environnement évolue vers l'état suivant S12. Ensuite, nous répétons cette procédure pour l'état S12. Nous calculons l'action A12, l'appliquons à l'environnement pour obtenir la récompense R12, tandis que l'environnement évolue vers l'état S13. Ce processus se poursuit jusqu'à la fin de l'épisode, qui compte étapes temporelles. L'état terminal est alors S1N1.

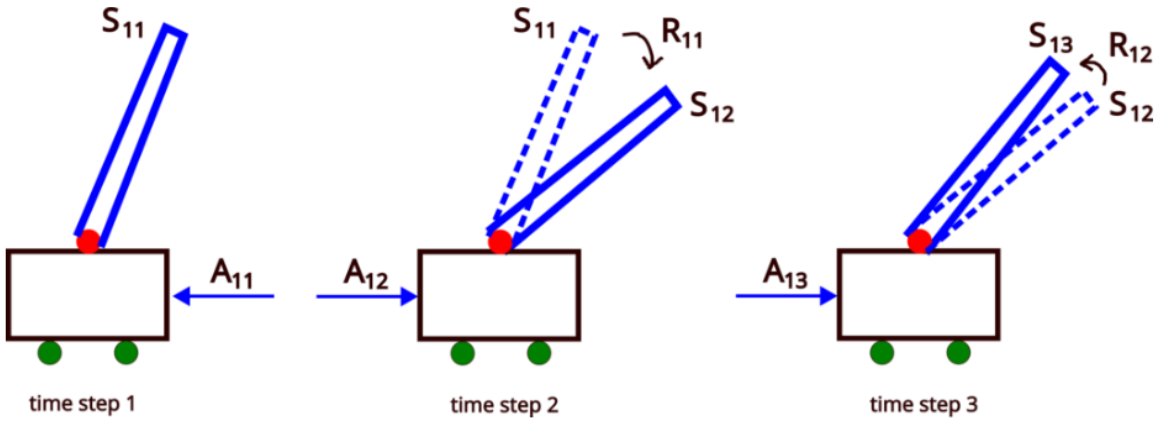


Figure 3: Illustration of the state transitions.

Nous commençons ensuite un deuxième épisode. L'état initial est S21. Nous calculons alors l'action A21, que nous appliquons à l'environnement. Cette action produit la

récompense R21, et l'environnement évolue vers l'état S22. Ensuite, dans l'état S22, nous répétons cette procédure jusqu'à atteindre l'état terminal (nous supposons que l'épisode compte étapes temporelles). Ce processus est répété pour le troisième, quatrième épisode, etc.

Nous stockons les données obtenues dans un replay buffer

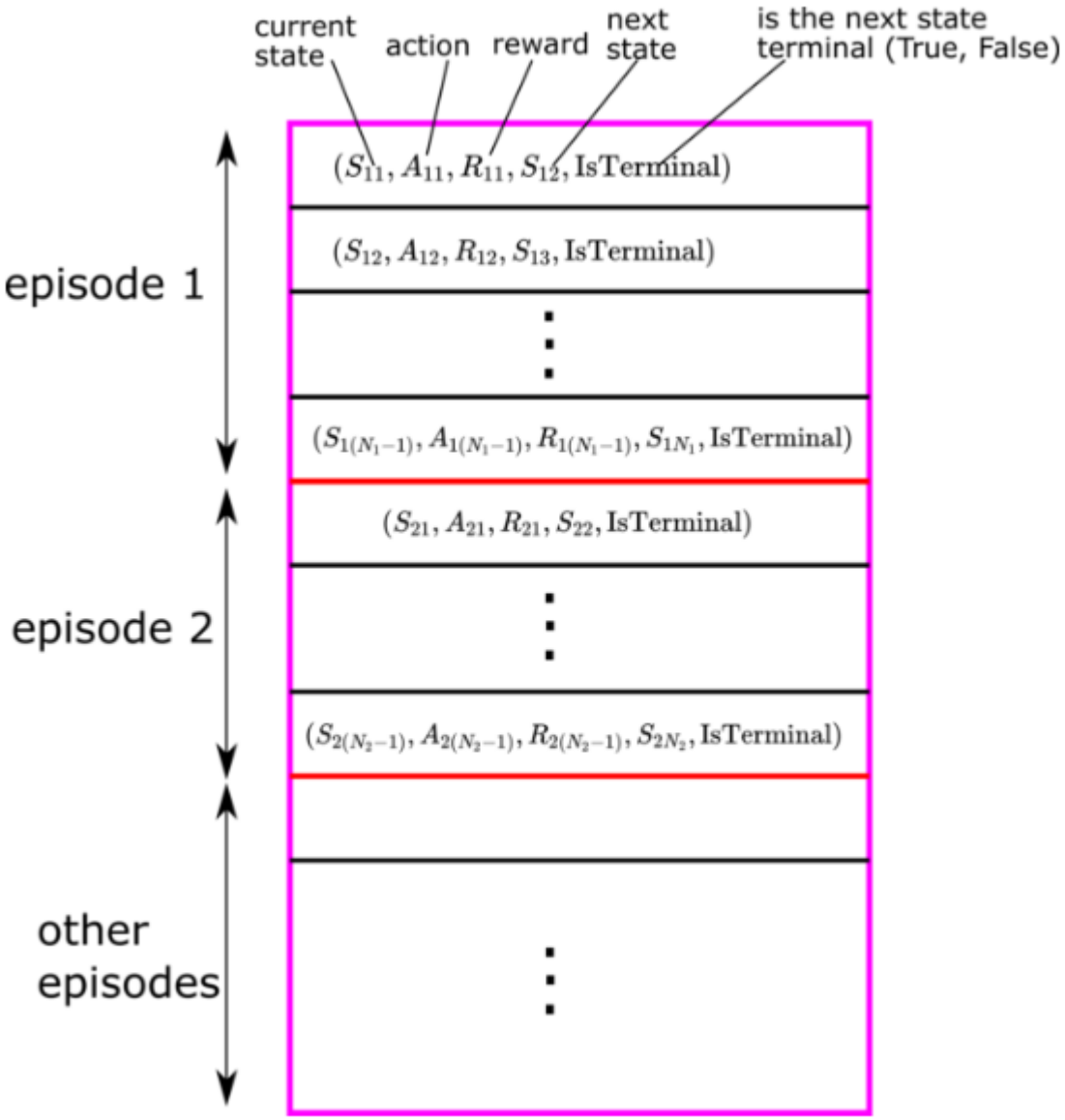


Figure 4: Illustration of the replay buffer

Pour créer ce replay buffer, nous stockons l'état actuel, l'action effectuée dans cet état, la récompense obtenue, l'état suivant et une variable booléenne « IsTerminal ». Cette variable indique si l'état suivant est terminal (OUI) ou non (FAUX). Les tuples sont empilés les uns sur les autres jusqu'à la fin de l'épisode. Ce processus est ensuite répété pour un certain nombre d'épisodes, générant une collection d'états, de récompenses et d'actions sur plusieurs épisodes.

Pour l'implémentation, une structure de données en file (queue) est utilisée pour gérer le replay buffer. Supposons que cette mémoire dispose de cellules pour enregistrer les tuples. Lorsqu'un nouveau tuple est enregistré, celui-ci est ajouté à la fin de la file, et en même temps, le premier tuple est supprimé. Cela garantit que nous avons toujours les données les plus récentes disponibles.

Lors de l'entraînement du réseau, nous n'utilisons pas le replay buffer complet (batch complet de données). Au lieu de cela, nous sélectionnons aléatoirement des tuples dans le buffer pour former un nouveau batch utilisé pour l'entraînement. Cela est illustré par le schéma ci-dessous.

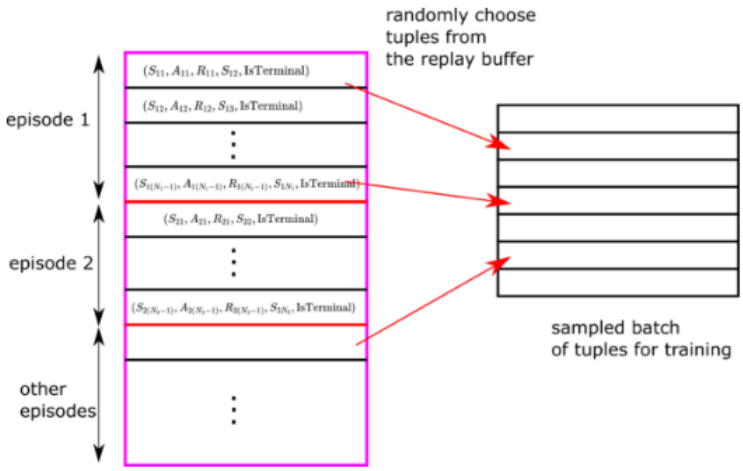


Figure 5: Random sampling from the replay buffer to form a batch of data for training.

Cette méthode est nécessaire car les états dans les tuples consécutifs du replay buffer peuvent être fortement corrélés. En conséquence, ces tuples successifs peuvent ne pas apporter d'informations significatives supplémentaires, ce qui ralentirait le processus d'apprentissage. Pour améliorer la vitesse d'apprentissage, nous échantillonnons aléatoirement des tuples et formons un nouveau batch de données moins corrélé mutuellement, accélérant ainsi le processus d'apprentissage.

Pendant l'entraînement du modèle Q-learning, nous utilisons en réalité deux réseaux :

Réseau Online :

- Ce réseau est également appelé réseau principal dans le code. Il est constamment mis à jour pendant le processus d'entraînement, d'où son nom de réseau online.
- Ce réseau est utilisé pour prédire les valeurs de fonction d'action et, une fois l'entraînement terminé, il est utilisé pour former la politique greedy.
- Les prédictions effectuées par ce réseau sont notées comme suit :

$$Q(S, A, \theta)$$

Où θ représente le vecteur des paramètres du réseau neuronal. L'algorithme d'entraînement met constamment à jour θ.

Réseau Target :

- Ce réseau est utilisé pour calculer les échantillons de sortie cible pour l'entraînement du réseau. Les prédictions effectuées par ce réseau, ainsi que les récompenses obtenues, sont utilisées pour former les sorties du processus d'entraînement.
- Les paramètres de ce réseau sont mis à jour moins fréquemment que ceux du réseau online (par exemple, toutes les 100 étapes de temps). Les valeurs prédites par ce réseau sont notées :

$$Q(S, A, \tilde{\theta})$$

Au début du processus d'entraînement, nous initialisons $\tilde{\theta} \leftarrow \theta$, c'est-à-dire que les deux réseaux sont identiques. Ensuite, après un certain nombre d'étapes (par exemple, 100 ou plus), nous mettons à jour $\tilde{\theta} \leftarrow \theta$, copiant ainsi les paramètres du réseau online au réseau target.

Fonction de coût

L'objectif est de minimiser la fonction de coût suivante :

$$\frac{1}{N} \sum_{i=1}^{N} \Big(y_i - Q(S_i, A_i, \theta) \Big)^2$$

Où le terme est calculé comme suit :

- Si l'état suivant $S_i^{'}$ n'est **pas terminal** :

$$y_i = R_i + \gamma Q_{\max}(S_i^{'}, \theta)$$
$$Q_{\max}(S_i^{'}) = \max_A Q(S_i^{'}, A, \tilde{\theta})$$

- Si l'état suivant $S_i^{'}$ est **terminal** :

$$y_i = R_i$$

Le paramètre représente le taux d'actualisation, et correspond aux paramètres du réseau target. Cette approche garantit que le processus d'apprentissage converge vers une solution stable tout en maintenant une certaine robustesse grâce à la séparation des rôles entre les deux réseaux.

Deep Q-Learning Algorithm

Pour compléter la mise en œuvre du processus d'apprentissage, nous devons introduire des explications importantes, telles que la formation de données pour la fonction de coût.

Le nombre représente le total des tuples dans le lot de données d'entraînement utilisé pour former la fonction de coût. Nous formons une seule entrée dans la somme ci-dessus en utilisant le tuple du lot d'entraînement $(S_i, A_i, R_i, S_i^{'}, \text{IsTerminal})$. Ici S_i, est l'état actuel, A_i est l'action effectuée dans cet état S_i , R_i est la récompense obtenue, et $S_i^{'}$ est l'état suivant produit par l'application de l'action A_i.

Le paramètre Y est le taux d'actualisation. La valeur de y_i est un réel représentant la sortie cible. L'entrée est l'état S_i . Si $S_i^{'}$ n'est pas un état terminal, alors est calculé comme suit :

$$y_i = R_i + \gamma Q_{\max}(S_i^{'}, \tilde{\theta})$$
$$Q_{\max}(S_i^{'}) = \max_A Q(S_i^{'}, A, \tilde{\theta})$$

$Q_{\max}(S_i^{'}, \tilde{\theta})$ Est calculé à l'aide du réseau target. Pour calculer cette valeur, nous introduisons simplement une entrée $S_i^{'}$ au réseau target et produisons les prédictions $Q(S_i^{'}, A = 0, \tilde{\theta})$ et $Q(S_i^{'}, A = 1, \tilde{\theta})$ par propagation $S_i^{'}$ avant à travers le réseau target. Ensuite, $Q_{\max}(S_i^{'}, \tilde{\theta})$ est simplement le maximum de ces deux prédictions.