UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

REINFORCEMENT LEARNING

Inteligenta Artificiala

Autor: KIRALY NICOLE ELENA
Grupa: 30237

Facultatea de Automatica
si Calculatoare

8 Ianuarie 2024

Cuprins

1 Introducere

Pentru proiectul 3 Pacman-Reinforcement Learning am avut de implementat Q1, Q3 si Q5, care presupun dezvoltarea partii de Value Iteration, Policies si Q-Learning. In prima parte este prezentata partea de MDP(Markov Decision Process), care presupune ca agentul se muta in pozitia specifica doar in **80% din cazuri**, si in 20% va avea ca rezultat o alta pozitie. Se mai adauga si conceptul de discount rate de 0.9 prin care rezulta ca rezultatul final va fi mai mic decat rezultatul asteptat. Toate tranzitiile vor avea un reward de 0, stabilit prin living reward.

2 Question1-Value Iteration

In acest Question, am avut de implementat 2 functii, computeActionFromValues(state), care calculeaza cea mai buna actiune pe care o poate lua agentul, luand in considerare value function si functia computeQValueFromValues(state, action), care returneaza Q-value a perechii (state,action) pe baza functiei valueIteration.

Pentru inceput trebuie implementata functia computeQValueFromValues(state, action), care calculeaza Q value pentru fiecare tranzitie, in functie de probabilitatea sa se intample un eveniment si starea urmatoare, dupa formula prezentata in cod, insumand toate Q-urile de la fiecare posibila tranzitie.

```
1  def computeQValueFromValues(self, state, action):
2          """
3            Compute the Q-value of action in state from the
4            value function stored in self.values.
5          """
6          "*** YOUR CODE HERE ***"
7          #util.raiseNotDefined()
8
9          Q = 0
10         #trans = (nextState,prob)
11         for trans in self.mdp.getTransitionStatesAndProbs(state,action):
12             reward = self.mdp.getReward(state,action,trans[0])
13             Q += trans[1] * (reward + self.discount * self.values[trans[0]])
14         return Q
```

A doua functie implementata este computeActionFromValues(state), care presupune returnarea celei mai bune actiuni dintre cele legale, iar daca e stare terminala None. Stocam valorile intr-un dictionar cu toate valorile 0 initial, apoi la cheia action se calculeaza Q-ul si se returneaza cea mai buna actiune cu ajutorul functiei argMax();

```
1  def computeActionFromValues(self, state):
2          """
3            The policy is the best action in the given state
4            according to the values currently stored in self.values.
5
6            You may break ties any way you see fit.  Note that if
7            there are no legal actions, which is the case at the
8            terminal state, you should return None.
9          """
```

```
        "*** YOUR CODE HERE ***"
        #util.raiseNotDefined()
        if self.mdp.isTerminal(state):
            return None

        possibleActions = self.mdp.getPossibleActions(state)
        QValues = util.Counter()

        for action in possibleActions:
            QValues[action] = self.computeQValueFromValues(state,action)

        #return the best action
        return QValues.argMax()
```

O alta functie care trebuie implementata este runValueIteration, pe care o ruleaza din initializare. Aceasta este functia care se bazeaza pe formula lui Bellman. Agentul folosit la value iteration ia un MDP in constructie si apeleaza functia runValueIteration. Aceasta functie ruleaza pe un numar de iteratii si pentru fiecare stare in cele existente, daca e stare terminala continua, altfel va calcula valoarea maxima calculata cu functia computeQFromValues dintre actiunile legale si va stoca pentru fiecare stare valoarea maxima. Acest nou dictionar creat cu starile ca si chei si valorile maxime i se va atribui lui self.values, actualizandu-se.

```
def runValueIteration(self):
        # Write value iteration code here
        "*** YOUR CODE HERE ***"
        for i in range(self.iterations):
            auxVal = util.Counter()

            for state in self.mdp.getStates():

                # a terminal state has zero future rewards
                if self.mdp.isTerminal(state):
                    continue

                else:
                    #Bellman
                    maxVal = float('-inf')
                    possibleActions = self.mdp.getPossibleActions(state)

                    for action in possibleActions:
                        value = self.computeQValueFromValues(state,action)
                        maxVal = max(value, maxVal)

                    if maxVal != float('-inf'):
                        auxVal[state] = maxVal

            self.values = auxVal
```

3 Question3-Policies

In Question3 avem un grid cu 2 stari terminale, una cu payoff +1 si cealalta cu +10. Problema in aceasta intrebare este ca avem doua cai de a ajunge in aceste puncte finale, unul mai scurt dar e riscul sa cada in -10, si altul mai lung, dar care o rata de a ajunge mai mare. In acest question am avut de investigat valorile care trebuie schimbate in discount, livingReward si noise pentru a potrivi mai multe situatii:

1. **Prefer the close exit (+1), risking the cliff (-10)**

In aceasta situatie am ales un discount cat de mic deoarece vrem ca agentul sa ajunga in 1, nu in 10 pentru a avea un scor mare, noise-ul este 0 deoarece vrem sa mearga pe acel drum fara sa pice in -10, iar livingReward l-am setat la 0 deoarece nu parcurge asa multi pasi.

```
def question3a():
    answerDiscount = 0.1
    answerNoise = 0
    answerLivingReward = 0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

2. **Prefer the close exit (+1), but avoiding the cliff (-10)**

In aceasta situatie se adauga putin noise pentru a alege sa mearga pe sus, si nu deasupra lui -10. In rest totul ramane la fel.

```
def question3b():
    answerDiscount = 0.1
    answerNoise = 0.1
    answerLivingReward = 0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

3. **Prefer the distant exit (+10), risking the cliff (-10)**

Aici, fata de cazul a se adauga discount pentru a-l face pe agent sa se indrepte spre 10.

```
def question3c():
    answerDiscount = 0.9
    answerNoise = 0
    answerLivingReward = 0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

4. **Prefer the distant exit (+10), avoiding the cliff (-10)**

In acest caz, fata de situatia de la b se adauga discount pentru a se indrepta spre 10.

```
def question3d():
    answerDiscount = 0.9
    answerNoise = 0.1
    answerLivingReward = 0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

5. **Avoid both exits and the cliff (so an episode should never terminate)**

In ultima situatie un episod trebuie sa ruleze de o infinitate de ori deci livingRewardul este mai mare ca 10 pentru a se misca incontinuu, iar pentru a evita cliff-ul trebuie ca noise-ul trebuie sa fie mai mare ca 0 pentru a nu alege sa se indrepte inspre -10.

```
def question3e():
    answerDiscount = 0.9
    answerNoise = 0.1
    answerLivingReward = 100
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

4 Question5-Q-Learning

Ultimul question pe care l-am implementat se bazeaza pe invatarea agentului din experiente, astfel incat acesta invata prin erori din interactiunea cu mediul prin functia update(state, action, nextState, reward).

Prin urmare, in functia de initializare am setat Q-values ca un dictionar cu elementele 0, ca si in cazurile anterioare.

Prima functie implementata este getQValue prin care returnam Q(state,action) precum ni se cere in enunt.

```
def getQValue(self, state, action):
        """
          Returns Q(state,action)
          Should return 0.0 if we have never seen a state
          or the Q node value otherwise
        """
        "*** YOUR CODE HERE ***"
        #util.raiseNotDefined()
        return self.Q[(state,action)]
```

A doua functie ceruta este computeValueFromQValues, unde se parcurg actiunile legale si se calculeaza maximul dintre Q values.

```
def computeValueFromQValues(self, state):
        """
          Returns max_action Q(state,action)
          where the max is over legal actions.  Note that if
          there are no legal actions, which is the case at the
          terminal state, you should return a value of 0.0.
        """
        "*** YOUR CODE HERE ***"
        #util.raiseNotDefined()

        maxVal = float('-inf')
        legalActions = self.getLegalActions(state)

```

```
        for action in legalActions:
            maxVal = max(maxVal, self.getQValue(state,action))

        if maxVal == float('-inf'):
            return 0.0
        else:
            return maxVal
```

In continuare se trece la functia computeActionFromQValues, in care se calculeaza cea mai buna actiune pe care sa o faca la o anumita stare dintre cele legale. Se adauga intr-o lista toate actiunile care sunt best si se alege random dintre acestea deoarece intr-o stare particulara se poate intampla ca una din actiunile nevazute din trecute sa fie optima. Ca si coonditie de verificare daca o anumita stare este cea mai buna am ales sa calculez pt fiecare actiune care are o anumita stare Q value si daca corespunde cu cel care este salvat deja, aceasta este ce cautam.

```
def computeActionFromQValues(self, state):
        """
          Compute the best action to take in a state.  Note that if there
          are no legal actions, which is the case at the terminal state,
          you should return None.
        """
        "*** YOUR CODE HERE ***"
        #util.raiseNotDefined()

        legalActions = self.getLegalActions(state)
        bestActions = []

        if not legalActions:
            return None

        for action in legalActions:
            if self.computeValueFromQValues(state) == self.getQValue(state,action):
                bestActions.append(action)

        return random.choice(bestActions)
```

Ultima functie ce trebuie implementata pentru acest Question este update prin care se foloseste de Q value vechi si noua valoare a Q-ului pentru urmatoarea stare, dupa cum urmeaza:

```
def update(self, state, action, nextState, reward):
        """
          The parent class calls this to observe a
          state = action => nextState and reward transition.
          You should do your Q-Value update here

          NOTE: You should never call this function,
          it will be called on your behalf
        """
        "*** YOUR CODE HERE ***"
```

```
        #util.raiseNotDefined()

        oldQ= self.Q[(state,action)]
        newVal = self.computeValueFromQValues(nextState)
        newQ = reward + (self.discount * newVal)

        self.Q[(state,action)] = (1-self.alpha) * oldQ + self.alpha * newQ
```