Aprendizado por Reforço

Alison de Oliveira Tristão
Otacilio Ribeiro da Silva Netto

18 de outubro de 2024

1 Aprendizado por reforço

Em aprendizado por reforço, o agente interage com o ambiente tomando várias ações. o agente é recompensando quando essas ações geram estados desejados e punidas quando geram estados indesejados. O objetivo do agente é aprender uma estratégia, chamado "policy"que guia as ações do agente para maximizar a recompensa que ele acumula no tempo. Esse processo de tentativa e erro refina o comportamento do agente, permitindo que aprenda comportamentos ótimos no ambiente. Nesse trabalho, vamos falar sobre dois algoritmos de aprendizado por reforço, o Value Iteration e o Q-Learning e seus resultados nos mini games do projeto 3 de "CS 188 | Spring 2020". Os minigames são Gridworld, Cliffwalking e Pacman

2 Markov Decision Processes

Para representar o problema de uma maneira que o agente entenda, precisamos formalizá-lo como um Processo de Decisão de Markov (MDP). Um MDP é um modelo matemático que descreve nosso problema de forma estruturada. Ele representa as interações do agente com o ambiente como um processo de decisão sequencial, onde uma ação afeta a outra. O MDP consiste nas dinâmicas do ambiente:

- Um conjunto finito de estados $s \in S$. No caso do GridWorld, os estados são as posições no plano cartesiano.
- Um conjunto finito de ações $a \in A$. No caso do GridWorld, as ações são os possíveis comandos do agente (norte, sul, leste, oeste).
- Uma função de transição $T(s'|s, a)$ que retorna a probabilidade de alcançar o estado s', dado o estado atual s e a ação atual a. No caso do GridWorld, a probabilidade de ir do estado atual para o próximo depende apenas do estado, da ação e do parâmetro de ruído (vamos considerar 20%):
  - Cima: 80% de chance de ir para o norte, 10% para leste, 10% para oeste.
  - Baixo: 80% de chance de ir para o sul, 10% para leste, 10% para oeste.
  - Direita: 80% de chance de ir para o leste, 10% para o norte, 10% para o sul.

  - Esquerda: 80% de chance de ir para o oeste, 10% para o norte, 10% para o sul.

- Uma função de recompensa $R(s, a, s')$ que retorna uma recompensa escalar baseada no estado atual, na ação tomada e no próximo estado. No caso do GridWorld, a recompensa depende apenas do estado atual, que normalmente retorna o livingReward que é zero por padrão, ou recompensas positivas para o estado final e recompensa negativas para estados que deve evitar.

O MDP do GridWorld é estocástico, porque as ações têm consequências probabilísticas. Isso significa que o agente, ao fazer a mesma ação no mesmo estado várias vezes, pode não terminar sempre no mesmo próximo estado. Isso pode ser observado jogando no modo manual e percebendo que, às vezes, mesmo pressionando "cima", o agente para na esquerda ou na direita. Por conta disso, o Value Iteration leva em consideração essa característica probabilística na sua fórmula.

3 Value Iteration

Value iteration é um algoritmo usado para resolver problemas onde temos conhecimento total dos componentes do MDP. Ele funciona iterativamente melhorando a estimativa do valor de estar em cada estado. Ele faz isso considerando as recompensas imediatas e as recompensas futuras esperadas ao tomar diferentes ações disponíveis. Esses valores são monitorados usando uma tabela de valores, que é atualizada a cada passo. Eventualmente, essa sequência de melhorias converge, resultando em uma política ótima de mapeamento estado → ação que o agente pode seguir para tomar as melhores decisões no ambiente dado.

Inicializar:
$V(s)$ arbitrariamente, para todo $s \in S$.
θ para um pequeno valor positivo.

Loop:
$\Delta \leftarrow 0$
Loop para cada $s \in S$:
$v \leftarrow V(s)$
$V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
$\Delta \leftarrow \max(\Delta, |v - V(s)|)$
Até que $\Delta < \theta$.

Na nossa implementação, utilizamos um número finito de iterações até que os valores sejam suficientemente bons, perto de convergir. Outra diferença é que adotamos a versão 'batch' do algoritmo, em contraste com a versão 'online'. Na abordagem 'batch', o valor do estado futuro é determinado com base nos resultados da iteração anterior.

```
def runValueIteration(self):
    """
      Run the value iteration algorithm. Note that in standard
      value iteration, V_k+1(...) depends on V_k(...)'s.
    """
    values = util.Counter()
    states = self.mdp.getStates() # os estados do MDP s o as
       possiveis posi  es no mundo
    for state in states:
        values[(state, 0)] = 0  # inicializa os valores de todos os
           estados da primeira itera  o com 0
    for k in range(1, self.iterations + 1):
        for state in states:
            acoes = self.mdp.getPossibleActions(state)  # north','west
               ','south','east
            valorMaximo = float('-infinity')
            for acao in acoes:
                value = 0  # variavel pra armazenar o somat rio dos
                   poss veis estados futuros
                proximoEstadoInfos = self.mdp.
                   getTransitionStatesAndProbs(state, acao)  # retorna
                    tuple (estado, probabilidade)
                for proximoEstadoInfo in proximoEstadoInfos:
                    proximoEstado = proximoEstadoInfo[0]
                    probabilidade = proximoEstadoInfo[1]
                    reward = self.mdp.getReward(state, acao,
                       proximoEstado)
                    v = values[(proximoEstado, k - 1)]
                    value += probabilidade * (reward + self.discount *
                        v)
                valorMaximo = max(valorMaximo, value)  # Entre todas
                   as a  es , pega a de maior valor
            if valorMaximo > float('-infinity'):
                values[(state, k)] = valorMaximo  # armazena o valor
                   esperado do estado
            else:
                values[(state, k)] = 0  # o menor valor esperado
                   poss vel    0
    for state in states:
        self.values[state] = values[(state, self.iterations)]  #
           armazena o valor esperado do estado na  ltima  itera  o
```

3.1 Extração de política

Após várias iterações do algoritmo Iteration Value, teremos uma lista de estados, cada um associado a um "valor esperado"que indica o quão vantajoso é estar naquele estado. A partir desses valores esperados, podemos derivar uma "política ótima". Essa política consiste em escolher a melhor ação a ser tomada em cada estado com base nos valores esperados já calculados durante o algoritmo.

A seguir estão as 2 funções que utilizamos para fazer isso:

```
def computeQValueFromValues(self, state, action):
        """
          Compute the Q-value of action in state from the
          value function stored in self.values.
        """
        proximoEstadoInfos = self.mdp.getTransitionStatesAndProbs(
           state, action)
        value = 0
        for proximoEstadoInfo in proximoEstadoInfos:
            proximoEstado = proximoEstadoInfo[0]
            probabilidade = proximoEstadoInfo[1]
            reward = self.mdp.getReward(state, action, proximoEstado)
            v = self.values[proximoEstado]
            value += probabilidade * (reward + self.discount * v)
        return value
```

```
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
        acoes = self.mdp.getPossibleActions(state)
        bestValue = float('-infinity')
        for acao in acoes:
            qValue = self.computeQValueFromValues(state, acao)
            if qValue > bestValue:
                bestValue = qValue
                bestAction = acao
        return bestAction
```

3.2 Resultados: ValueIteration

A seguir são os resultados para diferentes mapas com os parâmetros padrões: Discount = 0.9, noise = 0.2, livingReward = 0

Figura 1: Value Iteration - bookgrid - 5 iterações

Figura 2: Value Iteration - bookgrid - 10 iterações

Figura 3: Value Iteration - CliffGrid - 5 iterações

Figura 4: Value Iteration - CliffGrid - 10 iterações

Nesse mapa é interessante notar que ao convergir, todos os estados próximos dos "abismos"estão apontando para cima, embora o caminho mais rápido fosse ir reto para a direita, em direção ao estado final. Olhando especificamente o primeiro estado a esquerda do estado terminal, o que tem valor esperado de 6.79, ele prefere ir para cima, pois embora a chegada esteja logo a sua direita, por conta de o jogo ser estocástico e existir uma probabilidade de ele cair no abismo caso vá para a direita, ele prefere ir para cima, pois é um caminho mais seguro. Na conta do valor esperado isso vem do

somatório que leva em conta todos os possíveis estados futuros dado uma ação:

$$\sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$$

Para o mapa BridgeGrid, não foi possível vencer o mapa com os parâmetros padrões, tendo uma performance muito ruim nela.

Figura 5: Value Iteration - BridgeGrid - 100 iterações

Para conseguir vencer esse mapa, foi necessário modificar os parâmetros da seguinte forma: livingReward = 10, discount = 1, noise = 0.1

Figura 6: Value Iteration - BridgeGrid - 100 iterações

É interessante observar o resultado do algoritmo após 2 iterações. Na primeira iteração, todos os estados permanecem com valor zero, exceto o estado terminal, que tem uma recompensa imediata de 1.0. Já na segunda iteração, como o estado terminal é o único com um valor definido, ele começa a propagar sua pontuação para os estados anteriores.

Um ponto curioso é o valor de "0.72"no estado próximo ao terminal. Embora se esperasse um valor de 0.8 devido à probabilidade de mover para a direita, a explicação para esse valor vem da equação

$$p(s', r \mid s, a)[r + \gamma V(s')] = 0.8[0 + 0.9(1)] = 0.72$$

A diferença reflete o efeito da taxa de desconto (γ) aplicada à recompensa futura.

Figura 7: Value Iteration - MazeGrid - 2 iterações

4 Q-Learning

O Q-Learning é um algoritmo de aprendizado por reforço que permite que um agente aprenda a tomar decisões ótimas em um ambiente. Ele busca maximizar a recompensa total que o agente pode receber ao interagir com esse ambiente. O principal objetivo do Q-Learning é aprender uma função de ação-valor $Q(s, a)$, que estima o valor de tomar uma ação a em um estado s.

Assim, ele opera forma iterativa, onde o agente atualiza a estimativa de Q com base nas recompensas recebidas e nas ações futuras. A atualização da função Q é feita através da seguinte fórmula:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right)$$

Onde: - s: estado atual. - a: ação executada. - r: recompensa recebida após executar a ação. - s': novo estado resultante da ação. - α: taxa de aprendizado ($0 < \alpha$ 1). - γ: fator de desconto ($0 < \gamma < 1$).

Abaixo está um pseudocódigo simples para ilustrar o funcionamento do Q-Learning:

Inicializar:

- $Q(s,a)$ arbitrariamente, para todo $s \in S$ e $a \in A(s)$
- θ para um pequeno valor positivo
- α para a taxa de aprendizado
- γ para o fator de desconto
- Política $\pi(s)$ que seleciona as ações

Loop (para cada episódio):

- $\Delta \leftarrow 0$
- $s \leftarrow$ estado inicial

Loop para cada passo do episódio (enquanto s não for terminal):

- Escolher a ação a de acordo com a política ϵ-greedy de $Q(s,a)$
- Executar a ação a, observar a recompensa r e o novo estado s'
- $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
- $\Delta \leftarrow \max(\Delta, |r + \gamma \max_{a'} Q(s',a') - Q(s,a)|)$
- $s \leftarrow s'$

Até que $\Delta < \theta$ ou o episódio termine.

No nosso algoritmo, consideramos apenas o fim do episodio como parada, no qual temos:

```
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.
    """
    max_q_value = float('-inf')
    for action in self.getLegalActions(state):
        max_q_value = max(max_q_value, self.getQValue(state, action))
    return max_q_value if max_q_value != float('-inf') 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.
    """
    max_q_value = self.computeValueFromQValues(state)
    actions = self.getLegalActions(state)
    best_actions = []

    for action in actions:
        if self.getQValue(state, action) == max_q_value:
            best_actions.append(action)

    if len(best_actions) == 1:
        return best_actions[0]
    else:
        return best_actions[np.random.randint(0, len(best_actions))]

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
    """
    self.Q_values[(state, action)] += self.alpha * (reward + self.
       discount * self.computeValueFromQValues(nextState) - self.
       Q_values[(state, action)])
```

4.1 Exploração e Exploração: Estratégia ϵ-greedy

A estratégia ϵ-greedy é uma abordagem comum utilizada no Q-Learning para equilibrar a exploração, melhorando a politica otima explorando decisões "aleatórias". O objetivo é permitir que o agente explore novas ações, enquanto também aproveita as ações que já têm um valor conhecido. A implementação da estratégia ϵ-greedy funciona da seguinte forma:

- Definir um valor ϵ: Este valor representa a probabilidade de escolher uma ação aleatória. Tipicamente, ϵ é um pequeno número positivo, como 0.1 (10%).
- Escolher a ação:
  - Com probabilidade $1 - \epsilon$: Escolher a ação a que maximiza $Q(s, a)$ (exploração).
  - Com probabilidade ϵ: Escolher uma ação aleatória (exploração).

Para melhorar a política do agente ao longo do tempo, é comum reduzir o valor de ϵ gradualmente, permitindo mais exploração no início do treinamento e mais exploração à medida que o agente se torna mais experiente.

Sabendo disso, implementamos nossa ação levando em conta a possibilidade de tomarmos uma decisão aleatória a cada passo:

```
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

    flip = np.random.rand()
    if flip < self.epsilon:
        action = legalActions[np.random.randint(0, len(legalActions))]
    else:
        action = self.computeActionFromQValues(state)

    return action
```

4.2 Resultados: Q-Learning

A seguir são os resultados para diferentes mapas com os parâmetros padrões: Discount = 0.9, noise = 0.2, livingReward = 0

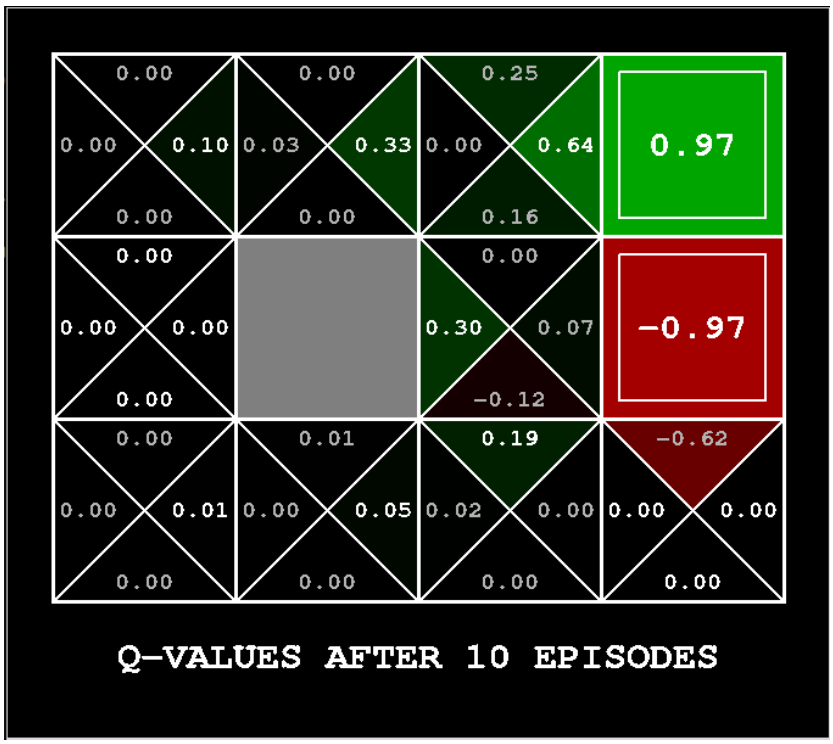

Figura 8: qlearning - bookgrid - 10 iterações

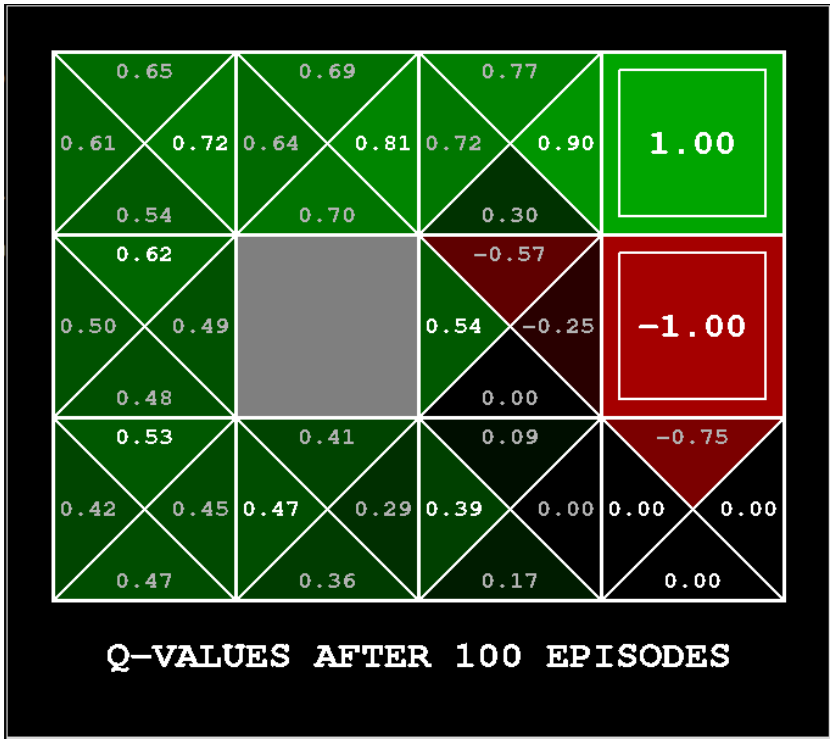

Figura 9: qlearning - bookgrid - 100 iterações

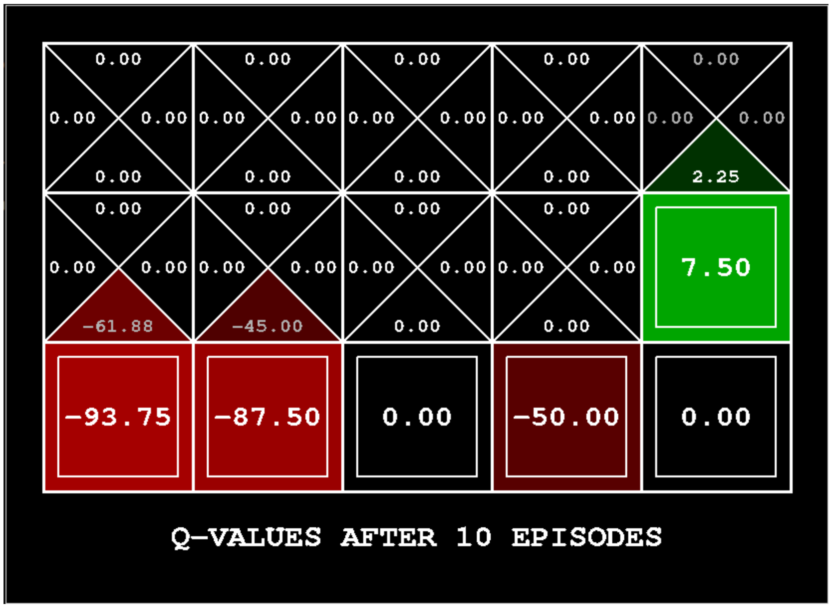

Figura 10: qlearning - CliffGrid - 10 iterações

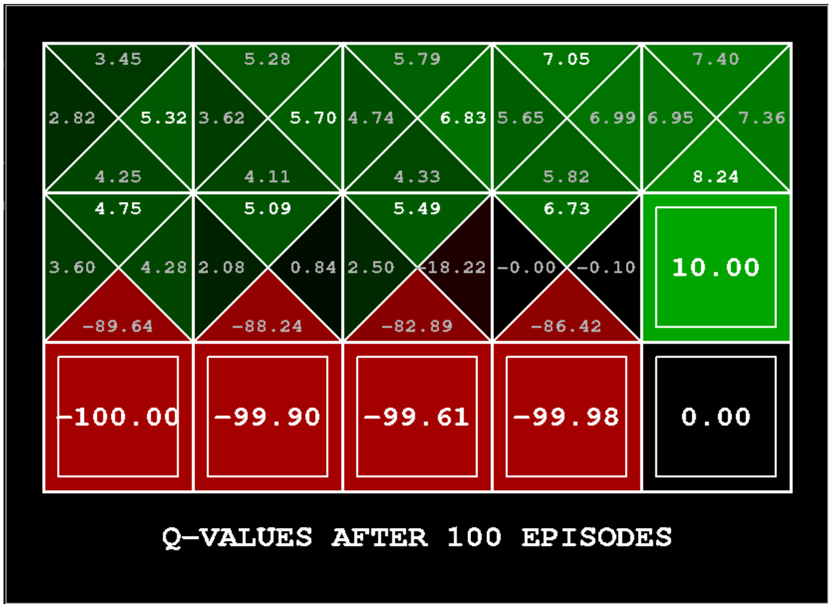

Figura 11: qlearning - CliffGrid - 10 iterações

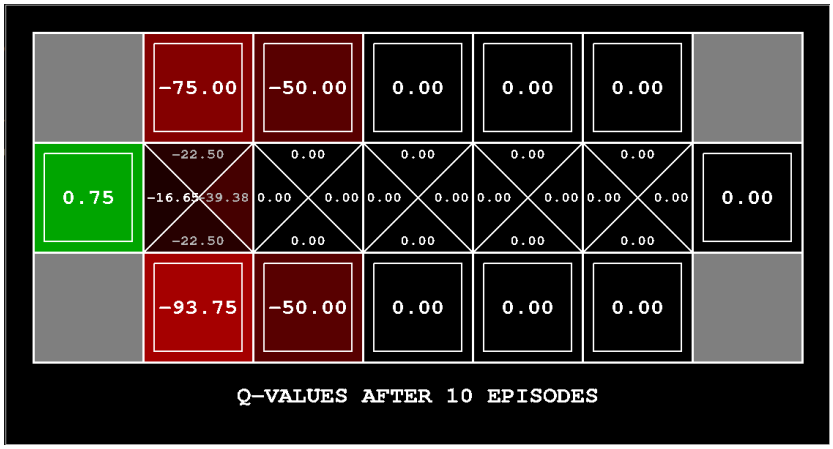

Figura 12: qlearning - BridgeGrid - 10 iterações

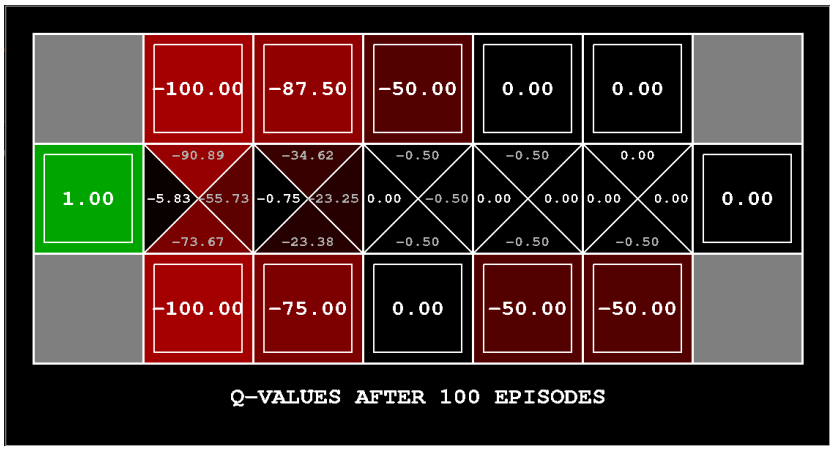

Figura 13: qlearning - BridgeGrid - 100 iterações

Nesse mundo, usando q-learning, não conseguimos fazer com que ele chegasse ao final, mesmo diminuindo o ruido para zero e rodando durante 1000 iterações. O fato de possuir uma recompensa no ponto inicial faz com que o indivíduo volte para atrás.

Referências

5 Método Sem Política (Q Learning)

O método Q-learning se apresenta como um método sem política explícita, no qual não considera diretamente uma regra fixa de ação, levando em conta apenas a ação escolhida de forma determinística. Dessa forma, o método não leva em consideração os "riscos"associados a realizar uma ação, sendo guiado apenas pela maior recompensa esperada. Isso pode ser observado no seguinte contexto: na simulação desenvolvida, temos um mundo em grade onde o indivíduo pode tomar 4 decisões por estado (norte, sul, leste e oeste). A cada movimento, ele recebe uma recompensa de -1, e ao cair no penhasco, recebe uma punição de -100 e é transportado para o início. O jogo termina

quando ele chega ao final, e a política ótima é alcançar o objetivo percorrendo a menor distância possível, ao longo do caminho situado na beira do penhasco, como mostrado na figura abaixo.

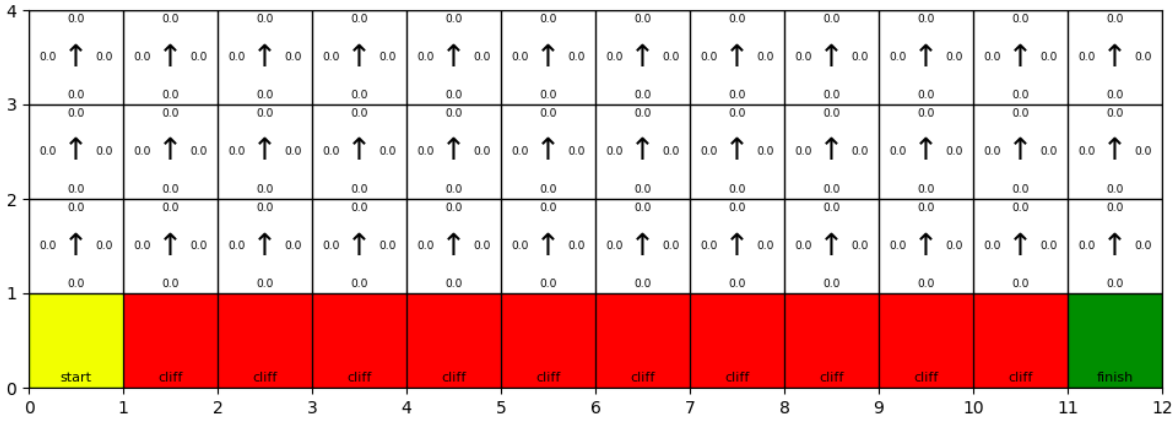

Figura 14: CliffWorld

O algoritmo Q-learning acaba encontrando essa política de forma relativamente rápida; no entanto, o fator de exploração faz com que, ocasionalmente, o agente caia no penhasco, gerando ruído nas iterações. Isso pode ser observado no gráfico:

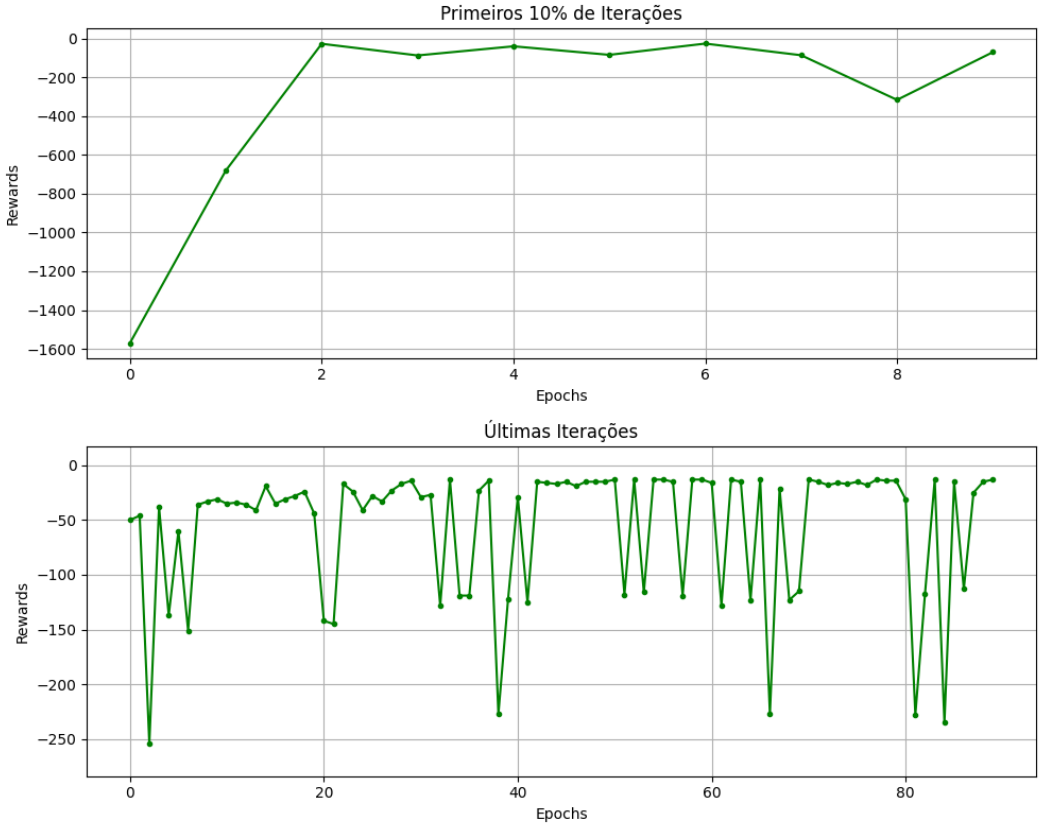

Figura 15: Politica Ótima - CliffWorld

A politica otima encontrada para cada espaço apos 100 epocas pode vista aqui:

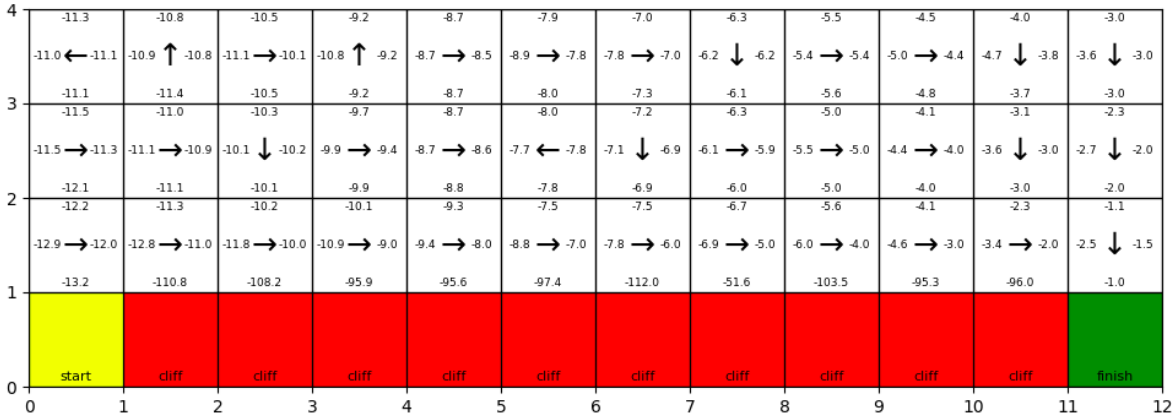

Figura 16: QValues - CliffWorld

6 PacMan

Aplicar o algoritmo de Q-Learning no PacMan foi bem-sucedido em pequenos mapas. No entanto, ele não obteve o mesmo sucesso em mapas maiores, pois a abordagem simples de considerar a configuração exata do jogo como estados se torna ineficiente e não escala bem. Isso ocorre porque cada configuração do mapa, que inclui a posição dos fantasmas e a posição das gemas, é tratada como um estado distinto, com seus próprios Q-values. Essa abordagem resulta em um grande número de estados, dificultando o aprendizado eficaz em ambientes mais complexos. Basicamente, o agente não consegue generalizar que colidir com um fantasma é prejudicial em todas as posições, o que limita a sua eficiente em mapas maiores

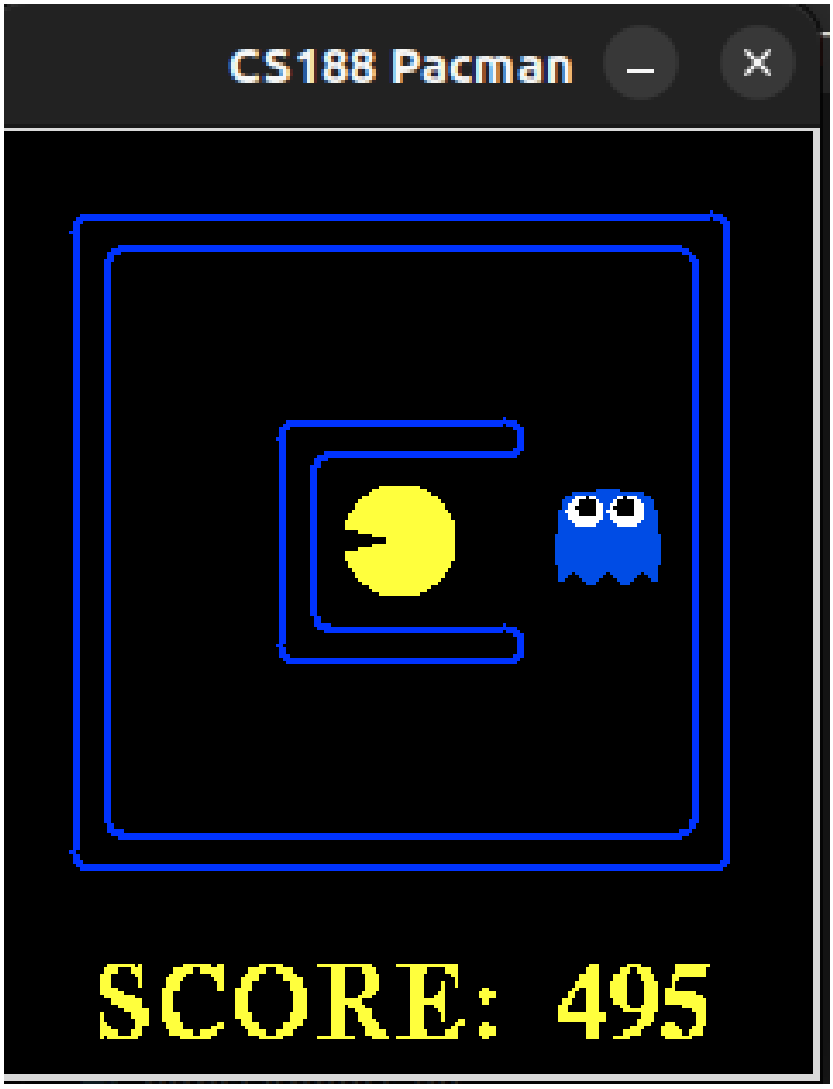

Figura 17: Pacman - smallgrid

Reinforcement Learning Status:
Completed 2000 out of 2000 training episodes
Average Rewards over all training: -75.19
Average Rewards for last 100 episodes: 256.85
Episode took 0.35 seconds
Training Done (turning off epsilon and alpha)

Scores: 499.0, 495.0, 499.0, 495.0, 503.0, 499.0, 503.0, 503.0, 495.0, 499.0
Win Rate: 10/10 (1.00)
Record: Win, Win, Win, Win, Win, Win, Win, Win, Win, Win

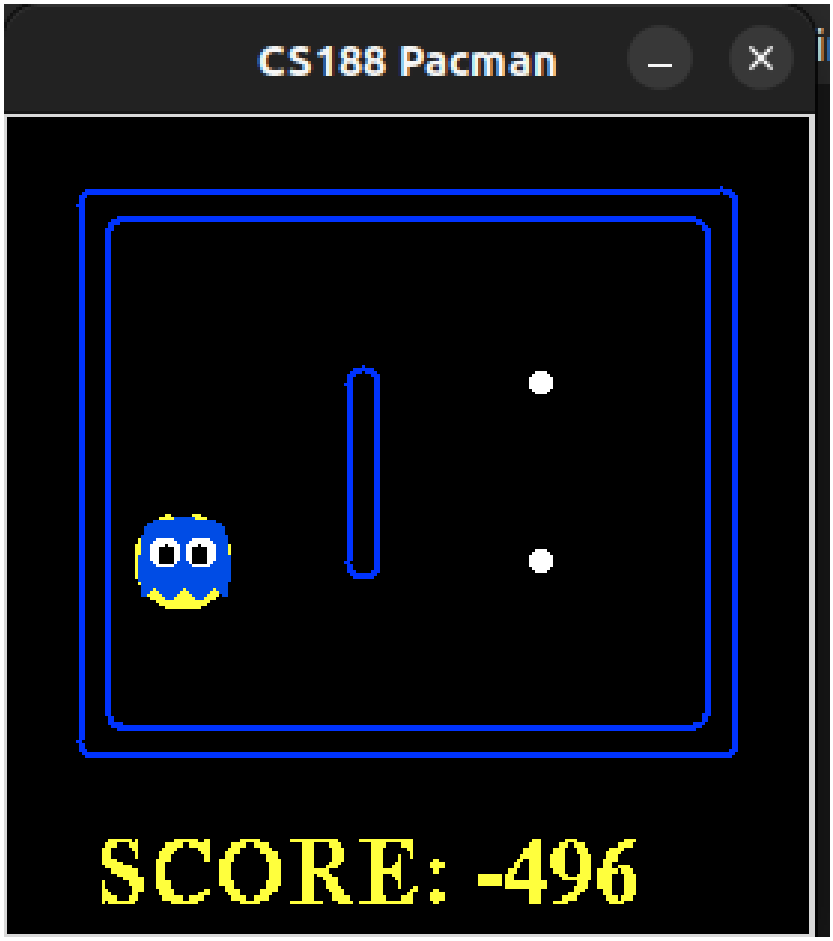

Figura 18: Pacman - mediumgrid

Reinforcement Learning Status:
Completed 2000 out of 2000 training episodes
Average Rewards over all training: -450.99
Average Rewards for last 100 episodes: -439.26
Episode took 0.62 seconds
Training Done (turning off epsilon and alpha)

Average Score: -394.9
Scores: -489.0, -487.0, -500.0, -494.0, -490.0, 496.0, -496.0, -489.0, -506.0, -494.0
Win Rate: 1/10 (0.10)
Record: Loss, Loss, Loss, Loss, Loss, Win, Loss, Loss, Loss, Loss

7 Conclusão

Neste trabalho, analisamos a eficiência dos métodos de aprendizado por reforço, incluindo o Q-learning e o Value Iteration. Em cenários onde temos controle total das recompensas e características do ambiente, como no CliffWorld, ambos os métodos podem encontrar a política ótima de maneira eficaz. O Value Iteration, por sua vez, se

destaca em ambientes onde todas as recompensas e transições são conhecidas, convergindo de forma mais rápida e eficiente, pois calcula diretamente os valores de estado sem precisar explorar o ambiente.

Por outro lado, o Q-learning apresenta vantagens em ambientes onde não temos acesso completo ao modelo, pois aprende por meio de exploração. No entanto, em ambientes não determinísticos com muitos estados e possibilidades, como o PacMan em mapas maiores, o Q-learning enfrenta limitações de escalabilidade, pois não consegue generalizar as ações para todos os estados. Ele precisa explorar extensivamente todas as ações possíveis para descobrir a política ótima, o que se torna ineficiente e custoso em termos de tempo e recursos computacionais.

8 Referências

- Towards Data Science, "Reinforcement Learning: An Easy Introduction to Value Iteration", disponível em: https://towardsdatascience.com/reinforcement-learning-an-eas acessado em: 18 de outubro de 2024.

- Sutton, Richard S. e Barto, Andrew G. "Reinforcement Learning: An Introduction", segunda edição, MIT Press, 2018.