Implémentation de l'Environnement du Jeu Force 3 :

Dans cette section, nous expliquerons en détail comment l'environnement du jeu Force 3 a été implémenté en utilisant la bibliothèque Gym. Cet environnement permet à l'agent DQN d'interagir avec le jeu et d'apprendre à jouer en utilisant l'apprentissage par renforcement.

Initialisation de l'Environnement du Jeu :

La méthode **__init__** de la classe **Force3Env** joue un rôle essentiel dans l'initialisation de l'environnement du jeu Force 3. Cette méthode met en place plusieurs éléments clés nécessaires pour la gestion du jeu et l'interaction avec un agent d'apprentissage par renforcement.

Liste des Actions Valides :

Les actions valides définissent les actions que l'agent peut entreprendre dans le jeu. Dans le cas du jeu Force 3, trois types d'actions sont possibles : placer un pion rond, déplacer un pion rond et déplacer un carré.

Les actions pour le type 0 (Placer) sont définies pour les cases du plateau allant de 0 à 3 et de 5 à 8. (Case centrale exclue)

Les actions pour le type 1 (Déplacer un pion rond) sont générées pour toutes les combinaisons possibles de positions de départ et d'arrivée sur le plateau.

Les actions pour le type 2 (Déplacer un carré) sont générées pour toutes les combinaisons possibles de positions de départ, d'arrivée et de décision de déplacement de deux carrés.

Espace d'Observation :

L'espace d'observation de l'environnement du jeu Force 3 est défini comme une boîte (box) permettant de représenter l'état actuel du plateau de jeu. Cet espace d'observation est de forme (9,) et peut contenir des valeurs entières allant de -1 à 2. Chaque composant de cet espace correspond à une case du plateau, et les valeurs possibles représentent les différents états qu'une case peut prendre :

- **-1** : La case est occupée par un pion rond.
- **0** : La case est vide.
- **1** : La case est occupée par un pion rond.
- **2** : La case est un carré inoccupé.

Cet espace d'observation permet à l'agent d'accéder à l'information sur l'état du jeu, ce qui est essentiel pour prendre des décisions stratégiques tout au long de la partie.

Initialisation de l'Environnement et de l'Agent :

Pour mettre en place l'environnement du jeu, nous créons une instance de la classe **Force3**, qui représente le jeu lui-même. Cette instance, nommée **self.force3**, est utilisée pour suivre l'état actuel du jeu et appliquer les règles spécifiques à Force 3.

De plus, nous initialisons l'agent en déterminant le joueur actuel, qui peut être soit le joueur 1 (1) soit le joueur 2 (-1). Cette information est stockée dans la variable **self.current_player**, et elle permet à l'agent de savoir quel joueur il représente au début de chaque partie.

Réinitialisation du Jeu :

La méthode **reset** est essentielle pour réinitialiser l'état du jeu à son point de départ, permettant ainsi de commencer une nouvelle partie. Elle effectue les actions suivantes :

1. Appel à la méthode **reset** de l'instance **self.force3**, ce qui remet le plateau de jeu dans son état initial. Cette étape garantit un point de départ propre pour chaque partie.
2. Changement du joueur actuel en inversant son rôle. Si l'agent était initialement le joueur 1, il devient le joueur 2, et vice versa. Cette alternance entre les joueurs est nécessaire pour assurer que chaque partie commence avec un joueur différent.
3. La méthode retourne l'état initial du jeu sous forme d'un tableau NumPy, préalablement remodelé pour être utilisé comme espace d'observation. Cet état initial est essentiel pour que l'agent puisse commencer à prendre des décisions dans la nouvelle partie.

Conversion d'Action en Tuple :

La méthode **convert_to_action_tuple** a pour rôle de convertir une action donnée en un tuple d'informations exploitables par l'environnement. Elle effectue les opérations suivantes :

1. Prend en entrée l'action sous forme d'un tuple comprenant **action_type_num** (le type d'action, 0 pour placer un pion, 1 pour déplacer un pion rond, 2 pour déplacer un carré), **start_pos** (la position de départ), **target_pos** (la position cible), et **move_two_decision** (la décision de déplacer deux carrés en un seul coup).
2. Utilise un dictionnaire **action_mapping** pour convertir le **action_type_num** en une chaîne de caractères descriptive, comme "place_round," "move_round," ou "move_square." Cette étape facilite la compréhension de l'action effectuée.
3. Si l'action est de type "place_round," calcule les coordonnées de la case cible en divisant **target_pos** par 3 pour obtenir les lignes et les colonnes correspondantes.
4. Si l'action n'est pas de type "place_round," calcule également les coordonnées de départ (**start_row** et **start_col**) et les coordonnées de la case cible (**target_row** et **target_col**) en utilisant les positions fournies.
5. Construit et retourne un tuple complet comprenant le type d'action, les coordonnées de départ, les coordonnées de la case cible, et la décision de déplacer deux carrés le cas échéant.

Exécution d'une Étape d'Action :

La méthode **step** est responsable de l'exécution d'une action dans l'environnement et de la mise à jour de l'état du jeu. Voici les étapes clés effectuées par cette méthode :

1. Appel de la méthode **convert_to_action_tuple(action)** pour obtenir les informations détaillées sur l'action à effectuer, y compris le type d'action, les coordonnées de départ, les coordonnées de la case cible, et la décision de déplacer deux carrés le cas échéant.
2. Appel à la méthode **self.force3.step(...)** pour effectuer l'action dans le jeu Force 3. Cette méthode renvoie le nouvel état du plateau de jeu, une indication sur la fin de la partie (**game_over**), le vainqueur de la partie (**winner**), un indicateur de réussite (**success**), et d'autres informations non utilisées dans cette étape.
3. Calcul de la récompense en appelant la méthode **self.calculate_reward(...)**, en utilisant les informations sur le nouvel état du jeu, la fin de la partie, le vainqueur, la réussite, ainsi que les coordonnées de la case cible.
4. Mise à jour de la variable **done** en fonction de l'indication de fin de partie (**game_over**) : si la partie est terminée, **done** est définie à **True**, sinon, elle reste à **False**.
5. Création d'un dictionnaire **info** pour stocker des informations supplémentaires, comme le vainqueur de la partie.
6. La méthode retourne un tableau NumPy représentant le nouvel état du plateau de jeu (remodelé pour être compatible avec l'espace d'observation), la récompense obtenue, l'indicateur de fin de partie, et le dictionnaire d'informations **info**.

Calcul de Récompense (calculate_reward) :

La méthode **calculate_reward** est responsable de la détermination de la récompense ou de la pénalité attribuée à l'agent à chaque étape du jeu. Elle prend en compte diverses conditions et stratégies du jeu pour calculer une valeur de récompense appropriée. Voici un résumé de ses éléments clés :

- Elle définit plusieurs constantes pour les récompenses de base, telles que **WIN_REWARD** pour la récompense de victoire, **LOSE_PENALTY** pour la pénalité de défaite, **INVALID_MOVE** pour la pénalité de mouvement invalide, **NEUTRAL_REWARD** pour une récompense neutre, **BLOCK_REWARD** pour la récompense de blocage, **OPPORTUNITY_FOR_GAIN** pour la récompense d'opportunité, et **MOVE_TWO_PENALTY** pour la pénalité de déplacement de deux carrés non judicieux.
- Elle commence par vérifier si le jeu est terminé (**game_over**). Si c'est le cas, elle attribue une récompense en fonction du vainqueur de la partie. Si l'agent gagne, il reçoit **WIN_REWARD**, sinon il subit **LOSE_PENALTY**.
- Ensuite, elle examine si l'agent a effectué un mouvement de deux carrés lorsqu'il n'est pas le second mouvement d'un double coup (**self.force3.is_second_move_of_double**). Elle évalue si ce mouvement est stratégiquement judicieux en vérifiant s'il présente à la fois une opportunité (**is_opportunity**) et s'il bloque le joueur adverse (**is_blocking_move**). En fonction de ces conditions, elle attribue des récompenses (**OPPORTUNITY_FOR_GAIN** et **BLOCK_REWARD**) ou une pénalité (**MOVE_TWO_PENALTY**) appropriées.
- Si l'agent n'effectue pas un mouvement de deux carrés et qu'il est son tour (**self.force3.current_player == self.current_player**), elle évalue à nouveau les opportunités et les blocages pour attribuer des récompenses en fonction de ces conditions.

- Si aucune des conditions ci-dessus n'est remplie et que le mouvement n'a pas réussi (**not success**), elle attribue une pénalité de mouvement invalide (**INVALID_MOVE**).
- Dans tous les autres cas, elle renvoie une récompense neutre (**NEUTRAL_REWARD**).

Ces calculs de récompense sont essentiels pour guider l'apprentissage de l'agent et l'encourager à adopter des stratégies gagnantes tout en évitant les mouvements inutiles ou contre-productifs.
Évaluation d'Opportunités et Blocages (is_blocking_move, is_opportunity, etc.)
Les méthodes associées, telles que **is_blocking_move**, **is_opportunity**, **check_line_for_opportunity**, **check_column_for_opportunity**, **check_diagonal_for_opportunity**, et **check_for_opportunity_in_sequence**, sont utilisées pour évaluer les opportunités et les blocages dans le jeu. Elles sont utilisées dans le calcul de la récompense.

- **is_blocking_move** détermine si un mouvement bloque efficacement l'adversaire en vérifiant les coins et les cases centrales des côtés du plateau.
- **is_opportunity** évalue si une case donnée présente une opportunité de gagner en vérifiant les lignes, les colonnes et les diagonales, en fonction de sa position.
- Les autres méthodes (**check_line_for_opportunity**, **check_column_for_opportunity**, **check_diagonal_for_opportunity**, et **check_for_opportunity_in_sequence**) sont utilisées pour décomposer ces vérifications en opérations plus spécifiques.

L’apprentissage par renforcement profond (Deep Q-Learning) :

Initialisation des bibliothèques et de l'environnement :

```
import time
from collections import deque, namedtuple
import numpy as np
import tensorflow as tf
import random
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.losses import MSE
from tensorflow.keras.optimizers.legacy import Adam
from force3env1 import Force3Env
```

Dans cette section, nous importons les bibliothèques et les outils nécessaires à l'implémentation de l'apprentissage par renforcement de notre agent. Ces importations comprennent des bibliothèques standard telles que **time**, **numpy**, **random**, ainsi que des bibliothèques spécifiques à l'apprentissage automatique, notamment TensorFlow. Ces outils seront utilisés pour la manipulation des données, la construction du modèle d'apprentissage, la gestion des épisodes d'entraînement et l'interaction avec l'environnement du jeu Force 3.

Initialisation des paramètres nécessaires pour l’apprentissage :

```
SEED = 0
MINIBATCH_SIZE = 64
TAU = 1e-3
E_DECAY = 0.99
E_MIN = 0.01
```

✓ 0.0s Python

Dans cette partie, nous définissons plusieurs paramètres importants pour notre algorithme d'apprentissage par renforcement DQN (Deep Q-Network) :

1. **SEED**: Cette valeur est utilisée comme graine (seed) pour le générateur de nombres pseudo-aléatoires, ce qui permet de rendre les expériences reproductibles.
2. **MINIBATCH_SIZE**: Il s'agit de la taille du mini-lot (mini-batch) d'échantillons d'expérience que nous utiliserons lors de la mise à jour du réseau de neurones. Cela affecte la stabilité de l'apprentissage.
3. **TAU**: Le paramètre de mise à jour souple (soft update) est utilisé lors de la mise à jour du réseau cible. Il contrôle la vitesse à laquelle les poids du réseau cible sont ajustés vers les poids du réseau principal. Une valeur plus petite signifie une mise à jour plus lente.
4. **E_DECAY**: Ce taux de décroissance ε (epsilon) est utilisé pour réduire progressivement l'exploration de l'agent au fil du temps. Cela permet à l'agent de passer d'une politique exploratoire (ε-greedy) à une politique plus déterministe à mesure qu'il gagne de l'expérience.
5. **E_MIN**: Il s'agit de la valeur minimale d'ε que l'agent atteindra après une certaine période d'entraînement. Cela garantit que même à un stade avancé de l'apprentissage, l'agent continuera à explorer un peu pour éviter de rester coincé dans des comportements sous-optimaux.

Extraction d'expériences de la mémoire tampon :

```python
def get_experiences(memory_buffer):
    experiences = random.sample(memory_buffer, k=MINIBATCH_SIZE)
    states = tf.convert_to_tensor(
        np.array([e.state for e in experiences if e is not None]), dtype=tf.float32
    )
    actions = tf.convert_to_tensor(
        np.array([e.action for e in experiences if e is not None]), dtype=tf.float3
    )
    rewards = tf.convert_to_tensor(
        np.array([e.reward for e in experiences if e is not None]), dtype=tf.float3
    )
    next_states = tf.convert_to_tensor(
        np.array([e.next_state for e in experiences if e is not None]), dtype=tf.fl
    )
    done_vals = tf.convert_to_tensor(
        np.array([e.done for e in experiences if e is not None]).astype(np.uint8),
        dtype=tf.float32,
    )
    return (states, actions, rewards, next_states, done_vals)
```

✓ 0.0s Python

Cette fonction, **get_experiences(memory_buffer)**, est responsable de l'extraction d'un échantillon d'expériences à partir de la mémoire tampon de l'agent. Voici ce qu'elle fait :

- Elle prend en entrée **memory_buffer**, qui est la mémoire tampon de l'agent, où il stocke ses expériences passées.
- À l'aide de la fonction **random.sample**, elle sélectionne aléatoirement un mini-lot (mini-batch) de taille **MINIBATCH_SIZE** d'expériences à partir de la mémoire tampon. Cela garantit que l'apprentissage est basé sur un échantillon représentatif d'expériences passées.
- Elle extrait ensuite les états, les actions, les récompenses, les états suivants et les valeurs booléennes de terminaison (**done**) à partir des expériences sélectionnées.
- Les données extraites sont converties en tenseurs TensorFlow pour être utilisées dans l'apprentissage du réseau de neurones.

En résumé, cette fonction joue un rôle essentiel dans la préparation des données d'apprentissage à partir de la mémoire tampon de l'agent, en garantissant que les données sont structurées et prêtes à être utilisées pour la mise à jour du réseau de neurones DQN.

Mise à jour de la valeur d'epsilon pour la politique ε-greedy :

```python
def get_new_eps(epsilon):
    return max(E_MIN, epsilon * E_DECAY)
```

✓ 0.0s Python

L'un des aspects essentiels de l'apprentissage par renforcement est la façon dont l'agent explore son environnement. Dans notre cas, nous utilisons une stratégie ε-greedy, où ε

représente la probabilité d'exploration et (1-ε) la probabilité d'exploitation. Au fil du temps, nous ajustons la valeur d'epsilon pour favoriser l'exploration au début et l'exploitation des connaissances acquises plus tard dans l'apprentissage.

Pour ce faire, nous utilisons la fonction get_new_eps(epsilon), qui met à jour la valeur d'epsilon selon une décroissance exponentielle. Cette fonction permet de réduire progressivement l'exploration au fur et à mesure que l'agent accumule de l'expérience et devient plus compétent dans le jeu.

En pratique, cela signifie que l'agent commence par explorer de manière aléatoire avec une probabilité élevée d'epsilon, puis réduit progressivement cette probabilité au fil du temps. Cette approche permet à l'agent d'explorer de manière plus efficace au début de l'apprentissage, tout en exploitant davantage ses connaissances lorsque son niveau de confiance augmente.

L'intégration de cette stratégie d'exploration dans notre algorithme d'apprentissage contribue à l'amélioration des performances de notre agent et à son adaptation à différentes situations de jeu. Cela fait partie des techniques clés pour résoudre avec succès le jeu Force 3 grâce à l'apprentissage par renforcement.

Mise à jour des Réseaux Cibles :

```python
update_target_network(q_network, target_q_network):
for target_weights, q_net_weights in zip(
    target_q_network.weights, q_network.weights
):
    target_weights.assign(TAU * q_net_weights + (1.0 - TAU) * target_weights)
```

✓ 0.0s Python

Nous utilisons la fonction update_target_network pour mettre à jour les réseaux cibles dans notre algorithme d'apprentissage. Cette mise à jour douce des poids des réseaux cibles est cruciale pour la stabilité de l'apprentissage. Elle permet au réseau cible de suivre progressivement les changements du réseau principal en utilisant un paramètre de douceur (TAU), favorisant ainsi une convergence plus rapide et des performances améliorées de notre agent.

Vérification des Conditions de Mise à Jour :

```python
def check_update_conditions(t, num_steps_upd, memory_buffer):
    if (t + 1) % num_steps_upd == 0 and len(memory_buffer) > MINIBATCH_SIZE:
        return True
    else:
        return False
```

✓ 0.0s Python

La fonction **check_update_conditions** est utilisée pour déterminer si les conditions sont remplies pour mettre à jour notre réseau de neurones de l'apprenant. L'apprentissage par renforcement implique généralement des mises à jour périodiques du réseau de l'agent. Ces mises à jour sont déclenchées après un certain nombre d'étapes d'interaction avec l'environnement, spécifié par **num_steps_upd**. De plus, la mise à jour n'a lieu que si notre mémoire tampon de l'expérience contient suffisamment de données, ce qui est vérifié en comparant sa taille à la taille du mini-lot spécifiée (**MINIBATCH_SIZE**).

Paramètres de l'Apprentissage par Renforcement :

```python
# Définissez la graine aléatoire pour TensorFlow
tf.random.set_seed(SEED)

MEMORY_SIZE = 100_000     # size of memory buffer
GAMMA = 0.99              # discount factor
ALPHA = 1e-3              # learning rate
NUM_STEPS_FOR_UPDATE = 2  # perform a learning update every C time steps
```

Python

Dans cette section, nous définissons plusieurs paramètres importants pour l'apprentissage par renforcement, qui affectent le comportement de notre agent et la manière dont il apprend.

- **MEMORY_SIZE**: C'est la taille de la mémoire tampon utilisée pour stocker les expériences passées de l'agent. Une mémoire tampon de plus grande taille permet à l'agent de conserver un historique plus long de ses interactions avec l'environnement.
- **GAMMA**: Le facteur de réduction est utilisé pour pondérer les récompenses futures par rapport aux récompenses immédiates. Il détermine l'importance accordée aux récompenses à long terme par rapport aux récompenses à court terme.
- **ALPHA**: Le taux d'apprentissage est un paramètre qui contrôle la taille des mises à jour de nos valeurs Q lors de l'apprentissage. Un taux d'apprentissage plus élevé peut accélérer l'apprentissage, mais il peut également le rendre instable.
- **NUM_STEPS_FOR_UPDATE**: C'est le nombre d'étapes d'interaction avec l'environnement que l'agent effectuera avant de mettre à jour ses valeurs Q et d'apprendre à partir de ses expériences passées. Une mise à jour périodique permet à l'agent de stabiliser son apprentissage et d'améliorer sa politique au fil du temps.

Environnement et Espace d'État :

```python
env = Force3Env()

state_size = env.observation_space.shape
num_actions = len(env.valid_actions)


print('State Shape:', state_size)
print('Number of actions:', num_actions)
```

✓ 0.0s Python

```
State Shape: (9,)
Number of actions: 251
```

Nous utilisons l'environnement **Force3Env()** pour créer un environnement de jeu Force 3. Nous examinons ensuite les caractéristiques de cet environnement.

- **state_size**: C'est la forme de l'espace d'état de notre environnement. Il représente la dimension de l'observation que l'agent reçoit de l'environnement. Dans notre cas, la forme de l'espace d'état est donnée par **env.observation_space.shape**, ce qui nous donne la forme spécifique de cet espace d'état.
- **num_actions**: C'est le nombre total d'actions valides que l'agent peut choisir dans cet environnement. Nous l'obtenons en comptant le nombre d'actions valides dans **env.valid_actions**.

Ces informations sont essentielles pour configurer l'architecture du réseau de neurones de notre agent d'apprentissage par renforcement.

Architecture du Réseau de Neurones :

```python
q_network = Sequential([
    Dense(128, activation='relu', input_shape=state_size),
    Dense(256, activation='relu'),
    Dense(512, activation='relu'),
    Dense(num_actions, activation='linear')
])

target_q_network = Sequential([
    Dense(128, activation='relu', input_shape=state_size),
    Dense(256, activation='relu'),
    Dense(512, activation='relu'),
    Dense(num_actions, activation='linear')
])

optimizer = Adam(learning_rate=ALPHA)
```

✓ 0.0s Python

Nous avons défini l'architecture du réseau de neurones pour notre agent d'apprentissage par renforcement. Le réseau de neurones comprend deux parties : le réseau principal (Q-Network) et le réseau cible (Target Q-Network). Ces réseaux sont utilisés pour estimer les valeurs Q des différentes actions dans notre environnement.

- **q_network**: Il s'agit du réseau principal, qui prend l'observation de l'environnement en entrée et produit les estimations des valeurs Q pour chaque action valide en sortie. L'architecture de ce réseau comprend plusieurs couches denses (fully connected) avec des fonctions d'activation ReLU, suivies d'une couche de sortie linéaire avec un nombre d'unités égal au nombre d'actions valides (**num_actions**).
- **target_q_network**: Il s'agit du réseau cible, qui suit une architecture similaire à celle du réseau principal. Le réseau cible est utilisé pour estimer les valeurs Q cibles que nous cherchons à atteindre lors de l'apprentissage par renforcement. Il est également mis à jour périodiquement en utilisant une opération de doux remplacement.
- **optimizer**: Nous utilisons l'optimiseur Adam avec un taux d'apprentissage (**learning_rate**) défini par **ALPHA** pour mettre à jour les poids du réseau lors de l'apprentissage.

Ces réseaux de neurones sont cruciaux pour l'apprentissage par renforcement, car ils nous permettent d'estimer les valeurs Q et d'ajuster nos actions en conséquence pour maximiser les récompenses cumulatives futures.

Expérience de l'Agent :

```
experience = namedtuple("Experience", field_names=["state", "action", "reward", "ne
```

✓ 0.0s Python

Nous utilisons une structure de données appelée **namedtuple** nommée "Experience" pour stocker les informations relatives à chaque expérience de l'agent. Cette structure est utilisée pour stocker des informations sur l'état actuel de l'environnement, l'action effectuée par l'agent, la récompense reçue, l'état suivant de l'environnement après avoir effectué l'action, et un indicateur pour savoir si l'épisode est terminé (**done**).

- **state**: Il s'agit de l'état actuel de l'environnement. Cet état est capturé au moment où l'agent prend une décision.
- **action**: Cette composante enregistre l'action effectuée par l'agent à partir de l'état actuel.
- **reward**: La récompense reçue par l'agent en raison de l'action qu'il a effectuée à partir de l'état actuel.
- **next_state**: Il s'agit de l'état de l'environnement après que l'action ait été effectuée. Cela reflète comment l'environnement a évolué en réponse à l'action de l'agent.
- **done**: Un indicateur binaire qui signale si l'épisode est terminé ou non. Il peut être utilisé pour déterminer si l'agent a atteint un état terminal dans l'environnement.

L'utilisation de cette structure de données permet de stocker de manière organisée et efficace les expériences passées de l'agent dans une mémoire tampon, ce qui est essentiel pour l'apprentissage par renforcement basé sur l'expérience.

Calcul de la Perte (Loss) dans l'Apprentissage par Renforcement :

```python
def compute_loss(experiences, gamma, q_network, target_q_network):
    states, actions, rewards, next_states, done_vals = experiences

    action_types, start_pos, target_pos, _ = tf.split(actions, num_or_size_splits=4, axis=1)

    max_qsa = tf.reduce_max(target_q_network(next_states), axis=-1)

    y_targets = rewards + (1 - done_vals) * gamma * max_qsa

    # Convertir les composants d'action en indices uniques
    N = 9  # Nombre de valeurs possibles pour start_pos et target_pos
    # Convertir les composants d'action en indices uniques
    # Pour les actions de type 0 (Placer), les indices vont de 0 à 8
    # Pour les actions de type 1 (Déplacer un pion rond), les indices vont de 9 à 89
    # Pour les actions de type 2 (Déplacer un carré), les indices vont de 90 à 251
    action_indices = action_types * (N**2 + N**2) + start_pos * N + target_pos
    # Pour les actions de type 0, les indices vont de 0 à 8
    action_indices = tf.where(action_types == 0, target_pos, action_types)
    # Pour les actions de type 1, les indices vont de 9 à 89 (9 + 9 * 9)
    action_indices = tf.where(action_types == 1, 9 + start_pos * N + target_pos, action_indices)
    # Pour les actions de type 2, les indices vont de 90 à 251 (90 + 9 * 9 * 2)
    action_indices = tf.where(action_types == 2, 90 + start_pos * N + target_pos * 2, action_indices)
    action_indices = tf.squeeze(action_indices, axis=-1)  # Pour enlever une dimension inutile


    q_values = q_network(states)
    q_values = tf.gather_nd(q_values, tf.stack([tf.range(q_values.shape[0]),
                                                tf.cast(action_indices, tf.int32)], axis=1))

    loss = MSE(q_values, y_targets)

    return loss
```
✓ 0.0s

Cette fonction, **compute_loss**, est utilisée pour calculer la perte (loss) lors de l'apprentissage par renforcement. Elle prend en entrée un mini-lot (mini-batch) d'expériences passées de l'agent, ainsi que d'autres paramètres tels que le facteur de réduction gamma, le réseau Q principal (**q_network**), et le réseau cible Q (**target_q_network**).
Voici comment la fonction fonctionne en résumé :

1. Les expériences sont décomposées en leurs composantes individuelles : états, actions, récompenses, états suivants et indicateurs de terminaison.
2. Les actions sont décomposées en leurs composantes : type d'action, position de départ, position cible et décisions de déplacement.
3. La valeur maximale de Q pour les états suivants est calculée en utilisant le réseau cible Q (**target_q_network**).
4. Les cibles de valeur (**y_targets**) sont calculées en fonction des récompenses et de la valeur maximale de Q des états suivants, en tenant compte de l'indicateur de terminaison de l'épisode.
5. Les indices uniques correspondant aux actions sont calculés en fonction de leur type et de leurs composantes, pour permettre de sélectionner les valeurs Q appropriées à partir du réseau principal Q (**q_network**).
6. Les valeurs Q correspondantes sont extraites du réseau principal Q et comparées aux cibles de valeur (**y_targets**) pour calculer la perte (loss).
7. La perte est calculée en utilisant la perte quadratique moyenne (MSE - Mean Squared Error) entre les valeurs Q prédites et les cibles de valeur.

En résumé, cette fonction est essentielle pour l'apprentissage par renforcement, car elle permet au réseau neuronal d'apprendre à estimer les valeurs Q correctes et à s'ajuster en conséquence pour améliorer la politique de l'agent.

Apprentissage de l'Agent :

```python
@tf.function
def agent_learn(experiences, gamma):
    with tf.GradientTape() as tape:
        loss = compute_loss(experiences, gamma, q_network, target_q_network)

    gradients = tape.gradient(loss, q_network.trainable_variables)

    optimizer.apply_gradients(zip(gradients, q_network.trainable_variables))

    update_target_network(q_network, target_q_network)
```

✓ 0.0s Python

Cette fonction effectue une étape d'apprentissage en calculant la perte, en calculant les gradients de la perte par rapport aux poids du réseau, en appliquant ces gradients pour mettre à jour le réseau principal Q, et en mettant à jour le réseau cible Q pour stabiliser l'apprentissage de l'agent.

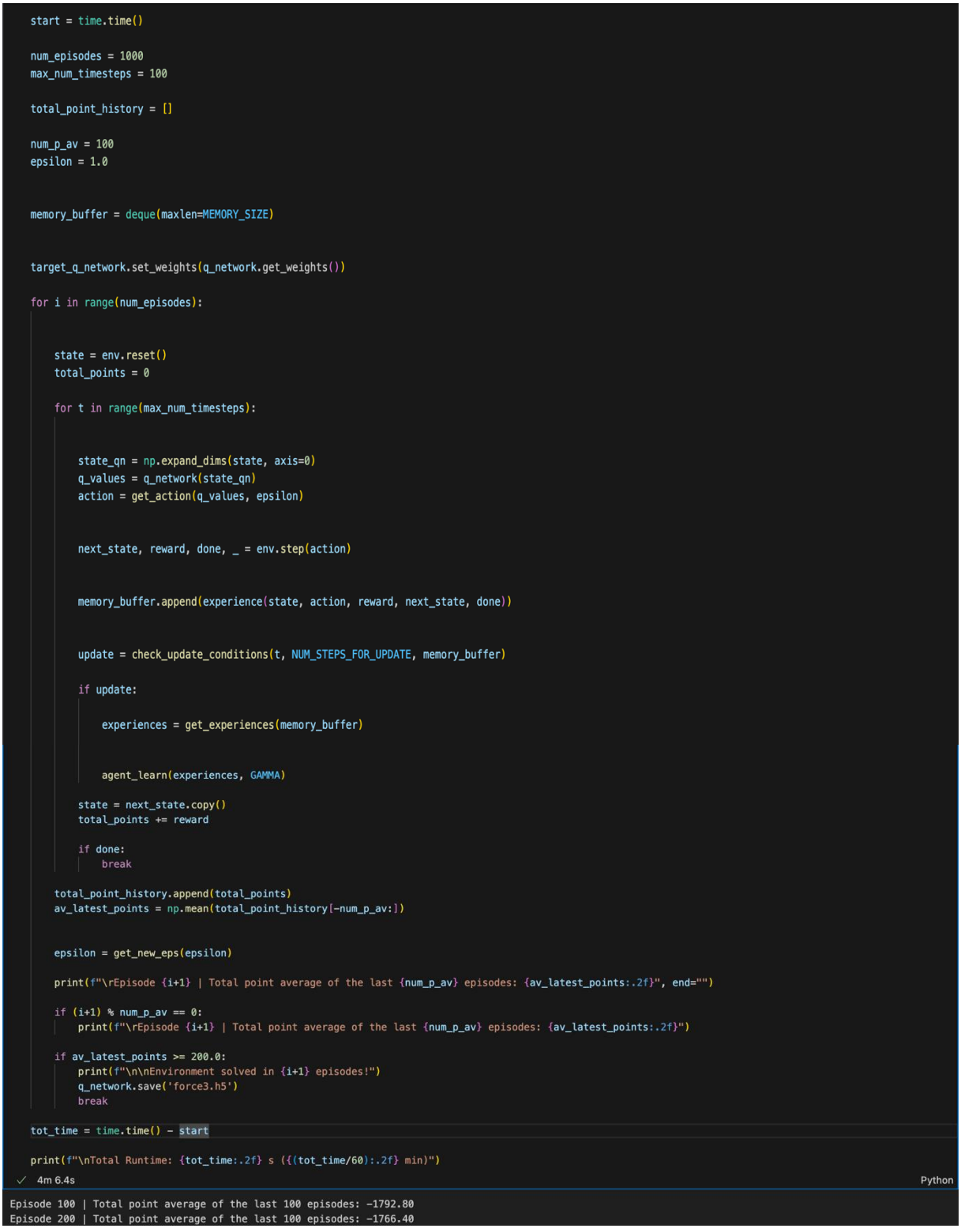
Stratégie de Sélection d'Action :

```python
def get_action(q_values, epsilon=0.0):
    if random.random() > epsilon:
        # Exploiter: choisir la meilleure action basée sur les valeurs Q prédites
        return env.valid_actions[np.argmax(q_values.numpy())]
    else:
        # Explorer: choisir une action au hasard
        return random.choice(env.valid_actions)
```

✓ 0.0s Python

La fonction get_action met en œuvre une stratégie d'exploration-exploitation en fonction du paramètre epsilon. Elle permet à l'agent de choisir entre exploiter ce qu'il sait être la meilleure action actuelle ou explorer de nouvelles actions de manière aléatoire. Cette stratégie est essentielle pour l'apprentissage efficace de l'agent dans un environnement d'apprentissage par renforcement.

Apprentissage par Renforcement Profond (Deep Reinforcement Learning) :



```
start = time.time()

num_episodes = 1000
max_num_timesteps = 100

total_point_history = []

num_p_av = 100
epsilon = 1.0


memory_buffer = deque(maxlen=MEMORY_SIZE)


target_q_network.set_weights(q_network.get_weights())

for i in range(num_episodes):


    state = env.reset()
    total_points = 0

    for t in range(max_num_timesteps):


        state_qn = np.expand_dims(state, axis=0)
        q_values = q_network(state_qn)
        action = get_action(q_values, epsilon)


        next_state, reward, done, _ = env.step(action)


        memory_buffer.append(experience(state, action, reward, next_state, done))


        update = check_update_conditions(t, NUM_STEPS_FOR_UPDATE, memory_buffer)

        if update:

            experiences = get_experiences(memory_buffer)


            agent_learn(experiences, GAMMA)

        state = next_state.copy()
        total_points += reward

        if done:
            break

    total_point_history.append(total_points)
    av_latest_points = np.mean(total_point_history[-num_p_av:])


    epsilon = get_new_eps(epsilon)

    print(f"\rEpisode {i+1} | Total point average of the last {num_p_av} episodes: {av_latest_points:.2f}", end="")

    if (i+1) % num_p_av == 0:
        print(f"\rEpisode {i+1} | Total point average of the last {num_p_av} episodes: {av_latest_points:.2f}")

    if av_latest_points >= 200.0:
        print(f"\n\nEnvironment solved in {i+1} episodes!")
        q_network.save('force3.h5')
        break

tot_time = time.time() - start

print(f"\nTotal Runtime: {tot_time:.2f} s ({(tot_time/60):.2f} min)")
```

✓ 4m 6.4s Python

```
Episode 100 | Total point average of the last 100 episodes: -1792.80
Episode 200 | Total point average of the last 100 episodes: -1766.40
```

Dans cette partie du code, nous mettons en œuvre l'apprentissage par renforcement profond (DRL) à l'aide de l'algorithme DQN (Deep Q-Network) pour résoudre l'environnement du jeu "Force3".

Voici une description concise de ce qui se passe dans cette section du code :

1. Nous initialisons des variables et des hyperparamètres tels que le nombre d'épisodes, la taille de la mémoire tampon, le taux d'apprentissage, le taux de décroissance epsilon, etc.
2. Nous créons une mémoire tampon (memory buffer) de capacité spécifiée pour stocker les expériences passées de l'agent. Cela nous permet de construire un ensemble d'expériences qui seront utilisées pour l'apprentissage.
3. Nous initialisons deux réseaux de neurones : **q_network** (le réseau principal) et **target_q_network** (le réseau cible). Ces réseaux sont utilisés pour estimer les valeurs Q des actions dans un état donné.
4. Nous initialisons les poids de **target_q_network** pour qu'ils soient identiques à ceux de **q_network** au départ. Ces poids seront ensuite mis à jour périodiquement.
5. Nous commençons à entraîner l'agent sur un certain nombre d'épisodes. Pour chaque épisode, l'agent interagit avec l'environnement en choisissant des actions basées sur une politique ε-greedy (exploration-exploitation).
6. L'agent stocke chaque expérience (état, action, récompense, état suivant, done) dans la mémoire tampon.
7. L'agent met à jour le réseau **q_network** en effectuant une étape de descente de gradient (apprentissage) uniquement à chaque **NUM_STEPS_FOR_UPDATE** étapes temporelles et si la mémoire tampon contient suffisamment d'expériences.
8. Nous surveillons les performances de l'agent en calculant la somme totale des récompenses pour chaque épisode (**total_points**) et en calculant la moyenne mobile de ces totaux (**av_latest_points**) sur les 100 épisodes précédents. L'objectif est d'obtenir une moyenne mobile de 200 points pour considérer que l'environnement est résolu.
9. Nous mettons à jour la valeur d'epsilon (**epsilon**) pour réduire progressivement l'exploration au fil du temps.
10. Si l'agent atteint une moyenne mobile de 200 points, l'environnement est considéré comme résolu, et nous enregistrons le modèle du réseau **q_network** dans un fichier "lunar_lander_model.h5".
11. Le processus d'apprentissage se termine après un certain nombre d'épisodes, ou lorsque l'environnement est résolu, et le temps d'exécution total est affiché.

En résumé, cette section de code met en œuvre l'apprentissage par renforcement profond à l'aide de l'algorithme DQN pour résoudre l'environnement "Force3". L'agent apprend à sélectionner les actions de manière à maximiser sa récompense cumulative au fil du temps, en utilisant un processus d'exploration-exploitation contrôlé par epsilon. Lorsque l'agent atteint un certain niveau de performance, l'environnement est considéré comme résolu, et le modèle appris est sauvegardé.