TP 3 : Agent Deep Q-Network (DQN) dans un environnement GridWorld 4x4

Environnement `GridWorld`

- L'environnement représente une grille de 4x4 où l'agent commence dans le coin supérieur gauche `(0, 0)` et doit atteindre l'objectif situé en `(3, 3)`.
- Il y a aussi un obstacle à la position `(1, 1)` qui pénalise l'agent s'il y entre.
- Chaque action correspond à un déplacement dans la grille (haut, bas, gauche, droite).
- La méthode `step` permet à l'agent de se déplacer dans la grille et renvoie l'état, la récompense et si l'épisode est terminé ou non.

L'agent DQNAgent

- L'agent utilise un **réseau de neurones** pour approximer les valeurs Q pour chaque état et action.
- Il utilise la stratégie **ε-greedy** : dans un certain pourcentage des cas, l'agent explore de nouvelles actions au lieu d'exploiter les actions déjà apprises.
- Le réseau de neurones comporte deux couches cachées avec 24 neurones et utilise la fonction d'activation **ReLU** pour les couches cachées et **linear** pour la couche de sortie.
- Lorsqu'il interagit avec l'environnement, l'agent stocke les transitions (état, action, récompense, nouvel état, terminé) dans sa mémoire et met à jour le modèle à partir de ces transitions via la méthode replay.

Réduction de l'exploration (epsilon)

- Au début, l'agent explore beaucoup (epsilon = 1.0), mais au fur et à mesure de l'entraînement, il devient de plus en plus exploiteur (utilise son expérience pour maximiser la récompense).
- Cela est réalisé par la décadence exponentielle de epsilon dans la méthode replay

Les bibliothèques requises :

```
import numpy as np
import random
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from collections import deque
```

Les paramètres du jeu à définir au début:

```
# Paramètres du jeu
GRID_SIZE = 4
STATE_SIZE = GRID_SIZE * GRID_SIZE
ACTION_SIZE = 4  # Haut, Bas, Gauche, Droite
GAMMA = 0.9  # facteur de reduction
LEARNING_RATE = 0.01
EPSILON = 1.0  # Exploration initiale
EPSILON_MIN = 0.01
EPSILON_DECAY = 0.995
BATCH_SIZE = 32
MEMORY_SIZE = 2000
EPISODES = 1000
```

Définir les actions possibles :

```
# Déplacements possibles (Haut, Bas, Gauche, Droite)
MOVES = {
    0: (-1, 0),  # Haut
    1: (1, 0),   # Bas
    2: (0, -1),  # Gauche
    3: (0, 1)    # Droite
}
```

Définir la classe du jeu GridWorld :

```
class GridWorld:
    """Environnement GridWorld 4x4"""
    def __init__(self):
        self.grid_size = GRID_SIZE
        self.reset()

    def reset(self):
        """Réinitialise l'agent à la position de départ."""
        self.agent_pos = (0, 0)
        self.goal_pos = (3, 3)
        self.obstacle_pos = (1, 1)  # Une case d'obstacle
        return self.get_state()

    def get_state(self):
        """Retourne l'état sous forme d'un vecteur binaire."""
        state = np.zeros((GRID_SIZE, GRID_SIZE))
        state[self.agent_pos] = 1
        return state.flatten()

    def step(self, action):
        """Fait avancer l'agent et renvoie (nouvel état, récompense,
terminé)."""
        x, y = self.agent_pos
        dx, dy = MOVES[action]
        new_x, new_y = x + dx, y + dy

        # Vérifier les limites
        if 0 <= new_x < GRID_SIZE and 0 <= new_y < GRID_SIZE:
            self.agent_pos = (new_x, new_y)

        # Vérifier la récompense
        if self.agent_pos == self.goal_pos:
            return self.get_state(), 10, True  # Objectif atteint
        elif self.agent_pos == self.obstacle_pos:
            return self.get_state(), -5, False  # Obstacle
        else:
            return self.get_state(), -1, False  # Déplacement normal
```

La classe de l'agent Deep Q-Learning

```
class DQNAgent:
    """Agent DQN utilisant un réseau de neurones pour apprendre."""
    def __init__(self):
        self.state_size = STATE_SIZE
        self.action_size = ACTION_SIZE
        self.memory = deque(maxlen=MEMORY_SIZE)
        self.epsilon = EPSILON
        self.model = self._build_model()

    def _build_model(self):
        """Construit le réseau de neurones."""
        model = Sequential([
            Dense(24, activation='relu', input_shape=(self.state_size,)),
            Dense(24, activation='relu'),
            Dense(self.action_size, activation='linear')
        ])
        model.compile(loss="mse", optimizer=Adam(learning_rate=LEARNING_RATE))
        return model

    def remember(self, state, action, reward, next_state, done):
        """Stocke une expérience dans la mémoire."""
        self.memory.append((state, action, reward, next_state, done))

    def act(self, state):
        """Choisit une action en suivant une stratégie ε-greedy."""
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)  # Exploration
        q_values = self.model.predict(np.array([state]), verbose=0)
        return np.argmax(q_values[0])  # Exploitation

    def replay(self):
        """Entraîne le modèle avec des expériences passées."""
        if len(self.memory) < BATCH_SIZE:
            return
        batch = random.sample(self.memory, BATCH_SIZE)
        for state, action, reward, next_state, done in batch:
            target = self.model.predict(np.array([state]), verbose=0)[0]
            if done:
                target[action] = reward
            else:
                target[action] = reward + GAMMA *
np.max(self.model.predict(np.array([next_state]), verbose=0)[0])
            self.model.fit(np.array([state]), np.array([target]), epochs=1,
verbose=0)

        if self.epsilon > EPSILON_MIN:
            self.epsilon *= EPSILON_DECAY  # Réduction de l'exploration
```

Entraînement de l'agent

- L'agent est entraîné sur plusieurs épisodes (ici, 1000 épisodes). À chaque épisode, l'agent explore l'environnement, choisit des actions, reçoit des récompenses et met à jour son modèle.
- Après chaque épisode, la méthode replay est appelée pour entraîner le modèle sur les transitions stockées dans la mémoire.

Mise à jour des Q-valeurs

- Lorsque l'agent a effectué une action, il met à jour ses Q-valeurs estimées en fonction de la récompense reçue et des valeurs des actions futures. La cible pour la mise à jour de Q est calculée avec l'équation de Bellman :

```
target[action] = reward + GAMMA *
np.max(self.model.predict(np.array([next_state]), verbose=0)[0])
```

- Si l'épisode est terminé, la cible est simplement la récompense reçue.

- `target[action] = reward`: Si l'agent a atteint un état terminal (objectif atteint ou obstacle rencontré), la récompense seule est utilisée pour mettre à jour la Q-valeur.
- `self.model.fit(np.array([state]), np.array([target]), epochs=1, verbose=0)`
  : Le modèle est mis à jour pour réduire l'erreur entre la Q-valeur estimée et la Q-valeur cible pour l'état et l'action donnés.

```
# Entraînement de l'agent
env = GridWorld()
agent = DQNAgent()

for episode in range(EPISODES):
    state = env.reset()
    total_reward = 0
    for step in range(50):  # Limite de 50 déplacements
        action = agent.act(state)
        next_state, reward, done = env.step(action)
        agent.remember(state, action, reward, next_state, done)
        state = next_state
        total_reward += reward
        if done:
            break
    agent.replay()  # Entraîne le modèle
    print(f"Épisode {episode+1}/{EPISODES}, Score: {total_reward}, Epsilon:
{agent.epsilon:.4f}")

# Sauvegarde du modèle
agent.model.save("my_model.keras")
```

Devoir : Double Deep Q Network

Dans un réseau de neurones classique, les valeurs cibles dépendent de la sortie du modèle lui-même, ce qui peut créer des **problèmes de divergence** (car les erreurs de prédiction peuvent s'accumuler et perturber l'apprentissage). En d'autres termes, la mise à jour de la Q-valeur dépend de la prédiction faite par le même modèle qui est en train de s'entraîner, ce qui peut rendre l'apprentissage instable.

Comment cela fonctionne dans la pratique ?

Lors de l'entraînement, l'agent utilise le réseau principal pour choisir une action dans un état donné, puis utilise le réseau cible pour calculer la cible pour la mise à jour de la Q-valeur. Cela permet d'éviter que le réseau soit directement influencé par ses propres erreurs de prédiction pendant l'entraînement.

L'idée de **Double DQN** est de **séparer** les étapes de **sélection** et **évaluation** des actions :

Le réseau principal (online network) choisit l'action à jouer dans next_state

$$a' = \arg\max_{a'} Q_\theta(s', a')$$

Le réseau cible (target network) évalue la valeur de cette action choisie

$$Q_{\text{target}}(s, a) = r + \gamma Q_{\theta^-}(s', a')$$

Au lieu de prendre directement max Q(s', a') pour la mise à jour de Q :
On sélectionne l'action avec le réseau principal (online).
On évalue cette action avec le réseau cible (target).

Avantages du Double DQN :

Évite la surestimation des valeurs Q.
Stabilise l'apprentissage et améliore la convergence.
Meilleures performances que DQN, surtout pour des problèmes complexes

Travail à faire : Une implémentation de l'agent Double DQN avec le même exemple.