ÉCOLE NATIONALE SUPÉRIEURE DE TECHNIQUES AVANCÉES

ROB311 : Apprentissage pour la robotique

Parcours robotique

Année universitaire : 2020/2021

TP3: Q-Learning et Pac-man

Auteurs :

M. AHMED YASSINE HAMMAMI

MME. HANIN HAMDI

Objectifs

L'objectif de ce TP est d'implémenter l'algorithme de Q-Learning en Python. En particulier, on va former un joueur IA du célèbre jeu d'arcade Pac-Man.

On va travailler sur un ensemble de fichiers et de bibliothèques Python afin de modifier certaines fonctions pour implémenter l'IA Q-Learning et Approximate Q-Learning pour le jeu Pac-Man.

L'algorithme Q-Learning

1. Initialiser Q(s,a) arbitrairement, pour tout état s et pour toute action a.
2. Répéter pour chaque épisode :
   (a) Trouver pour chaque action et pour chaque état de la matrice π et calculer la fonction de valeur à partir de l'équation de Bellman :
   $$V^*(S_t) = R(S_t) + max_a \gamma \sum T(S_t, a, S_{t+1}) V^*(S_{t+1})$$
   (b) Pour chaque état, choisir l'action optimale et mettre à jour π selon :
   $$\pi(S_t) = arg\ max_a \sum T(S_t, a, S_{t+1}) V^*(S_{t+1})$$
   (c) Répéter ces deux étapes pour chaque état jusqu'à convergence. Dans notre cas, on considère que l'algorithme converge si la matrice π ne change pas.

Implémentation des fonctions de la classe *QLearningAgent(ReinforcementAgent)* :

init() :

Dans cette fonction, on va initialiser les Qvalues :

```
def __init__(self, **args):
    ReinforcementAgent.__init__(self, **args)
    self.QValues = util.Counter()
```

getvalues() :

Cette fonction retourne la Q-valeur[state,action]

```
def getQValue(self, state, action):
    return self.QValues[state, action]
```

computeValueFromQValues() :

Cette fonction retourne l'action maximale parmi tous les actions possibles, s'il n'y a pas d'action possible, la fonction retourne 0.0.

```
def computeValueFromQValues(self, state):
    all_values = []
    for action in self.getLegalActions(state) :
```

```
        vv=self.getQValue(state, action)
        all_values.append(vv)

    if (all_values):
        return max(all_values)
    else:
        return 0.0
```

computeActionFromQValues() :

Cette fonction retourne la meilleure action qu'on peut avoir dans un état. S'il n y a pas d'action possible, elle retourne None.

```
def computeActionFromQValues(self, state):

    #we start first of all to determine the legal actions
    l_actions = self.getLegalActions(state)

    value = self.getValue(state)
    for action in l_actions:
        if (value == self.getQValue(state, action)):
            return action
    return None
```

getAction() :

Elle calcule l'action dans l'état actuel avec la probabilité epsilon. On choisit une action aléatoire dans un état, sinon on choisit la "Policy" optimale actuelle.

```
def getAction(self, state):

    legalActions = self.getLegalActions(state)
    action = None

    if (util.flipCoin(self.epsilon)):
        action = random.choice(legalActions)
    else:
        action = self.getPolicy(state)

    return action
```

update() :

```
def update(self, state, action, nextState, reward):

    #calculate the new Qvalue
    new = (1 - self.alpha) * self.getQValue(state, action)
    new += self.alpha * (reward + (self.discount * self.getValue(nextState)))
```

```
        self.QValues[state, action] = new
```

Implémentation des fonctions de la classe *ApproximateQAgent(PacmanQAgent)* :

Dans cette partie, on va implémenté un agent Q-learning approximatif. On commence tout d'abord par former l'agent puis le tester.

getQValue() :

```
def getQValue(self, state, action):

        features = self.featExtractor.getFeatures(state,action)
        Qvalue = 0.0

        for f in features:
            QValue += self.weights[f] * features[f]

        return QValue
```

L'idée ici est que QValue est représentée comme une somme linéaire pondérée des valeurs de caractéristiques pour un état, nous avons maintenant besoin d'un moyen pour obtenir la valeur de caractéristique de l'état afin de mettre à jour les poids.

Après avoir effectuer une action à partir d'un état, on passe à un nouvel état et on obtient un "reward", basée sur *features*, *reward* et *l'état suivant.*

update() :

```
    def update(self, state, action, nextState, reward):

        QValue = 0
        difference = reward + (self.discount * self.getValue(nextState) - self.getQValue(
    ↪ state, action))
        features = self.featExtractor.getFeatures(state, action)

        for feature in features:
          self.weights[feature] += self.alpha * features[feature] * difference
```

Test et résultat

Pour tester l'ensemble des fonctions implémentées, on va lancer cette commande *python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid*.

Cette commande va apprendre à partir de 2000 épisodes d'entraînement, on testera ensuite l'IA résultante sur 10 matchs.

```
Training Done (turning off epsilon and alpha)
-----------------------------------------
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Average Score: 501.4
Scores:        503.0, 503.0, 499.0, 503.0, 495.0, 503.0, 503.0, 499.0, 503.0, 503.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

FIGURE 1 – Résultat du test

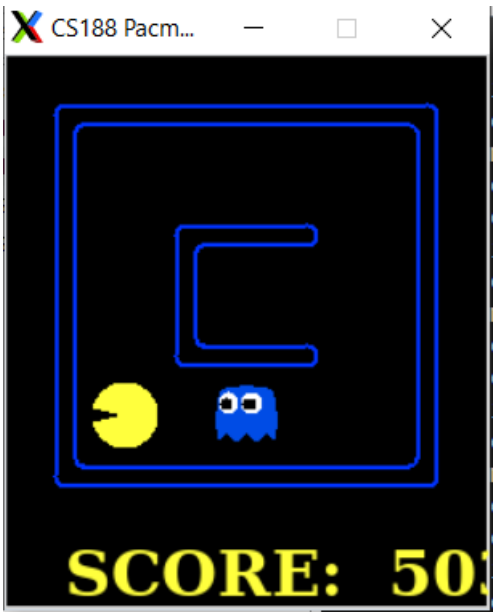

FIGURE 2 –

On remarque que le win Rate est égale à 10/10 ce qui affirme que notre algorithme est en train d'apprendre correctement dans la phase training.