Universidad del País Vasco
Euskal Herriko Unibertsitatea

Técnicas de Inteligencia Artificial

Ingeniería Informática de Gestión y Sistemas de Información

Practica 4
Reinforcement Learning

Autor(es):
Xabier Gabiña
Diego Montoya

3 de diciembre de 2024

Índice general

Índice de Códigos

1. Introducción

2. Ejercicios

2.1. Iteración de valores

Descripción

El primer ejercicio se trata de implementar un agente que realice la iteración de valores. Un agente de Iteración de Valores toma un proceso de decisión de Markov en la inicialización y ejecuta la iteración de valores durante un número dado de iteraciones utilizando el factor de descuento proporcionado.

Implementación

```
class ValueIterationAgent(ValueEstimationAgent):
    """
        * Please read learningAgents.py before reading this.*

        A ValueIterationAgent takes a Markov decision process
        (see mdp.py) on initialization and runs value iteration
        for a given number of iterations using the supplied
        discount factor.
    """
    def __init__(self, mdp, discount = 0.9, iterations = 100):
        """
          Your value iteration agent should take an mdp on
          construction, run the indicated number of iterations
          and then act according to the resulting policy.

          Some useful mdp methods you will use:
              mdp.getStates()
              mdp.getPossibleActions(state)
              mdp.getTransitionStatesAndProbs(state, action)
              mdp.getReward(state, action, nextState)
              mdp.isTerminal(state)
        """
        self.mdp = mdp
        self.discount = discount
        self.iterations = iterations
        self.values = util.Counter() # A Counter is a dict with default 0
        self.runValueIteration()

    def runValueIteration(self):
        # Write value iteration code here
        for _ in range(self.iterations):
            new_values = self.values.copy()
            for state in self.mdp.getStates():
                if not self.mdp.isTerminal(state):
                    new_values[state] = max(self.computeQValueFromValues(state, action) for action
 in self.mdp.getPossibleActions(state))
            self.values = new_values


    def getValue(self, state):
        """
          Return the value of the state (computed in __init__).
        """
        return self.values[state]


    def computeQValueFromValues(self, state, action):
        """
        computeQValueFromValues(state, action) devuelve el valor Q del par (estado, acción)
        dado por la función de valor dada por self.values. Nota: Recuerda que para calcular
        el valor de un estado calcularemos los q_values (estado,acción), es decir los valores
        de las acciones posibles para quedarnos con el mayor (max de entre los q_values).
        """
        q_value = 0
        for next_state, prob in self.mdp.getTransitionStatesAndProbs(state, action):
```

```
55              q_value += prob * (self.mdp.getReward(state, action, next_state) + self.discount *
    self.values[next_state])
56          return q_value
57
58
59      def computeActionFromValues(self, state):
60          """
61          computeActionFromValues(state) calcula la mejor acción de acuerdo con la función de
62          valor dada por self.values. Es decir, de acuerdo al estado siguiente al que nos lleva
63          cada acción y que mayor valor tiene. Este método llamará al siguiente.
64          """
65          if self.mdp.isTerminal(state):
66              return None
67
68          return max(self.mdp.getPossibleActions(state), key=lambda x: self.computeQValueFromValues(
    state, x))
69
70
71      def getPolicy(self, state):
72          return self.computeActionFromValues(state)
73
74      def getAction(self, state):
75          "Returns the policy at the state (no exploration)."
76          return self.computeActionFromValues(state)
77
78      def getQValue(self, state, action):
79          return self.computeQValueFromValues(state, action)
```

Código 2.1: Iteración de valores

Comentarios

2.2. Análisis de cruce de puentes

Descripción

Implementación

```
def question2():
  answerDiscount = 0.9
  answerNoise = 0.01
  return answerDiscount, answerNoise

if __name__ == '__main__':
  print('Answers to analysis questions:')
  import analysis
  for q in [q for q in dir(analysis) if q.startswith('question')]:
      response = getattr(analysis, q)()
      print('  Question %s:\t%s' % (q, str(response)))
```

Código 2.2: Análisis de cruce de puentes

Comentarios

La solucion mas sencilla para este ejercicio es el de bajar el ruido por debajo de 0.01. Esto hace que las probabilidades de que el agente cometa un error y se caiga a un acantilado de puntos negativos sean muy bajas. No haria falta tocar el valor gamma ya que al tener un factor alto, lo que conseguimos, es que las recompensas futuras tengan un peso mayor lo que con conviene para no acabar llegando a la casilla +1 en vez de a la +10.

2.3. Q-Learning

Descripción

En este ejercicio se trata de implementar un agente que realice el aprendizaje por refuerzo mediante Q-Learning. Q-Learning es un algoritmo de aprendizaje por refuerzo que aprende una política óptima sin conocer el modelo del entorno. Se basa en ir probando acciones, en un principio de forma aleatoria, y actualizando los valores de la función Q en función de las recompensas recibidas. A medida que se va actualizando la función Q, el agente va aprendiendo la mejor acción a tomar en cada estado y va diminuyendo el numero de acciones aleatorias.

Implementación

```
class QLearningAgent(ReinforcementAgent):
    """
      Q-Learning Agent

      Functions you should fill in:
        - computeValueFromQValues
        - computeActionFromQValues
        - getQValue
        - getAction
        - update

      Instance variables you have access to
        - self.epsilon (exploration prob)
        - self.alpha (learning rate)
        - self.discount (discount rate)

      Functions you should use
        - self.getLegalActions(state)
          which returns legal actions for a state
    """
    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)] if (state, action) in self.qValues else 0.0


    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.
        """
        legalActions = self.getLegalActions(state)
        if not legalActions:
          return 0.0
        return max(self.getQValue(state, action) for action in legalActions)

    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.
        """
        legalActions = self.getLegalActions(state)
        if not legalActions:
            return None

```

```
        best_value = float('-inf')
        best_actions = []

        for action in legalActions:
            q_value = self.getQValue(state, action)
            if q_value > best_value:
              best_value = q_value
              best_actions = [action]
            elif q_value == best_value:
              best_actions.append(action)

        return random.choice(best_actions)

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

        return action

    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
        """
        sample = reward + self.discount * self.computeValueFromQValues(nextState)
        self.qValues[(state, action)] = (1 - self.alpha) * self.getQValue(state, action) + self.
    alpha * sample

    def getPolicy(self, state):
        return self.computeActionFromQValues(state)

    def getValue(self, state):
        return self.computeValueFromQValues(state)
```

Código 2.3: Q-Learning

Comentarios

3. Resultados