Atelier 4
«QLearning »

Réalisée par:
EL KTIBI El hassane
CHAOUKI Mouad

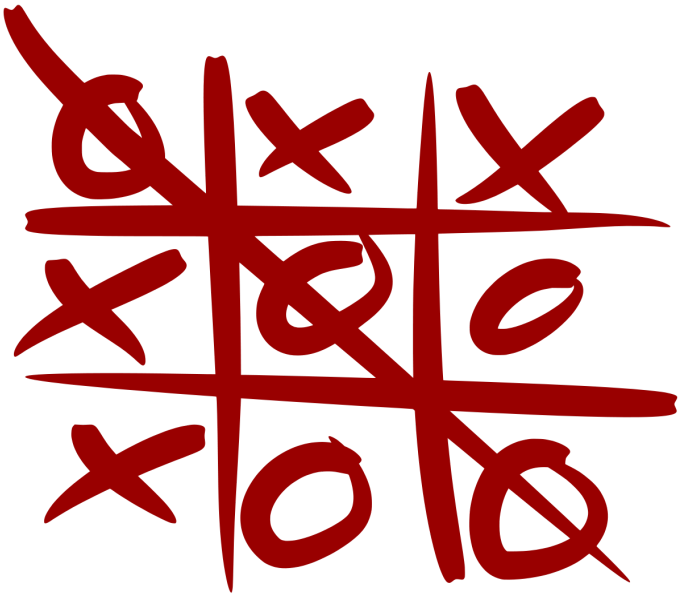
Encadrée par:
Pr . EL AACHAK LOTFI

Cycle Ingénieur
Logiciel et Systèmes Intelligents LSI2_S3
Module : Méthodologies d'intelligence Artificielle
Année Universitaire 2019/2020

Département Génie Informatique

Table des matières

Objective



L'objectif principal de cet atelier est d'implémenter l'algorithme Q-learning dans un agent qui va contrôler et enseigner le joueur 2 les stratégies du jeu Tic Tac Toe.

Règles

Deux joueurs s'affrontent. Ils doivent remplir chacun à leur tour une case de la grille avec le symbole qui leur est attribué : O ou X. Le gagnant est celui qui arrive à aligner trois symboles identiques, horizontalement, verticalement ou en diagonale. Il est coutume de laisser le joueur jouant X effectuer le premier coup de la partie.

Outils

IDE : VS code

Language : Python

Libraries : numpy, matplotlib, argparse os, pickle, sys, random.

Partie 1 (jeu Tic tac Toe):

Le fichier game.py contient la Class **Game** qui doit contenir tous les méthodes nécessaires pour jouer le jeu Tic Tac Toe.

Test des fonctionnalités :

```
if __name__ == "__main__":
    alpha=0.4
    gamma=0.9
    epsilon=0.2
    agen= Qlearning(alpha, gamma, epsilon)
    gm= Game(agen)
    gm.start()
```

Test d'une partie:

```
Your move! Please select a row and column from 0-2 in the format row,col: 0 1

    0   1   2
0   -   X   -
1   -   -   O
2   -   -   -
Your move! Please select a row and column from 0-2 in the format row,col: 1 1

    0   1   2
0   O   X   -
1   -   X   O
2   -   -   -
Your move! Please select a row and column from 0-2 in the format row,col: 2 1

    0   1   2
0   O   X   -
1   -   X   O
2   -   X   -
Player wins!
```

Partie 2 (Agent Q learning):

Développement de la classe Agent qui contient tous les méthodes nécessaires pour assurer les fonctionnalités de l'algorithme Qlearning.

Initialisation des paramètres, actions possibles, Q table, et la liste des récompenses.

```
def __init__(self, alpha, gamma, eps, eps_decay=0.):
    # Agent parameters
    self.alpha = alpha
    self.gamma = gamma
    self.eps = eps
    self.eps_decay = eps_decay
    # Possible actions correspond to the set of all x,y coordinate pairs
    self.actions = []
    for i in range(3):
        for j in range(3):
            self.actions.append((i,j))
    # Initialize Q values to 0 for all state-action pairs.
    # Access value for action a, state s via Q[a][s]
    self.Q = {}
    for action in self.actions:
        self.Q[action] = collections.defaultdict(int)
    # Keep a list of reward received at each episode
    self.rewards = []
```

Méthode **get_actions()** pour sélectionner une action en fonction de l'état actuel du jeu.

```
def get_action(self, s):
    # Only consider the allowed actions (empty board spaces)
    possible_actions = [a for a in self.actions if s[a[0]*3 + a[1]] == '-']
    if random.random() < self.eps:
        # Random choose.
        action = possible_actions[random.randint(0,len(possible_actions)-1)]
    else:
        # Greedy choose.
        values = np.array([self.Q[a][s] for a in possible_actions])
        # Find location of max
        ix_max = np.where(values == np.max(values))[0]
        if len(ix_max) > 1:
            # If multiple actions were max, then sample from them
            ix_select = np.random.choice(ix_max, 1)[0]
        else:
            # If unique max action, select that one
            ix_select = ix_max[0]
        action = possible_actions[ix_select]

    # update epsilon; geometric decay
    self.eps *= (1.-self.eps_decay)
    return action
```

Méthode **update()** pour effectuer la mise à jour Q-Learning des valeurs Q.
Paramètres:

s: état précédent

s_: nouvel état

a: action précédente

a_: nouvelle action.

r: récompense reçue après avoir exécuté l'action "a" dans l'état "s"

```
def update(self, s, s_, a, a_, r):
        # Update Q(s,a)
        if s_ is not None:
            # hold list of Q values for all a_,s_ pairs. We will access the max later
            possible_actions = [action for action in self.actions if s_[action[0]*3 + action[1]] == '-']
            Q_options = [self.Q[action][s_] for action in possible_actions]
            # update
            self.Q[a][s] += self.alpha*(r + self.gamma*max(Q_options) - self.Q[a][s])
        else:
            # terminal state update
            self.Q[a][s] += self.alpha*(r - self.Q[a][s])
        # add r to rewards list
        self.rewards.append(r)
```

Développement la classe GameLearning du fichier play.py.

Initialisation des paramètres, agent.

```
class GameLearning:
    """
    A class that holds the state of the learning process. Learning
    agents are created/loaded here, and a count is kept of the
    games that have been played.
    """
    def __init__(self, alpha=0.5, gamma=0.9, epsilon=0.1):
        self.games_played = 0
        self.agent = Qlearning(alpha,gamma,epsilon)
```

Méthode **beginPlaying()** pour jouer avec un humain et apprendre à travers plusieurs jeux.

```
def beginPlaying(self):
    """ Loop through game iterations with a human player. """
    print("Welcome to Tic-Tac-Toe. You are 'X' and the computer is 'O'.")

    def play_again():
        print("Games played: %i" % self.games_played)
        while True:
            play = input("Do you want to play again? [y/n]: ")
            if play == 'y' or play == 'yes':
                return True
            elif play == 'n' or play == 'no':
                return False
            else:
                print("Invalid input. Please choose 'y' or 'n'.")

    while True:
        game = Game(self.agent)
        game.start()
        self.games_played += 1
        if not play_again():
            print("OK. Quitting.")
            break
```

Méthode **beginTeaching()** pour démarrer le processus d'apprentissage automatique a travers un agent qui sait les stratégies optimales (fichier teach.py).

```
def beginTeaching(self, episodes):
    """ Loop through game iterations with a teaching agent. """
    teacher = Teacher()
    # Train for alotted number of episodes
    while self.games_played < episodes:
        game = Game(self.agent, teacher=teacher)
        game.start()
        self.games_played += 1
        # Monitor progress
        if self.games_played % 1000 == 0:
            print("Games played: %i" % self.games_played)
```

Méthode **plot_agent_reward()** pour visualiser l'évolution des récompenses.

```
def plot_agent_reward(rewards):
    """ Function to plot agent's accumulated reward vs. iteration """
    plt.plot(np.cumsum(rewards))
    plt.title('Agent Cumulative Reward vs. Iteration')
    plt.ylabel('Reward')
    plt.xlabel('Episode')
    plt.show()
```

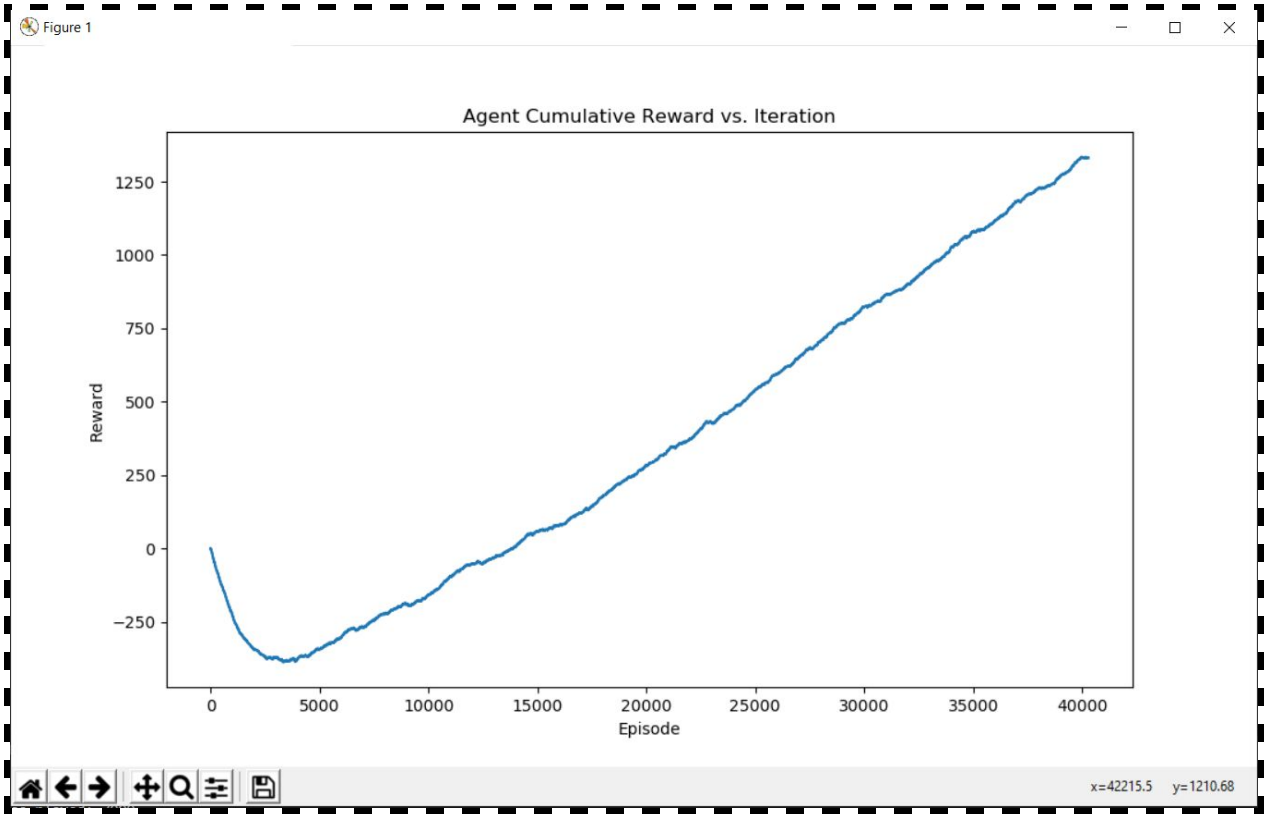
Deux modes d'apprentissage: manuel ou automatique via l'agent enseignant **teacher** .

```
if __name__ == "__main__":
    # instantiate a new GameLearning instance
    gl = GameLearning()

    # Automatic Teaching mode
    gl.beginTeaching(10000)

    # Manual Teaching mode
    # gl.beginPlaying()
```

Visualisation des récompenses après 10000 jeux avec l'agent **teacher**.



Au début, l'agent commence à découvrait l'environnement, donc les récompenses étaient très faibles et négatives, et après environ 3000 itérations, l'agent commence à comprendre l'environnement alors les récompenses commencent à croître.