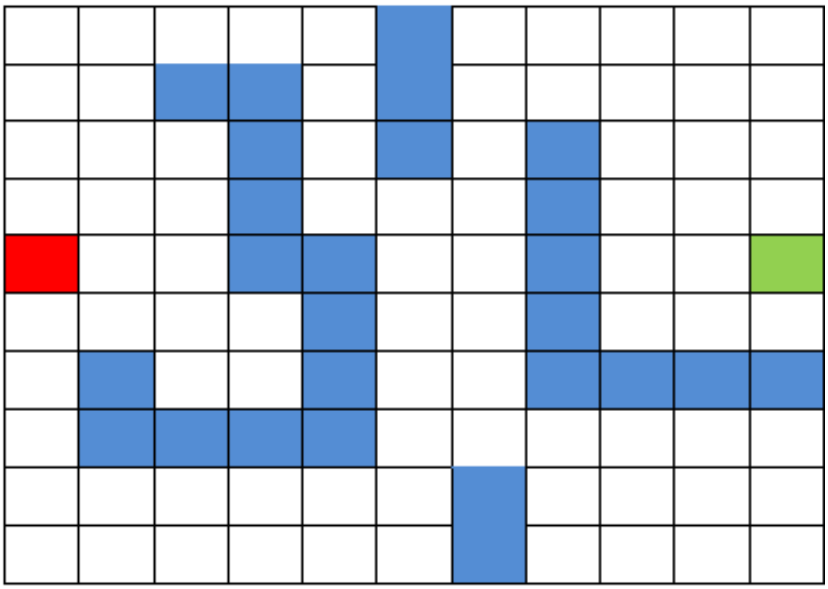
Problème du labyrinthe

Youssef Abdelhedi et Aicha Ettriki

Contents

1 Introduction

Le Q-learning est une technique d'apprentissage par renforcement dans le domaine de l'intelligence artificielle. Il est utilisé pour résoudre des problèmes où un agent doit prendre des décisions séquentielles pour maximiser une récompense cumulée à long terme. Le Q-learning fonctionne en apprenant une fonction d'évaluation de l'action, appelée fonction Q, qui estime la récompense attendue pour prendre une certaine action dans un état donné. L'agent utilise cette fonction pour prendre des décisions en choisissant l'action qui maximise la valeur Q. Le Q-learning est basé sur une méthode itérative où l'agent ajuste progressivement ses estimations de la fonction Q en explorant et en exploitant différentes actions dans son environnement. Dans le cadre de ce rapport, nous explorerons le Q-learning, en le mettant en contexte avec le problème du labyrinthe. Imaginez un labyrinthe avec des passages, des murs et une sortie. Dans ce labyrinthe, un agent virtuel doit trouver son chemin de l'entrée à la sortie en prenant une série de décisions à chaque intersection. Chaque fois que l'agent atteint la sortie, il reçoit une récompense positive, tandis que s'il heurte un mur, il reçoit une récompense négative.



Utilisant le Q-learning, l'agent commence par explorer le labyrinthe de manière aléatoire, en prenant des décisions au hasard à chaque intersection. À mesure qu'il explore, il met à jour ses estimations des valeurs Q, qui représentent la récompense attendue pour chaque action dans chaque état du labyrinthe. En apprenant par essais et erreurs, l'agent découvre progressivement les chemins les plus efficaces pour atteindre la sortie.

2 Pseudo-code du Q-learning

Algorithm 1 Q-Learning

Initialiser Q(s, a) arbitrairement pour toutes les paires état-action
for chaque épisode **do**
    Initialiser l'état s
    while s n'est pas l'état terminal **do**
        Choisir une action a pour s en utilisant une politique dérivée de Q (par exemple, ϵ-greedy)
        Exécuter l'action a, observer la récompense r et l'état suivant s'
        Mettre à jour la valeur Q de l'état-action actuel :
            $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$
        $s \leftarrow s'$
    end while
end for

Voici une explication des étapes clés de cet algorithme :

1. **Initialisation :** Les valeurs $Q(s, a)$ sont initialisées arbitrairement pour toutes les paires état-action.
2. **Boucle principale :** Pour chaque épisode :
   (a) **Initialisation de l'état :** L'agent commence dans un état initial.
   (b) **Boucle de l'épisode :**
      i. **Choix de l'action :** L'agent choisit une action à prendre dans cet état.
      ii. **Exécution de l'action :** L'agent exécute l'action choisie dans l'environnement et observe la récompense résultante ainsi que l'état suivant.
      iii. **Mise à jour de Q :** Les valeurs Q sont mises à jour en fonction de la récompense reçue, de la valeur de Q actuelle pour l'état-action, de la meilleure valeur Q pour l'état suivant, et des paramètres d'apprentissage α (taux d'apprentissage) et γ (facteur d'actualisation).
      iv. **Transition d'état :** L'agent passe à l'état suivant et la boucle continue jusqu'à ce qu'un état terminal soit atteint.
   (c) **Fin de l'épisode :** Une fois l'épisode terminé, l'algorithme revient à l'étape précédente et commence un nouvel épisode.

3 Résolution du Problème du Labyrinthe avec le Q-learning

Problème du Labyrinthe et Q-learning :

Le problème du labyrinthe est un cas, où un agent doit naviguer à travers un labyrinthe pour atteindre une destination tout en évitant les obstacles. L'objectif est d'apprendre une politique de décision optimale qui guide l'agent vers la sortie du labyrinthe tout en minimisant le nombre de pas et en évitant les chemins sans issue.

L'algorithme du Q-learning est une approche pour résoudre ce problème. Dans ce contexte, **les états représentent les positions possibles de l'agent dans le labyrinthe**, **les actions sont les mouvements disponibles** (par exemple, haut, bas, gauche, droite...), et **les récompenses sont définies pour chaque état-action**, avec une récompense positive pour atteindre la sortie et une récompense négative pour heurter un mur.

L'agent utilise l'algorithme du Q-learning pour explorer le labyrinthe, apprendre les valeurs de Q pour chaque paire état-action, et déterminer la meilleure action à prendre dans chaque situation pour maximiser les récompenses cumulatives. Au fil du temps, l'agent converge vers une politique optimale qui lui permet de naviguer efficacement à travers le labyrinthe et d'atteindre la sortie de manière optimale.

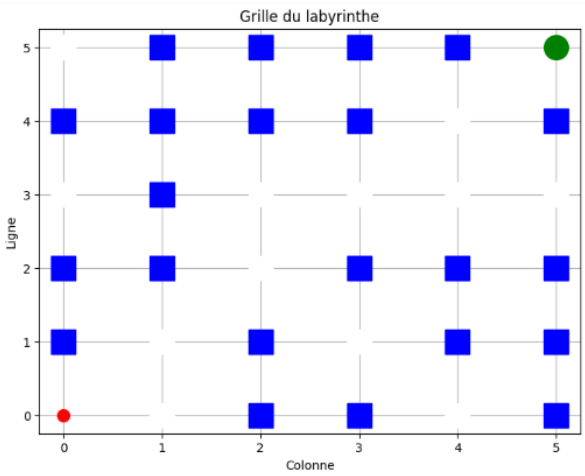
3.1 Mise en place de notre grille

Nous allons tout d'abord, définir notre grid que nous allons utiliser comme support pour le labyrinthe.

```
# 0: case vide
# -10: mur infranchissable
# 100: case de sortie

grid = [
    [-10, 0, -10, -10, 0, -10],
    [-10, 0, -10, 0, -10, -10],
    [-10, -10, 0, -10, -10, -10],
    [0, -10, 0, 0, 0, 0],
    [-10, -10, -10, -10, 0, -10],
    [0, -10, -10, -10, -10, 100]
]
```

Nous aurons comme résulat ce graph :
Les carrés bleus sont des murs infranchissables , le point rouge c'est le point de départ et le point vert c'est la sortie.



Nous créons ensuite ce que l'on appelle une table ou une matrice de type Q qui suit la forme [état, action] et nous initialisons nos valeurs à zéro. Nous mettons aprés à jour et stockons nos q-valeurs après chaque épisode. Ce tableau de valeurs devient une table de référence pour notre agent qui sélectionne la meilleure action en fonction des valeurs de cette matrice.

```
Q = np.zeros((len(grid), len(grid[0]), 8))
```

En paramètre on a len(grid) qui renvoie la longueur de la liste 'grid', qui correspond au nombre de lignes dans la grille , len(grid[0]) qui renvoie la longueur de la première sous-liste de 'grid', qui correspond au nombre de colonnes dans la grille et le troisième paramètre c'est la dimension supplémentaire pour stocker les valeurs Q pour chaque paire (état, action). Dans ce contexte, le robot peut se déplacer de 8 façons différentes, donc il y a 8 actions possibles pour chaque état.

3.2 Choix d'action (epsilon-greedy)

L'agent interagit avec son environnement de deux façons distinctes. Tout d'abord, il explore en agissant de manière aléatoire. C'est ce qu'on appelle l'exploration. Au lieu de sélectionner des actions basées sur la récompense future maximale, il choisit une action au hasard. Cette démarche est cruciale car elle permet à l'agent d'explorer et de découvrir de nouveaux états qui pourraient autrement être négligés lors du processus d'exploitation. Ensuite, l'agent se tourne vers l'exploitation en utilisant la table Q comme référence et en évaluant toutes les actions possibles pour un état donné. Il sélectionne alors l'action qui offre la plus grande valeur. Ce processus d'exploitation utilise les informations disponibles pour prendre des décisions.

Voici un code d'une fonction illustrant ce concept :

```
def choose_action(state,epsilon):
    if random.random() < epsilon:
        return random.randint(0, 7) #exploration
    else:
        return np.argmax(Q[state[0]][state[1]]) #exploitation
```

3.3 Déplacement du robot dans la grille

Nous allons créer la fonction `get_new_state(state, action)` qui représente une fonction de transition d'état pour le robot. Il se déplace dans la grille en fonction de l'action qu'il prend à partir de son état actuel. Cette fonction prend en entrée l'état actuel du robot (ses coordonnées dans la grille) ainsi que l'action qu'il souhaite entreprendre. Elle calcule ensuite les nouvelles coordonnées du robot en fonction de cette action, en vérifiant les limites de la grille et en s'assurant qu'il ne se déplace pas vers un mur (représenté par une valeur de -10 dans la grille). Si le nouvel état mène à un mur, la fonction retourne l'état actuel du robot ; sinon, elle retourne les nouvelles coordonnées calculées. En résumé, cette fonction permet de déplacer le robot dans la grille tout en évitant les obstacles.

```
    def get_new_state(state, action):
    i, j = state  # Les coordonn es actuelles du robot
    rows = len(grid)
    cols = len(grid[0])

    # Calcul des nouvelles coordonn es en fonction de l'action
    if action == 0 and i > 0 and j > 0:  # Haut gauche
        new_i, new_j = i - 1, j - 1
    elif action == 1 and i > 0:  # Haut
        new_i, new_j = i - 1, j
    elif action == 2 and i > 0 and j < cols - 1:  # Haut droite
        new_i, new_j = i - 1, j + 1
    elif action == 3 and j > 0:  # Gauche
        new_i, new_j = i, j - 1
    elif action == 4 and j < cols - 1:  # Droite
        new_i, new_j = i, j + 1
    elif action == 5 and i < rows - 1 and j > 0:  # Bas gauche
        new_i, new_j = i + 1, j - 1
    elif action == 6 and i < rows - 1:  # Bas
        new_i, new_j = i + 1, j
    elif action == 7 and i < rows - 1 and j < cols - 1:  # Bas droite
        new_i, new_j = i + 1, j + 1
    else: # Pas dans la grille
        return state

    # V rification si le nouvel  tat  m ne   un mur
    if grid[new_i][new_j] == -10:
        # Si le nouvel  tat  m ne   un mur, retourner l' tat  actuel
        return state
    else:
        # Sinon, retourner le nouvel  tat
        return (new_i, new_j)
```

3.4 Implémentation de l'algorithme Q-learning

1. La fonction `Q_learning(epsilon, gamma, grid)` prend trois paramètres en entrée : `epsilon`, qui contrôle l'exploration par rapport à l'exploitation, `gamma`, qui représente le facteur de remise pour les récompenses futures, et `grid`, qui est la grille de l'environnement dans lequel l'agent évolue.

2. La boucle `for` principale exécute l'algorithme d'apprentissage par renforcement pendant un certain nombre d'itérations (dans cet exemple, 1000 itérations).

3. À chaque itération, l'état de départ du robot est initialisé à `(0, 0)`, représentant la position de départ dans la grille.

4. La boucle `while` est utilisée pour continuer à exécuter l'algorithme jusqu'à ce que l'agent atteigne la sortie de la grille, représentée par les coordonnées `(len(grid)-1, len(grid[0])-1)`.

5. À chaque itération de la boucle `while`, l'agent choisit une action pour l'état actuel en utilisant notre fonction `choose_action(state, epsilon)`.

6. Ensuite, l'agent obtient le nouvel état après avoir pris cette action en appelant la fonction `get_new_state(state, action)`. Cela met à jour la position de l'agent dans la grille en fonction de l'action choisie.

7. L'agent reçoit une récompense pour avoir atteint le nouvel état, récupérée à partir de la grille à l'aide des coordonnées du nouvel état.

8. Ensuite, l'agent calcule la meilleure récompense future pour le nouvel état en prenant la valeur maximale de la table Q associée à cet état.

9. Enfin, l'agent met à jour la valeur Q pour l'état et l'action actuels en utilisant la formule de mise à jour de la valeur Q du Q-learning.

10. L'agent met à jour l'état actuel pour le prochain pas et répète le processus jusqu'à ce qu'il atteigne la sortie de la grille ou que le nombre d'itérations défini soit atteint.

```
def Q_learning(epsilon,gamma,grid):
    for n in range(1000):
        state = (0, 0)
        while state != (len(grid)-1, len(grid[0])-1):
            action = choose_action(state,epsilon)
            new_state = get_new_state(state, action)
            reward = grid[new_state[0]][new_state[1]]
            best_future_reward = np.max(Q[new_state[0]][new_state[1]])
            Q[state[0]][state[1]][action] = reward + gamma *best_future_reward
            state = new_state
```

4 Illustration graphique

4.1 Le chemin le plus optimale

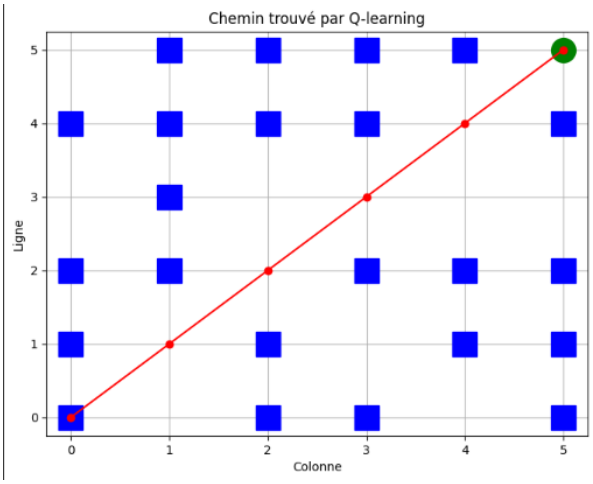
La fonction `Q_learning(0.1,0.9)` est appelée ensuite, la fonction `plot_path(grid, Q)` est appelée pour tracer le chemin optimal trouvé par l'algorithme Q-learning dans la grille. À l'intérieur de la fonction `plot_path`, on définit un état initial $(0, 0)$ et un chemin est initialisé avec cet état. En utilisant la politique apprise représentée par la table Q, l'algorithme sélectionne itérativement l'action avec la valeur Q maximale pour chaque état. Cela permet de déterminer le prochain état à visiter dans le chemin.Le chemin complet est stocké dans la liste `path`. Ensuite, le chemin est tracé sur la grille à l'aide de la bibliothèque Matplotlib. Le chemin optimal est tracé en reliant les états successifs du chemin avec des ronds rouges.

```
Q_learning(0.1,0.9)

def plot_path(grid, Q):
    current_state = (0, 0)
    path = [current_state]
    while current_state != (len(grid)-1, len(grid[0])-1):
        action = np.argmax(Q[current_state[0], current_state[1]])
        new_state = get_new_state(current_state, action)
        path.append(new_state)
        current_state = new_state

    # Tracer le chemin
    path_x = [step[1] for step in path]
    path_y = [step[0] for step in path]
    plt.figure(figsize=(8, 6))
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            if grid[i][j] == -1:  # Case blanche
                plt.plot(j, i, 'ws', markersize=20)
            elif grid[i][j] == -10:  # Mur
                plt.plot(j, i, 'bs', markersize=20)
            elif grid[i][j] == 100:  # Sortie
                plt.plot(j, i, 'go', markersize=20)
    plt.plot(path_x, path_y, marker='o', color='red')
    plt.title("Chemin trouv  par Q-learning")
    plt.xlabel('Colonne')
    plt.ylabel('Ligne')
    plt.grid(True)
    plt.show()
    plot_path(grid, Q)
```

Voici la sortie de ce code :



5 Variation des Paramètres ϵ et γ

5.1 Impact sur le Nombre de Cycles

Nous apportons une modification à notre fonction Q_learning afin de calculer le nombre de cycles. À chaque itération du cycle, nous enregistrons les valeurs d'ϵ, γ et le nombre de cycles pour chaque combinaison de ϵ et γ.

```
    def q_learning_cycle(epsilon, gamma):
        for n in range(1000):
            state = (0, 0)
            cycles = 0
            while state != (len(grid)-1, len(grid[0])-1):
                action = choose_action(state, epsilon)
                new_state = get_new_state(state, action)
                reward = grid[new_state[0]][new_state[1]]
                best_future_reward = np.max(Q[new_state[0]][new_state[1]])
                Q[state[0]][state[1]][action] = reward + gamma *
                    best_future_reward
                state = new_state
                cycles += 1

        return {'epsilon': epsilon, 'gamma': gamma, 'cycles': cycles}
```

Ce code explore différentes combinaisons de valeurs pour les paramètres ϵ et γ , en enregistrant le nombre de cycles nécessaires.

```
epsilon_values = [0.01,0.1, 0.5, 0.9]
gamma_values = [0.01,0.1, 0.5, 0.9]
results = []
for epsilon in epsilon_values:
    for gamma in gamma_values:
        cycles = q_learning_cycle(epsilon, gamma)
        results.append(cycles)


print(results)
```

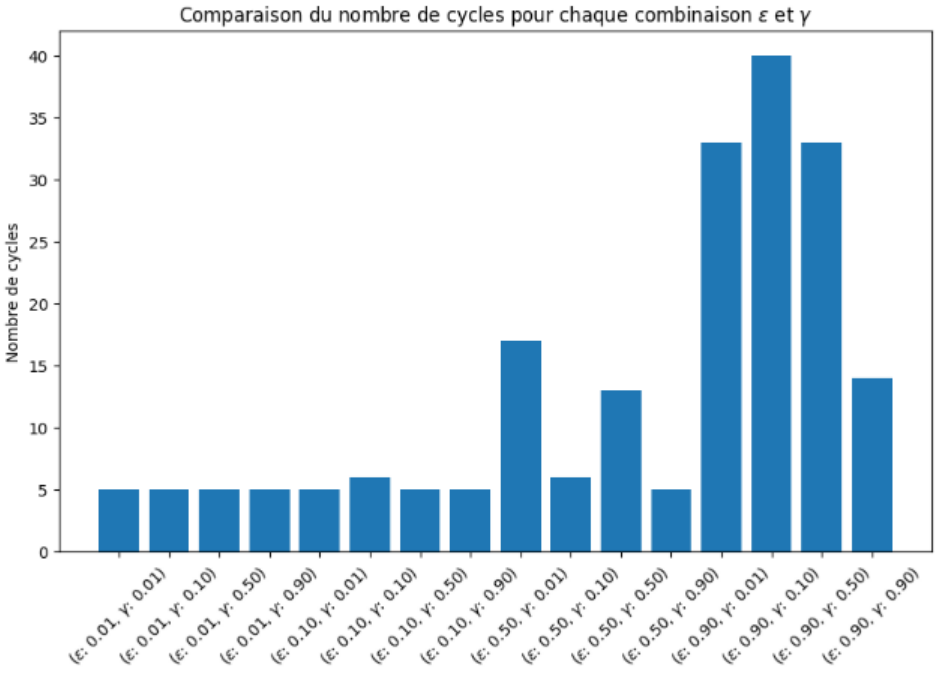
Le code ci-dessus nous donne cette sortie :

```
[[{'epsilon': 0.01, 'gamma': 0.01, 'cycles': 5},
{'epsilon': 0.01, 'gamma': 0.1, 'cycles': 5},
{'epsilon': 0.01, 'gamma': 0.5, 'cycles': 5},
```

```
{'epsilon': 0.01, 'gamma': 0.9, 'cycles': 5},
{'epsilon': 0.1, 'gamma': 0.01, 'cycles': 5},
{'epsilon': 0.1, 'gamma': 0.1, 'cycles': 5},
{'epsilon': 0.1, 'gamma': 0.5, 'cycles': 5},
{'epsilon': 0.1, 'gamma': 0.9, 'cycles': 5},
{'epsilon': 0.5, 'gamma': 0.01, 'cycles': 12},
{'epsilon': 0.5, 'gamma': 0.1, 'cycles': 9},
{'epsilon': 0.5, 'gamma': 0.5, 'cycles': 7},
{'epsilon': 0.5, 'gamma': 0.9, 'cycles': 7},
{'epsilon': 0.9, 'gamma': 0.01, 'cycles': 61},
{'epsilon': 0.9, 'gamma': 0.1, 'cycles': 59},
{'epsilon': 0.9, 'gamma': 0.5, 'cycles': 80},
{'epsilon': 0.9, 'gamma': 0.9, 'cycles': 20}]]
```

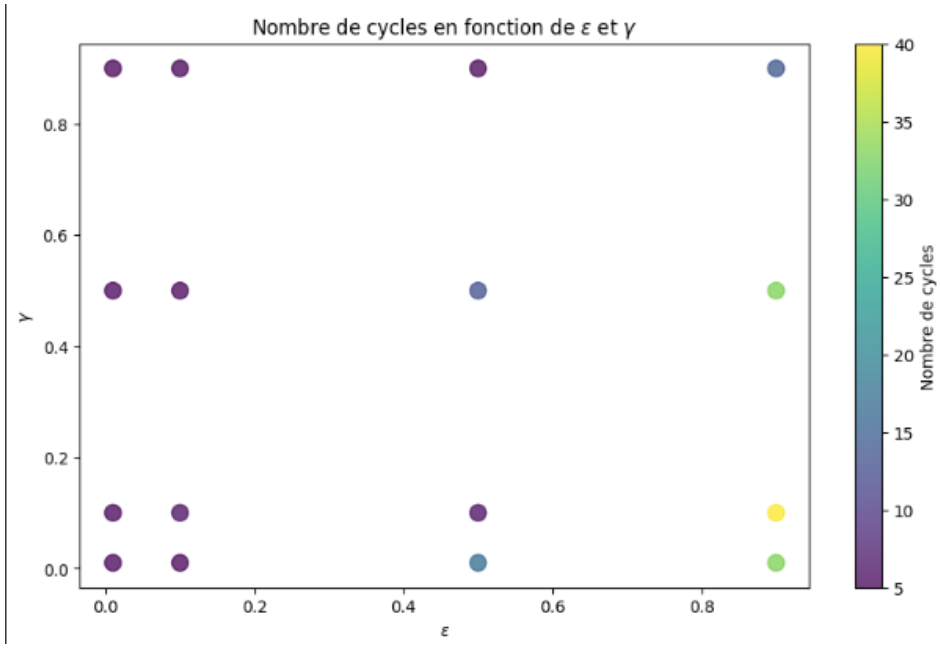
Effet de ϵ :
Pour de petites valeurs de ϵ (par exemple, $\epsilon = 0.01$), l'agent privilégie fortement l'exploitation plutôt que l'exploration. Cela signifie qu'il choisit principalement les actions basées sur ce qu'il connaît déjà, ce qui peut conduire à une convergence rapide vers une solution, comme indiqué par le faible nombre de cycles (par exemple, 5 cycles pour $\epsilon = 0.01$). En revanche, pour des valeurs plus élevées de ϵ (par exemple, $\epsilon = 0.9$), l'agent privilégie davantage l'exploration, ce qui l'amène à explorer plus activement de nouvelles actions. Cela peut conduire à une convergence plus lente vers une solution optimale, comme indiqué par un nombre plus élevé de cycles nécessaires (par exemple, 61 cycles pour $\epsilon = 0.9$).
Effet de γ :
Pour de petites valeurs de γ (par exemple, $\gamma = 0.01$), l'agent accorde peu d'importance aux récompenses futures, ce qui peut entraîner une convergence rapide vers une solution locale mais potentiellement sous-optimale. Cependant, cela peut nécessiter moins de cycles pour converger car l'agent est moins attiré à explorer de nouvelles options.

5.2 Quelques illustrations



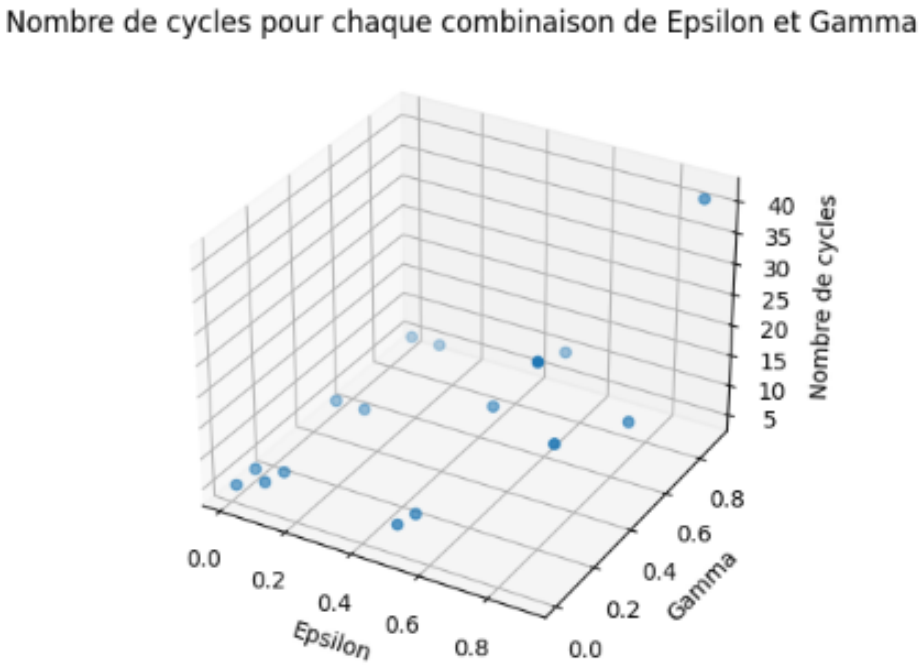
Graphique à barres

Ce graphique compare le nombre de cycles nécessaires pour différentes combinaisons d'ϵ et de γ utilisées. On peut remarquer que lorsque γ est augmenté, le nombre de cycles augmente également.



Nuage de points

En général, en se déplaçant de gauche à droite sur l'axe des x (ϵ plus élevé), le nombre de cycles augmente pour toutes les valeurs de γ. En effet, avec plus d'exploration (ϵ plus élevé), l'agent met plus de temps à converger vers une politique optimale.Pour chaque valeur de ϵ, à mesure que γ augmente , le nombre de cycles tend à diminuer.



Visualisation en 3D

Voici ci-dessus un graph en 3D, l'objectif idéal est de trouver un équilibre entre l'exploration et l'exploitation. Le graphique suggère que, pour ce cas spécifique, une combinaison d'epsilon modéré et de gamma plus élevé pourrait conduire au plus faible nombre de cycles nécessaires à l'apprentissage de l'agent