ROB311 - TD3

BRAMBILLA Davide Luigi - GOMES DA SILVA Rafael

September 30, 2019

In order to execute the program the command to run is:

python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l ["layout_name"].

python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l ["layout_name"].

The code is available on the git lab repository:
https://gitlab.data-ensta.fr/gomesdasilva/rob311.git

1 Introduction

Reinforcement learning

Reinforcement learning is a technique used in machine learning to train algorithms using a reward and punishment system without programmer interference and is inspired by the behaviorist psychology. The purpose of this technique is to train agents so they can interact in a given environment, moving from one state to another through actions until they reach their goal, as shown in Figure 1. Each action taken by the agent is rewarded (or punished) until he learns which actions to take to increase the reward and achieve the goal.

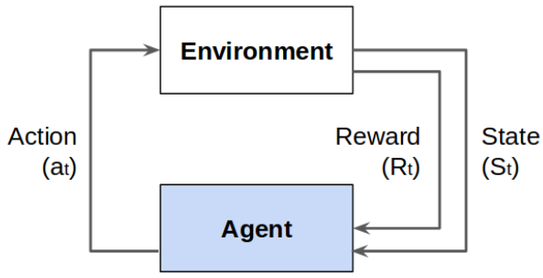

Figure 1: Reinforcement learning scheme

Q-Learning

Q-Learning is an off policy RL algorithm that uses a table called Q-Table that maps the possible values of an action in a given state to enable the agent to just look up the table to choose an action.

The Q-table values are calculated by performing a stage of training in which the agent chooses random actions and, based on the rewards received, updates the table following the equation (1).

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)] \tag{1}$$

In the equation 1 we have the Q-function ($Q(s,a)$), the state (s), the action (a), the learning rate (α), the reward (r), the discounted reward (γ).

In the training phase the agent performs a mapping of the possibility of state change, always taking into account the rewards considering future states. A negative point in using the Q-learning method is that for problems with a wide range of states the computational cost to generate the Q-table is high.

2 Objectif

This TP aims to train an artificial intelligence for the pac-man game. For this, it was necessary to complete the classes *QLearningAgent* and *ApproximateQAgent* in order to implement, respectively, the traditional Q-learning method and a method that approximates Q-learning that learns the weights for states that share the same characteristics.

3 The Class *QLearningAgent*

Traditional Q-learning method provides a table with all possible q-values. This method has the negative point for cases where the number of states to visit is very wide, and this ends up compromising the speed of calculation of the algorithm. One of the solutions to solve this problem will be presented in the next topic, with approximate Q-learning.

3.1 Method *__init__(self, **args)*

This method was used to initialize *qValues* as a structure of type *util.Counter()*. The structure *util.Counter()* is provided in the original code and presents useful methods for performing the search routines for the Q-learning method.

```
"*** YOUR CODE HERE ***"
self.qValues = util.Counter()
```

3.2 Method *getQValue(self, state, action)*

This method has to look inside the *Q-table* and find out the value of Q (*qValue*) associated to the state and to the action given as arguments to the function. It should also returns the value of 0.0 in the case that one between state or action are not inside the table.

```
"*** YOUR CODE HERE ***"
if (state,action) in self.qValues:
    return self.qValues[(state,action)]
else:
    return 0.0
```

3.3 Method *computeValueFromQValues(self, state)*

This method returns the maximum value of $Q(s, a)$, within all the available actions for a given state. The function *getLegalActions* present in the file *learningAgents.py* is responsible to verify what are the actions available for a given state. In case there's no legal actions, the method should return 0.

```
"*** YOUR CODE HERE ***"
legalActions = self.getLegalActions(state)
if len(legalActions) == 0:
    return 0.0
max_action = None
max_qValue = None
for action in legalActions:
    if self.getQValue(state, action) > max_qValue:
        max_action = action
        max_qValue = self.getQValue(state, action)
return max_qValue
```

3.4 Method *computeActionFromQValues(self, state)*

This method returns the action associated to the maximum value of $Q(s, a)$, within all the available actions for a given state. The method follow the same logic as the previous one. As there, in case there's no legal actions, the method should return *None*.

```
"*** YOUR CODE HERE ***"
legalActions = self.getLegalActions(state)
if len(legalActions) == 0:
    return None
max_action = None
max_qValue = None
for action in legalActions:
    if self.getQValue(state, action) > max_qValue:
        max_action = action
        max_qValue = self.getQValue(state, action)
return max_action
```

3.5 Method *getAction(self, state)*

This method has to compute and choose the action to take in the current state. With a probability given by the parameter ϵ the action to be taken must be a random action otherwise the action to be taken must be the one related to the optimal policy. In order to calculate the probability we have used the function *util.flipCoin(self.epsilon)* and in order to take a random action between the legal ones we have used the function *randomChoice(legalActions)*. If there are no possible legal actions it has to return the *None* value.

```
"*** YOUR CODE HERE ***"
if len(legalActions) == 0:
    return action
if util.flipCoin(self.epsilon):
    action = random.choice(legalActions)
else:
    action = self.computeActionFromQValues(state)
return action
```

3.6 Method *update(self, state)*

This method is responsible to update the *qValues* in the *Q-table* using (1) to evaluate the state, action, next state and reward related to the current state thta is the one given as input to this function.

```
"*** YOUR CODE HERE ***"
new_qValue = reward + self.discount * self.computeValueFromQValues(nextState)
self.qValues[(state, action)] = (1 - self.alpha) * self.getQValue(state, action) + self.alpha *
    new_qValue
```

4 Class *ApproximateQAgent*

The approximate Q-learning model reduces the number tables present on the traditional Q-learning method, by reducing the number of states due the changing on the computation of $Q(s, a)$ as shown in (2). This reduction is made based on the generalization of the agent's experience, considering that there are situations that have features in common and thus, he has to learn weights for state features where states share characteristics. A negative point of this solution is the fact that states that have common characteristics can, however, produce different results.

$$Q(s, a) = \sum_{i=1}^{n} f_i(s, a) w_i \tag{2}$$

For the case of the approximate Q-learning applied to the pac-man game, the features considered to simplify the Q-learning are:

- **eats-food**: to verify if the Pac-man is not in danger of ghosts, so he can eat the food by taking the given action in a given state
- **closest-food**: the distance between Pac-man and the nearest food, disregarding the walls on the way
- **#-of-ghosts-1-step-away**: Number of ghosts that are one step away from Pac-man (either on safe mode or dangerous mode)

4.1 Method *getQValue(self, state, action)*

This method gets the value of Q (state,action) based on the features and considering its weights.

```
"*** YOUR CODE HERE ***"
return self.getWeights() * self.featExtractor.getFeatures(state, action)
```

4.2 Method *update(self, state, action, nextState, reward)*

This method is responsible to update the *qValues* in the *Q-table* using the approximate version of the Q-learning using (3), and also to update the value of the weights w_i using (4).

$$difference = (r + \gamma \max_{a'} Q(s', a')) - Q(s, a) \tag{3}$$

$$w_i \leftarrow w_i + \alpha . difference . f_i(s, a) \tag{4}$$

```
"*** YOUR CODE HERE ***"
difference = reward + self.discount * self.computeValueFromQValues(nextState) -
    self.getQValue(state, action)
features = self.featExtractor.getFeatures(state, action)
for feat in features:
    self.weights[feat] += self.alpha * difference * features[feat]
```

5 Results

In order to verify the quality of the algorithm developed, both the Q-learning method and the approximate Q-learning method were tested with all the game layout available, and the value of the average training time, the average score and the win rate were recorded in the Table 1 and Table 2, respectively. For the Q-learning method, the game layout went under 2000 episodes of training before being tested, while for the approximate Q-learning method the game layouts went under only 50 episodes of training.

Table 1: Results obtained for the Q-learning method tests for different game layouts

Layout	**Averagetrainningtime**	**Averagescore**	**WinRate**
capsuleClassic	3.84	-464.6	0/10
contestClassic	6.29	-464.4	0/10
mediumClassic	11.79	-466.3	0/10
mediumGrid	2.26	-510.5	0/10
minimaxClassic	0.76	211.9	07/10
openClassic	x	x	x
originalClassic	x	x	x
smallClassic	11.54	-450.6	0/10
smallGrid	2.20	500.2	10/10
testClassic	2.93	-475.3	0/10
trappedClassic	0.78	14	05/10
trickyClassic	23.03	-520.2	0/10

In the Table 1 above, it wasn't possible execute tests on the *OpenClassic* and *OriginalClassic* layouts using the first Q-learning method, due to an error *memory error ("Processus arrêté")* returned by the computer. The values are not available for those cases, and are shown in the table with the value of "x".

Table 2: Results obtained for the approximate Q-learning method tests for different game layouts

Layout	**Average trainning time**	**Average score**	**Win rate**
capsuleClassic	3.83	253.8	06/10
contestClassic	10.65	1105.2	10/10
mediumClassic	10.77	1203.5	09/10
mediumGrid	0.70	527.2	10/10
minimaxClassic	0.40	110.3	06/10
openClassic	8.29	1254.4	10/10
originalClassic	81.75	2346.0	09/10
smallClassic	4.75	746.6	08/10
smallGrid	0.73	200.5	07/10
testClassic	0.76	563.0	10/10
trappedClassic	0.25	117.4	06/10
trickyClassic	22.03	1115.3	06/10

Looking at the results obtained for the two implemented methods, it is possible to verify that for most cases of Q-learning, the win rate is zero and this is due to the fact that for each different scenario it is necessary to change the amount of training to be performed according to its complexity and size, as this method creates a complete table and this implies a need to increase the episodes of training so that all possible situations can be addressed by the analysis done during training.

On the other hand, the approximate Q-learning method proved to be a more effective and have a lower computational cost. By considering that there were only 50 training episodes and seeing that no win rate was zero, we can conclude that the approximate method provides an approach similar to the one a human usually uses when making decisions, for this method makes the agent learn in a more intelligent way and provides the means for him to interpret the environment and make the right decisions to win the game based on the experience of similar scenarios.