UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Markov Decision Process & Reinforcement learning

Inteligență Artificială

Autor: Marcu Ariana-Mălina
Grupa: 30232/2

Facultatea de Automatică
și Calculatoare

17 Ianuarie 2024

Cuprins

1 Introducere

În acest proiect, am implementat value iteration și Q-learning, iar agenții au fost testați în gridworld. Învățarea prin consolidare (Reinforcement Learning) este un tip de învățare automată care permite mașinilor și agenților software să determine automat comportamentul ideal într-un context specific, în scopul maximizării performanței. Este necesară furnizarea unui feedback simplu sub formă de recompense pentru ca agentul să învețe comportamentul dorit.

Există numeroși algoritmi care abordează această problemă. Un agent trebuie să decidă cea mai bună acțiune de selectat pe baza stării sale curente. Atunci când acest pas este repetat, problema este cunoscută sub numele de Proces de Decizie Markov (Markov Decision Process - MDP). Acesta cuprinde:

- un set de stări posibile
- un set de modele
- un set de acțiuni posibile
- o funcție de recompensă cu valori reale
- o politică - soluția Procesului MDP

2 Question 1: Value Iteration

În această întrebare, se cere implementarea unui agent de învățare automată, numit ValueIterationAgent, care utilizează tehnica de iterare a valorii pentru a planifica acțiunile într-un Mediu de Decizie Markovian (MDP). Iterarea valorii calculează estimările valorilor optime pentru k etape. În plus față de runValueIteration, am implementat următoarele metode pentru ValueIterationAgent:

computeActionFromValues(state) calculează cea mai bună acțiune conform funcției de valoare dată de self.values.

computeQValueFromValues(state, action) returnează valoarea Q a perechii (stare, acțiune) dată de funcția de valoare din self.values.

Am realizat implementările în valueIterationAgents.py:

```
def runValueIteration(self):
    self.values = util.Counter()
    states = self.mdp.getStates()
    for k in range(self.iterations):
        newValues = util.Counter()
        for state in states:
            if not self.mdp.isTerminal(state):
                qValues = [self.computeQValueFromValues(state, action) for action in
    self.mdp.getPossibleActions(state)]
                newValues[state] = max(qValues, default = 0)
        self.values = newValues
```

Funcția ***runValueIteration*** realizează iterarea valorii în cadrul unui agent de învățare prin reinforcement learning.

- se inițializează dicționarul values cu valori implicite de zero
- se obține lista stărilor din MDP-ul asociat agentului
- se iterează de un număr specificat de ori

- se inițializează un nou dicționar newValues pentru stocarea noilor valori ale stărilor
- pentru fiecare stare ne-terminală, se calculează valorile Q pentru acțiunile posibile
- se actualizează newValues cu maximul dintre valorile Q calculate
- se actualizează dicționarul values al agentului cu noile valori calculate

```
def computeQValueFromValues(self, state, action):
        def rewardFunc(s, a, s_):
            return self.mdp.getReward(s, a, s_) + self.discount * self.getValue(s_)
        statesAndProbs = self.mdp.getTransitionStatesAndProbs(state, action)
        return sum(p * rewardFunc(state, action, s) for s, p in statesAndProbs)
```

Funcția ***computeQValueFromValues*** calculează valoarea Q pentru o pereche dată de stare și acțiune utilizând funcția de recompensă așteptată și probabilitățile de tranziție. Valoarea Q reprezintă o estimare a recompenselor anticipate pentru a efectua acțiunea respectivă în starea dată.

```
def computeActionFromValues(self, state):
        def func1(a):
            return self.computeQValueFromValues(state, a)
        if self.mdp.isTerminal(state):
            return None
        return max(self.mdp.getPossibleActions(state), key = func1, default = None)
```

Funcția ***computeActionFromValues*** utilizează computeQValueFromValues pentru a evalua valoarea Q pentru fiecare acțiune posibilă în starea dată. Apoi, returnează acțiunea cu cea mai mare valoare Q, iar în cazul de egalitate, poate face o alegere arbitrară. Dacă starea este terminală, se returnează None.

3 Question 2: Bridge Crossing Analysis

Harta "BridgeGrid" este o hartă tip grid world cu două stări terminale, una cu recompensă mică și alta cu recompensă mare, separate printr-un "pod". Pe fiecare parte a podului se află o prăpastie cu recompensă negativă mare. Agentul începe în apropierea stării cu recompensă mică. Cu discountul implicit de 0.9 și zgomotul implicit de 0.2, politica optimă nu determină agentul să traverseze podul. Este cerut să modific doar unul dintre parametrii de discount sau zgomot astfel încât politica optimă să determine agentul să încerce să traverseze podul. Această modificare am făcut-o în funcția question2() din fișierul analysis.py. (Zgomotul se referă la cât de des ajunge un agent într-o stare succesoare neintenționată atunci când efectuează o acțiune).

```
def question2():
    answerDiscount = 0.9
    answerNoise = 0
    return answerDiscount, answerNoise
```

Am implementat astfel codul deoarece modificarea valorii answerNoise la 0 indică absența zgomotului, ceea ce înseamnă că atunci când agentul face o acțiune, acesta va ajunge întotdeauna în starea intenționată, fără a avea probabilitatea de a ajunge într-o altă stare neintenționată. Valoarea answerDiscount = 0.9 reprezintă un discount de 0.9 pentru problemele de învățare prin consolidare. Discountul este o valoare cuprinsă între 0 și 1 și indică cât de mult agentul își ia în considerare recompensele viitoare în procesul de luare a deciziilor.

4 Question 3: Policies

În această cerință, am avut de setat parametrii discount, noise și living reward pentru un MDP în scopul de a produce politici optime de diferite tipuri. Configurarea valorilor parametrilor pentru fiecare parte ar trebui să aibă proprietatea că, dacă agentul ar urma politica sa optimă fără a fi supus zgomotului, acesta ar realiza comportamentul specificat. Tipurile de politici optime pe care le-am implementat:

1. *Preferă ieșirea apropiată (+1), riscând prăpastia (-10)* - **question3a()** - Un discount foarte mic(0.1) indică că agentul acordă o atenție minimă recompenselor viitoare, punând mai mare accent pe recompensele imediate. Acest lucru este în concordanță cu preferința pentru ieșirea apropiată. Nicio incertitudine (zgomot) în alegerile agentului indică că acesta va alege mereu acțiunea specificată, fără a se abate, așadar, answerLivingReward = 0.

2. *Preferă ieșirea apropiată (+1), evitând totuși prăpastia (-10)* - **question3b()** - Un discount intermediar (0.5) sugerează că agentul are în vedere atât recompensele imediate, cât și cele viitoare, încurajându-l să facă alegeri echilibrate între proximitatea și riscul prăpastiei. Un nivel moderat de zgomot(0.2) indică că există o oarecare incertitudine în alegeri, permițând adaptarea la situații specifice pentru a evita prăpastia. O recompensă mică pentru supraviețuire (-1) încurajează agentul să evite în mod activ prăpastia, deoarece căderea în prăpastie va atrage o recompensă negativă și o încetare a episodului.

3. *Preferă ieșirea îndepărtată (+10), riscând prăpastia (-10)* -**question3c()** - Discount mare (0.9) pentru a acorda importanță recompenselor viitoare. Lipsa zgomotului indică alegerea strictă a acțiunilor. Recompensă mică pentru supraviețuire indică atenție pe recompensele terminale.

4. *Preferă ieșirea îndepărtată (+10), evitând totuși prăpastia (-10)* - **question3d()** - Discount intermediar (0.8) pentru a echilibra recompensele imediate și cele viitoare. Zgomot moderat pentru a permite alegeri precaute. Lipsa recompensei de supraviețuire pentru a se concentra pe recompensele asociate stărilor terminale.

5. *Evită ambele ieșiri și prăpastia* - **question3e()** - Discount intermediar (0.8) pentru a lua în considerare recompensele viitoare. Nivel mai mare de zgomot pentru alegeri variate. Recompensă mare pentru supraviețuire încurajează explorarea prelungită.

5 Question 5: Q-Learning

Mai departe, avem un agent Q-learning, care nu face prea multe, ci învață prin încercare și eroare din interacțiunile cu mediul, folosind metoda sa ***update(state, action, nextState, reward)***. Un șablon al unui agent Q-learning este specificat în QLearningAgent în qlearningAgents.py. Pentru această întrebare, am implementat metodele update, computeValueFromQValues, getQValue și computeActionFromQValues.

Se inițializează obiectul Q-learning cu metoda inițială a clasei părinte și se creează un dicționar qValues pentru a stoca valorile Q. getQValue returnează valoarea Q asociată stării și acțiunii date. computeValueFromQValues calculează valoarea maximă a Q-urilor disponibile pentru o anumită stare. computeActionFromQValues calculează acțiunea asociată celei mai mari valori Q într-o anumită stare. update: obține valoarea Q curentă pentru perechea (stare, acțiune). Calculează valoarea maximă a Q-urilor pentru starea următoare (newValue). Actualizează valoarea Q pentru perechea (stare, acțiune) utilizând regula de actualizare Q-learning.

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

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

def computeValueFromQValues(self, state):
    legalActions = self.getLegalActions(state)
    if not legalActions:
        return 0.0
    return max([self.getQValue(state, action) for action in legalActions])

def computeActionFromQValues(self, state):
    legalActions = self.getLegalActions(state)
    if not legalActions:
        return None
    qValues = [self.getQValue(state, action) for action in legalActions]
    i = qValues.index(max(qValues))
    return legalActions[i]

def update(self, state, action, nextState, reward):
    Val1 = self.getQValue(state, action)
    newValue = self.computeValueFromQValues(nextState)
    self.qValues[(state, action)] = Val1*(1 - self.alpha) + self.alpha
                  (reward + self.discount * newValue)
```

6 Question 7: Bridge Crossing Revisited

La această cerință, tot ce am făcut a fost să pun question8() să returneze imposibil. Dacă un comportament specific nu poate fi obținut pentru nicio configurație a parametrilor, se va afirma că politica este imposibilă prin returnarea șirului 'NOT POSSIBLE'. Astfel, am obținut un punct și la această cerință.