Instituto Tecnológico de Aeronáutica – ITA

Graduação

CTC-17 : Inteligência Artificial - Projeto 3

Bruno De Souza Neves , Rahyan Azin

4 de novembro de 2018

1 Resultados

Para o presente trabalho espera-se treinar um agente que , dado um ambiente como o do Wumpus World , consiga executar ações que o levem a estado final desejado da melhor maneira possível , ou seja , caminhos curtos que não tenham Wumpus ou Pits .

1.1 Equação de Bellman

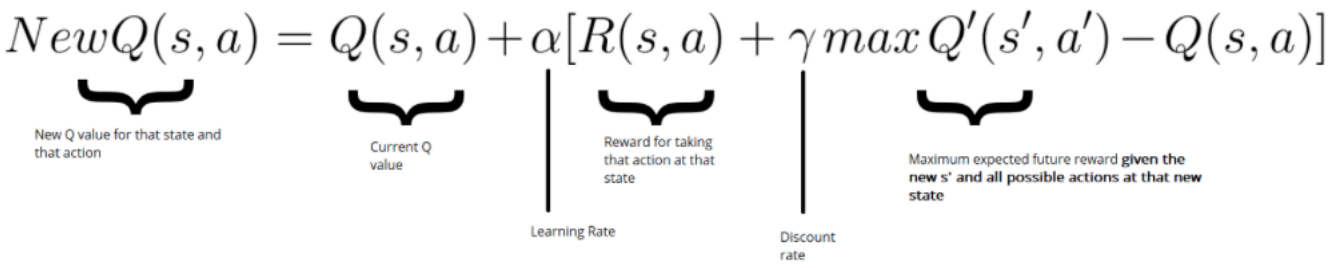

Figura 1: Equacão de Bellman

A equação de Bellman é usada de maneira iterativa para o preenchimento da tabela Q , tal que , dado um estado "s"e uma ação "a", Q[s][a] retorna uma métrica para verificar o quão bom é executar tal ação em tal estado , considerando eventos futuros. R é a matrix que R[s][a] tal que retorna a recompensa imediata de executar a ação "a"no estado "s". γ é o fator de desconto para eventos futuros . O Max no final da equação significa obter o Q máximo de um estado futuro e analisando qual a melhor ação a ser executada em tal estado. α é a learning rate.

1.2 Modelagem

Os estados do problema foram considerados como sendo cada bloco do tabuleiro , de 0 a 31. Como a quantidade de estados não é grande , optou-se por definir de maneira "hard code"as matrizes R (rewards) e T(s,a) (matriz de transição de estado) . Evidentemente que poderia ter sido criada uma classe "estado"e definir métodos para retornar a recompensa imediata, se é parede ou não , se é Gold ou Wumpus , mas , por simplicidade para execução de Bellman, optou-se por modelar da seguinte forma (as linhas são os estados, as colunas as ações , de forma que 0 é 'UP' , 1 é 'DOWN' , 2 é 'LEFT' , 3 é 'RIGHT' e o conteúdo a recompensa; na matriz de transição , as linhas são os estados, as colunas são ações e o conteúdo o próximo estado).

```
R = np.array([[-10,-100,-10,-50],[-10,+100,0,0],[-10,-50,-50,0],
              [-10,0,0,0],[-10,0,0,0],[10,0,0,-50],[-10,-50,0,0],
              [-10,0,-50,-10],[0,0,-10,+100],[-50,0,-100,-50],
              [0,0,+100,0],[0,0,-50,0],[0,-100,0,0],[0,+100,0,-50],
              [-50,0,0,0],[0,0,-50,-10],[-100,0,-10,0],[+100,0,0,0],
              [-50,-50,0,0],[0,0,0,-100],[0,0,0,+100],[0,0,-100,0],
              [-50,-50,+100,0],[0,0,0,-10],[0,-10,-10,0],[0,-10,0,-50],
              [0,-10,0,0],[0,-10,-50,0],[-100,-10,0,0],[+100,-10,0,-50],
              [0,-10,0,0],[0,-10,-50,-10]])

R = R - 0.1 #cada moviemento tem desconto de -0.1

Q = np.zeros((32,4))
```

```
trans_matrix = np.array( [
                      [0,8,0,1],[1,9,0,2],[2,10,1,3],[3,11,2,4],[4,12,3,5],
                      [5,13,4,6],[6,14,5,7],[7,15,6,7],[0,16,8,9],[1,17,8,10],
                      [2,18,9,11],[3,19,10,12],[4,20,11,13],[5,21,12,14],[6,22,13,15],
                      [7,23,14,15],[8,24,8,17],[9,25,16,18],[10,26,17,19],[11,27,18,20],
                      [12,28,19,21],[13,29,20,22],[14,30,21,23],[15,31,22,23],[16,24,24,25],
                      [17,25,24,26],[18,26,25,27],[19,27,26,28],[20,28,27,29],[21,29,28,30],
                      [22,30,29,31],[23,31,30,31]
                      ])
```

O ”bias”do agente (probabilidade de ele ir à direita do movimento escolhido) é implementado da seguinte maneira:

```
def future_value(state):
   v1 = 0.6*Q[state][UP] +0.4*Q[state][RIGHT]
   v2 = 0.6*Q[state][DOWN] + 0.4*Q[state][LEFT]
   v3 = 0.6*Q[state][LEFT] + 0.4*Q[state][UP]
   v4 = 0.6*Q[state][RIGHT] + 0.4*Q[state][DOWN]
   return max([v1,v2,v3,v4])

def getaction(a):
    r = random.random()
    if a == UP:
        if r<=0.6:
            return UP
        else:
            return RIGHT
    if a == DOWN:
        if r<=0.6:
            return DOWN
        else:
            return LEFT
    if a == LEFT:
        if r<=0.6:
            return LEFT
        else:
            return UP
    if a == RIGHT:
        if r<=0.6:
            return RIGHT
        else:
            return DOWN
```

1.3 Treinamento

Para o treinamento , executou-se 1000 episódios do agente interagindo com o ambiente. O paramêtro γ é ajustado para 0.8 , pois apresentou resultados melhores. O vetor scores dará uma ideia de quantas iterações são necessárias para a tabela Q convergir para a política ótima.

```
#treinamento
##########################################
episodes = 1000
scores = []
```

```
#alpha = np.linspace(0,1,episodes)
gamma = 0.8
for e in range(episodes):
    start_state = random.randint(0,number_states-1)
    current_state = start_state
    while current_state not in done_states :
        action = getaction(random.choice(ACTIONS))
        next_state = trans_matrix[current_state][action]
        future_rewards = []
        for act in ACTIONS:
            future_rewards.append(future_value(next_state))
        #bellman equation
        Q[current_state][action] = R[current_state][action] +
            (gamma)*max(future_rewards)
        current_state = next_state
    if np.max(Q) == 0:
        scores.append(0)
    else:
        scores.append(np.sum(Q/np.max(Q)*100))

##########################################
```

1.4 Resultados

* = Espaço Vazio
p = Pit
w = Wumpus
a = Agent
g = Gold

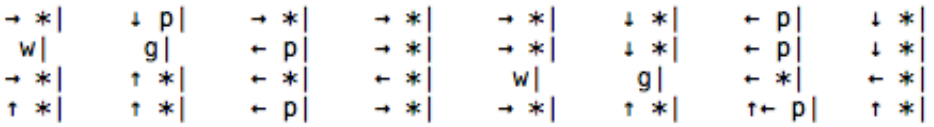

Figura 2: Tabela com setas indicando melhor ação

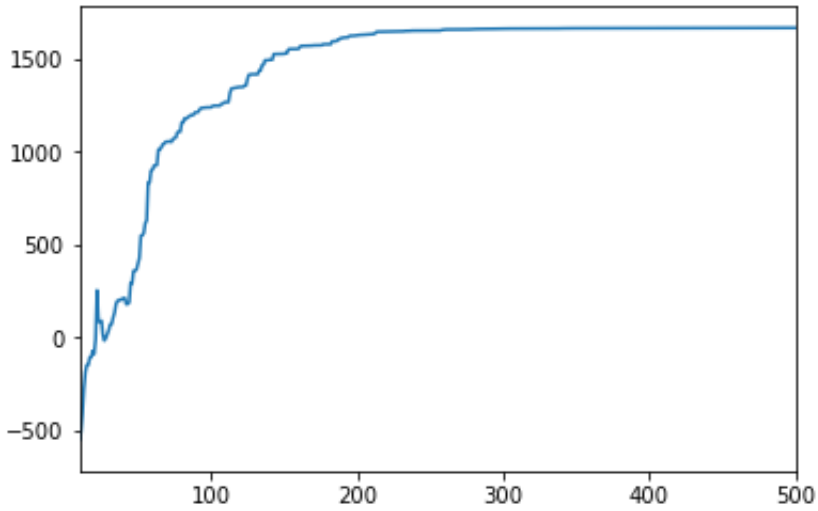

Figura 3: Iterações para convergência (em torno de 200)

Simulações do agente , dado um estado inicial aleatório . Devido sua tendência de escorregar , o agente na simulação 1 não seguiu o caminho ótimo , mas na simulação 2 seguiu a política prevista. Cada matriz é um episódio diferente.

Simulação 1:

```
* p * * * * p *

 w g p * * * a *

 * * * * w g * *

 * * p * * * p *


 * p * * * * p *

 w g p * * a p *

 * * * * w g * *

 * * p * * * p *


 * p * * * * p *

 w g p * a * p *

 * * * * w g * *

 * * p * * * p *


 * p * * * * p *

 w g p * * * p *
```

```
* * * * a g * *

* * p * * * p *
```

Simulação 2

```
* p * * * * a *

w g p * * * p *

* * * * w g * *

* * p * * * p *


* p * * * a p *

w g p * * * p *

* * * * w g * *

* * p * * * p *


* p * * * * p *

w g p * * a p *

* * * * w g * *

* * p * * * p *


* p * * * * p *

w g p * * * p *

* * * * w a * *

* * p * * * p *
```
