UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

MDP și Reinforcement Learning

Inteligență Artificială

Autor: Sabău Oana-Maria
Grupa: 30232

Facultatea de Automatică
și Calculatoare

18 Ianuarie 2024

Cuprins

1 Introducere

1.1 Prezentare

Inteligența artificială reprezintă simularea proceselor de inteligență umană de către mașini, în special de către sisteme informatice. Printre aplicațiile specifice ale inteligenței artificiale se numără sistemele experte, prelucrarea limbajului natural, recunoașterea vorbirii și vederea artificială.[3]

Un proces decizional Markov, sau MDP, este un cadru matematic pentru modelarea procesului decizional în situații în care rezultatele sunt incerte. MDP-urile sunt utilizate în mod obișnuit în inteligența artificială (AI) pentru a ajuta agenții să ia decizii în medii complexe și incerte. MDP se bazează pe conceptul de lanț Markov, care este un model matematic al unui sistem în care starea viitoare a sistemului este determinată de starea sa actuală. Agentul ia o decizie la fiecare stare, care determină următoarea stare a sistemului. Scopul agentului este de a găsi o politică, care este un set de decizii, care va maximiza un anumit obiectiv sau recompensă. MDP-urile sunt instrumente puternice pentru modelarea procesului decizional, dar sunt, de asemenea, complexe și pot fi dificil de rezolvat. În multe cazuri, nu este posibil să se găsească o politică optimă pentru un MDP. Cu toate acestea, există o varietate de metode care pot fi utilizate pentru a aproxima o politică optimă. Aceste metode includ iterația valorii, iterația politicii și învățarea Q. [1]

Reinforcement Learning (RL) este o tehnică de machine learning (ML) care antrenează software-ul să ia decizii pentru a obține cele mai bune rezultate. Aceasta imită procesul de învățare prin încercare și eroare pe care oamenii îl folosesc pentru a-și atinge obiectivele. Acțiunile software care contribuie la atingerea obiectivului sunt întărite, în timp ce acțiunile care afectează obiectivul sunt ignorate. Algoritmii RL utilizează o paradigmă de recompensă și pedeapsă în timp ce procesează datele. Ei învață din feedback-ul fiecărei acțiuni și descoperă singuri cele mai bune căi de procesare pentru a obține rezultatele finale. Reinforcement Learning este o metodă puternică pentru a ajuta sistemele de inteligență artificială (AI) să obțină rezultate optime în medii nevăzute.[2]

1.2 Motivație

Utilizarea MDP și a Reinforcement Learning-ului în domeniul inteligenței artificiale este motivată de abordarea problemelor complexe și incerte în luarea de decizii, precum și de optimizarea obiectivelor sau aplicabilitatea în diverse domenii.

2 Question 1 : Value Iteration

Value Iteration este o metodă de găsire a funcției de valoare optimă prin rezolvarea iterativă a ecuațiilor Bellman. Aceasta utilizează conceptul de programare dinamică pentru a menține o funcție de valoare care se apropie de funcția de valoare optimă, îmbunătățind iterativ până când aceasta converge către (sau aproape de aceasta).[4] Value Iteration reprezintă o metodă eficientă pentru rezolvarea MDP-urilor, furnizând soluții aproximative sau exacte pentru problemele de luare a deciziilor în medii incerte și complexe.

Pentru această parte a proiectului, am implementat metoda descrisă mai sus pornind de la explicațiile primite la cursul și laboratorul de Inteligență Artificială, dar și din cursurile celor de la Berkeley.

Atașez rezolvarea scrisă în cod Python:

```
def runValueIteration(self):
    # Write value iteration code here

    for k in range(0, self.iterations):
        QValues = util.Counter()
        for state in self.mdp.getStates():
            if self.mdp.isTerminal(state):
                continue
            QValues[state] = self.computeQValueFromValues(state,
                self.computeActionFromValues(state))

        self.values = QValues


def computeQValueFromValues(self, state, action):
    """
      Compute the Q-value of action in state from the
      value function stored in self.values.
    """

    #compute the QValue for an action in state (I'm using list comprehension)

    QValueList=[ probability * (self.mdp.getReward(state, action, nextState) +
        self.discount * self.getValue(nextState))
          for nextState, probability in
              self.mdp.getTransitionStatesAndProbs(state, action)]

    return sum(QValueList)


def computeActionFromValues(self, state):
    """
      The policy is the best action in the given state
      according to the values currently stored in self.values.

      You may break ties any way you see fit. Note that if
      there are no legal actions, which is the case at the
      terminal state, you should return None.
    """

    if self.mdp.isTerminal(state):
        return None
    else:
        actions = self.mdp.getPossibleActions(state)
        QValues = {action: self.computeQValueFromValues(state, action) for action
            in actions} # for each action, compute the QValue and store it in a
            dictionary accordingly (I'm using dictionary comprehension)
    return max(QValues, key=QValues.get)
```

Am implementat 3 metode pentru acest pas:

- *def computeQValueFromValues(self, state, action)*: Metoda determină valoarea Q pentru o anumită acțiune într-o anumită stare, luând în considerare recompensele, probabilitățile de tranziție.

- *def computeActionFromValues(self, state)* : Această metodă calculează cea mai bună acțiune care trebuie întreprinsă într-o anumită stare pe baza valorilor Q obținute din funcția de valoare curentă stocată în self.values. Aceasta returnează None pentru stările terminale sau acțiunea cu cea mai mare valoare Q pentru stările non-terminale.
- *def runValueIteration(self):* Metoda efectuează un număr specificat de iterații ale algoritmului Value Iteration. La fiecare iterație, aceasta calculează valorile Q pentru fiecare stare non-terminală și actualizează funcția de valoare în consecință. Acțiunea pentru fiecare stare este determinată cu ajutorul metodei *computeActionFromValues(self, state)* .

3 Question 2 : Bridge Crossing Analysis

Se cere modificarea unuia dintre parametrii discount sau noise pentru ca agentul să ajungă în starea finală (să treacă podul). Am modificat parametrul noise, pe care l-am setat la 0.01. Am observat, testând cu mai multe valori, că daca valoarea lui noise este mare, value iteration are o valoare din ce în ce mai mică, iar șansele de succes scad considerabil. Astfel, prin modificarea valorii de la 0.2 la 0.01, agentul trece podul.

4 Question 3 : Policies

La fel ca și la întrebarea precedentă, se cere modificarea parametrilor discount, noise și living reward astfel încât să obținem politicile cerute. În acest caz, am variat parametrii până am ajuns la rezultatele dorite. Parametrii pentru fiecare politică sunt:

- Preferați ieșirea din apropiere (+1), riscând prăpastia (-10) : discount = 0.3, noise = 0.0, living reward = 0.0,
- Preferați ieșirea din apropiere (+1), dar evitați prăpastia (-10) : discount = 0.3, noise = 0.1, living reward = 0.0,
- Preferați ieșirea îndepărtată (+10), riscând prăpastia (-10) : discount = 0.5, noise = 0.01, living reward = 0.0,
- Preferați ieșirea îndepărtată (+10), evitând prăpastia (-10) : discount = 0.4, noise = 0.1, living reward = 0.0,
- Evitați ambele ieșiri și prăpastia (astfel încât un episod să nu se încheie niciodată). : discount = 0.99, noise = 0.0001, living reward = 1,

Un discount mai mare tinde să acorde prioritate recompenselor pe termen lung, în timp ce un nivel de noise mai mare introduce un caracter aleatoriu în acțiunile agentului. Living reward reprezintă recompensa pe care o primește agentul la fiecare pas de timp. Se pare că fiecare politică este concepută pentru a încuraja sau descuraja anumite comportamente ale agentului în funcție de valorile parametrilor specificați. Ajustarea acestor parametri permite ajustarea cu precizie a strategiei agentului în mediul înconjurător.

5 Question 5 : Q-Learning

Algoritmul Q-value este un concept fundamental în Reinforcement Learning și este adesea asociat cu algoritmul Q-learning, un algoritm de învățare prin întărire fără model. Valoarea Q, cunoscută și sub numele de funcția acțiune-valoare, reprezintă recompensa cumulativă așteptată pe care un agent o poate obține dacă întreprinde o anumită acțiune într-o anumită stare și urmează o anumită politică ulterioară.

Implementarea funcțiilor date se regăsește mai jos:

```
def __init__(self, **args):
    "You can initialize Q-values here..."
    ReinforcementAgent.__init__(self, **args)

    self.QValues = util.Counter()

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
    """

    return self.QValues[(state, action)]


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.
    """

    values = [self.getQValue(state, action) for action in
        self.getLegalActions(state)]
    return max(values) if values else 0.0

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.
    """

    selectedAction = next(
        (action for action in self.getLegalActions(state) if self.getValue(state)
            == self.getQValue(state, action)),
        None )

    return selectedAction

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
    """
```

```
        updatedQValue = (1 - self.alpha) * self.getQValue(state, action) + self.alpha
            * (reward + (self.discount * self.getValue(nextState)))
        self.QValues[(state, action)] = updatedQValue
```

6 Question 6 : Epsilon Greedy

Epsilon-greedy este o strategie utilizată în mod obișnuit în RL pentru a echilibra explorarea și exploatarea în procesul decizional. În contextul învățării Q sau al altor algoritmi de îreinforcement learning, epsilon-greedy este utilizat pentru a determina modul în care agentul alege acțiunile.

Epsilon Greedy a fost implementat astfel:

```
    def getAction(self, state):
        """
          Compute the action to take in the current state. With
          probability self.epsilon, we should take a random action and
          take the best policy action otherwise. Note that if there are
          no legal actions, which is the case at the terminal state, you
          should choose None as the action.

          HINT: You might want to use util.flipCoin(prob)
          HINT: To pick randomly from a list, use random.choice(list)
        """
        # Pick Action
        legalActions = self.getLegalActions(state)
        action = None

        if len(legalActions) == 0:
            return None
        if util.flipCoin(self.epsilon):
            action = random.choice(legalActions)
        else:
            action = self.getPolicy(state)

        return action
```

Strategia epsilon-greedy este o modalitate de a echilibra explorarea și exploatarea prin intermediul unui parametru numit epsilon. Cu o probabilitate epsilon, agentul alege să exploreze prin selectarea unei acțiuni aleatorii. Cu probabilitatea 1 - epsilon, agentul alege să exploateze prin selectarea acțiunii cu cea mai mare valoare estimată (valoarea Q în contextul învățării Q).

7 Question 7 : Bridge Crossing Revisited

Se cere să modificăm parametrii answerEpsilon și answerLearningRate astfel încât politica să fie învățată după 50 de iterații. Acest lucru nu este posibil deoarece numărul de iterații este prea mic, deci se va returna 'NOT POSSIBLE'.

8 Question 8 : Q-Learning and Pacman

Se pare că dispunem un set detaliat de instrucțiuni pentru a rula și testa un agent de învățare Q pentru Pacman. Întrebarea subliniază importanța antrenării corespunzătoare a agentului cu parametri adecvați pentru a asigura o învățare eficientă și succesul ulterior în jocurile de test. Agentul trebuie să câștige cel puțin 80 la sută din timpul jocurilor de test pentru a primi creditul complet. Această funcționalitate a fost implementată odată cu algoritmul Q-Value deoarece agentul se descurcă, iar testele sunt trecute.

9 Question 9 : Approximate Q-Learning

Învățarea Q aproximativă este o extensie a învățării Q tradiționale care este deosebit de utilă în cazurile în care spațiul de stare este prea mare pentru a fi reprezentat în mod explicit. În învățarea Q aproximativă, se utilizează o tehnică de aproximare a funcțiilor pentru a estima valorile Q. În loc să se stocheze în mod explicit valorile Q pentru fiecare pereche stare-acțiune, se învață o funcție care face o corespondență între perechile stare-acțiune și valorile Q corespunzătoare.

Pornind de la explicațiile de la cerință, am reușit să implementez și această ultimă cerință:

```
def getQValue(self, state, action):
    """
      Should return Q(state,action) = w * featureVector
      where * is the dotProduct operator
    """

    features = self.featExtractor.getFeatures(state, action)
    QValue = [self.weights[feature] * features[feature] for feature in features]
    return sum(QValue)

def update(self, state, action, nextState, reward):
    """
       Should update your weights based on transition
    """

    features = self.featExtractor.getFeatures(state, action)
    difference = reward + (self.discount * self.getValue(nextState) -
        self.getQValue(state, action))
    for feature in features:
        self.weights[feature] += self.alpha * features[feature] * difference

def final(self, state):
    "Called at the end of each game."
    # call the super-class final method
    PacmanQAgent.final(self, state)

    # did we finish training?
    if self.episodesSoFar == self.numTraining:
        # you might want to print your weights here for debugging
        "*** YOUR CODE HERE ***"

        pass
```

Am modificat doar 2 din cele 3 metode predefinite pentru ca agentul să funcționeze conform așteptărilor, iar modificarea acestora a constat prin implementarea ecuațiilor cerute.

10 Concluzii

MDP-urile oferă un cadru formal pentru modelarea problemelor de luare a deciziilor, în timp ce RL este o abordare algoritmică pentru rezolvarea acestor probleme. Utilizarea aproximării funcțiilor în învățarea Q este esențială pentru gestionarea spațiilor de stare mari sau continue. Codul furnizat demonstrează principiile învățării Q cu aproximație funcțională liniară, în care ponderile sunt actualizate iterativ pentru a îmbunătăți predicțiile valorii Q. Astfel am reușit să antrenez un agent pacman. Proiectul a reprezentat însușirea acestor noi informații într-o formă relativ ușoară deoarece efortul depus a fost mai mic decât la primul proiect, singura parte dificilă fiind rezolvarea pe hârtie a acestora (șanse de a greși la calcul, neverificarea corectă a stărilor - erori umane frecvente)...

Referințe

[1] Autoblocks, *Markov decision process (MDP)*, URL: https://www.autoblocks.ai/glossary/markov-decision-process.

[2] AWS, *What is reinforcement learning?*, URL: https://aws.amazon.com/what-is/reinforcement-learning/.

[3] Nicole Laskowski, *artificial intelligence (AI)*, URL: https://www.techtarget.com/searchenterpriseai/definition/AI-Artificial-Intelligence.

[4] Tim Miller, *Value Iteration*, URL: https://gibberblot.github.io/rl-notes/single-agent/value-iteration.html.