UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS FLORIANÓPOLIS

Trabalho Prático 2 - Q-Learning

Alunos Guilherme dos Santos (202402018)

Professor Prof. Dr. Eric Aislan Antonelo

Florianópolis, 02 de Maio de 2024

Conteúdo

1 Estrutura do Projeto

Os códigos implementados e utilizados no presente trabalho foram desenvolvidos exclusivamente por minha autoria e estão acessíveis por meio do repositório hospedado na plataforma GitHub.

Este repositório contém os códigos e arquivos necessários para a compreensão e reprodução das análises e experimentos conduzidos no âmbito deste projeto. A utilização deste recurso proporciona transparência e replicabilidade dos resultados obtidos, permitindo a validação das conclusões e a verificação da metodologia empregada.

A disposição dos códigos neste repositório segue uma estrutura organizada e documentada, facilitando a navegação e compreensão por parte de outros pesquisadores e interessados no tema abordado. Além disso, eventuais atualizações ou correções podem ser incorporadas de forma transparente e rastreável, contribuindo para a melhoria contínua do trabalho desenvolvido.

2 Questão 1

2.1 Problema

O problema consiste na implementação de um agente de Iteração de Valor para um Processo de Decisão de Markov (MDP). O agente deve realizar um planejamento offline, executando um número especificado de iterações de Iteração de Valor antes de agir de acordo com a política resultante.

2.2 Passo a passo

As necessidades e passos a seguir para completar o código são:

1. Inicialização: Método inicializar, o fator descent, o número de iterações e os valores de estado.

2. Execução da Iteração de Valor: No método ‘runValueIteration‘, implementar o algoritmo de Iteração de Valor, calculando o valor do estado para cada iteração com base nos valores dos estados sucessores.

3. Cálculo do Valor Q: No método ‘computeQValueFromValues‘, calcular o valor Q de uma ação em um estado com base na função de valor armazenada.

4. Cálculo da Ação: No método ‘computeActionFromValues‘, calcular a melhor ação em um estado de acordo com os valores armazenados.

5. Política: Os métodos ‘getPolicy‘ e ‘getAction‘ devem retornar a política no estado, que é a melhor ação de acordo com os valores atuais.

6. Teste: Após a implementação, testar o agente usando o autograder fornecido e executando-o no ambiente do gridworld.

2.3 Resultado

Quanto aos resultados obtidos, podemos dizer que foram interessantes. Embora tenha sido desafiador alcançá-los, com persistência e ajustes adequados, conseguimos obter sucesso. O processo exigiu bastante tentativa e erro.

Executando os códigos para teste do código obtemos um treinamento interessante.

```
EPISODE 10 COMPLETE: RETURN WAS 0.5904900000000002

AVERAGE RETURNS FROM START STATE: 0.5300297289000001
```

Figura 1: Gridworld iteration

Após 100 iterações feitas, o algoritmo aprendeu que não deve cair no buraco e criou uma rota otimizada até o estado final. Observa-se uma melhora na estatística de caminho ao aumentar o tamanho das iterações.

Figura 2: 100 Iterations

3 Questão 2

3.1 Problema

O problema do "Cliff Walking", conforme descrito por Sutton e Barto no Exemplo 6.6, é um cenário clássico de aprendizado por reforço. Neste problema, um agente precisa navegar por um ambiente que se assemelha a um penhasco, onde há um caminho seguro no fundo, mas também um penhasco onde ao cair o agente retorna ao início e é penalizado.

O objetivo do agente é chegar do canto inferior esquerdo do grid 4x12 para o canto inferior direito, minimizando o número de passos e evitando cair no penhasco. O desafio está em equilibrar a exploração (tentativa de encontrar o caminho mais seguro) e a explotação (seguir o caminho conhecido que leva à meta) para otimizar o número de passos necessários.

O ambiente é representado como uma grade, onde cada célula é um estado, e o agente pode se mover em quatro direções: para cima, para baixo, para a esquerda e para a direita. Ao se mover, o agente recebe uma recompensa negativa por cada passo, incentivando-o a alcançar o objetivo o mais rápido possível.

O desafio para o agente é aprender uma política ótima que o guie de forma segura e eficiente do ponto inicial ao objetivo, evitando os perigos do penhasco. Isso envolve aprender a tomar decisões inteligentes sobre quando explorar novos caminhos e quando seguir o caminho mais seguro já conhecido. O método de Q-Aprendizado é uma abordagem comum para resolver esse tipo de problema, onde o agente aprende a associar ações a estados específicos com base em uma função Q que estima a utilidade de cada ação em cada estado.

3.2 Código

```
import numpy as np
import matplotlib.pyplot as plt

# Ambiente
num_states = 48
num_actions = 4
start_state = 0
goal_state = 47Z
cliff_states = np.arange(37, 47)


# Par metros
alpha = 0.1  # Taxa de aprendizado
gamma = 0.95   # Fator de desconto
```

```
epsilon = 0.1  # Taxa de explora  o
num_episodes = 450

actions = {0: -1,  # Esquerda
           1: 1,   # Direita
           2: -12, # Cima
           3: 12}  # Baixo

Q = np.zeros((num_states, num_actions))

# Fun  o para escolher a  o
def choose_action(state):
    if np.random.uniform(0, 1) < epsilon:
        action = np.random.choice(num_actions)
    else:
        action = np.argmax(Q[state, :])
    return action

# Algoritmo Q-Learning
total_rewards = []
for episode in range(num_episodes):
    state = start_state
    episode_reward = 0
    while state != goal_state:
        action = choose_action(state)
        move = actions[action]
        next_state = state + move
        next_state = max(0, min(next_state, num_states - 1))
        if next_state in cliff_states:
            reward = -100
            next_state = start_state
        else:
            reward = -1
        episode_reward += reward
        Q[state, action] = Q[state, action] + alpha * (reward
                                    + gamma * np.max(Q[
                                    next_state, :]) - Q[
                                    state, action])
        state = next_state
    total_rewards.append(episode_reward)

# Plotar recompensa por epis dio
def plot_reward():
    plt.plot(total_rewards)
    plt.xlabel('Epis dios')
    plt.ylabel('Recompensa')
    plt.title('Recompensa por epis dio')
    plt.show()
plot_reward()
```

```
# Plotar pol tica  tima
def plot_optimal_policy():
    optimal_policy = np.argmax(Q, axis=1)
    optimal_policy = np.reshape(optimal_policy, (4, 12))

    fig, ax = plt.subplots()
    ax.matshow(optimal_policy, cmap='viridis')

    for i in range(optimal_policy.shape[0]):
        for j in range(optimal_policy.shape[1]):
            ax.text(j, i, str(optimal_policy[i, j]), va='
                                                center', ha='
                                                center')

    plt.xlabel('Colunas')
    plt.ylabel('Linhas')
    plt.title('Pol tica  tima  com Valores Q')
    plt.show()

plot_optimal_policy()
```

3.3 Informações do código

O código implementa o algoritmo Q-Learning para resolver um problema de caminhada sobre um penhasco. O ambiente tem 48 estados e 4 ações possíveis. Os parâmetros do algoritmo são definidos, incluindo taxa de aprendizado, fator de desconto, taxa de exploração e número de episódios. Uma função escolhe uma ação com base na exploração ou explotação. O algoritmo Q-Learning é executado para atualizar os valores Q. A recompensa por epochs é plotada para verificar o aprendizado. A policy optimization é plotada como uma matriz 4x12.

3.4 Recompensa por Época e Otimização da Política

Nos gráficos abaixo, é possível observar a evolução da recompensa por época e a otimização da política, oferecendo uma visão clara do aprendizado e do caminho escolhido pelo agente. Para explorar diferentes caminhos e obter um aprendizado mais eficaz, é possível ajustar os parâmetros do código conforme necessário.

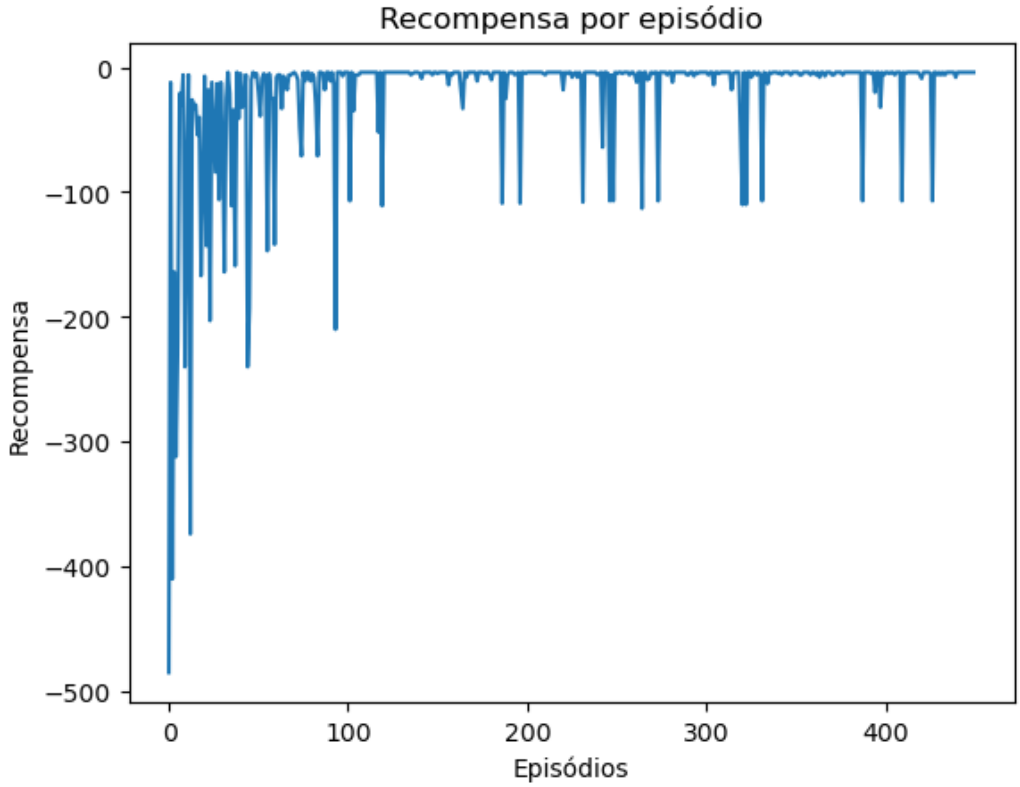

Figura 3: Rewards per epochs

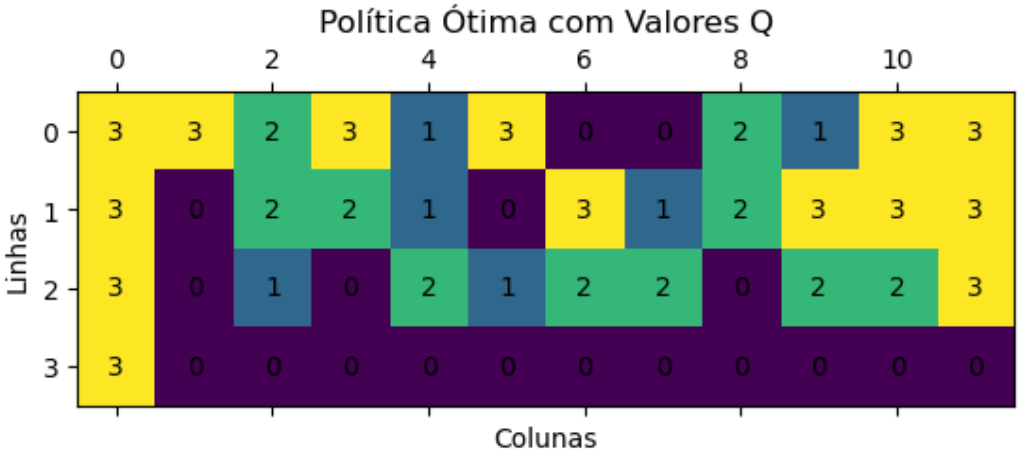

Figura 4: Policy Optimization

4 Questão 3

4.1 Questão 6 - Q-Learning

Esta questão aborda a implementação do Q-Learning com uma função de aproximação. O Q-Learning é um algoritmo de aprendizado por reforço que busca aprender uma política ótima que maximiza a recompensa total esperada. No entanto, em muitos problemas práticos, o espaço de estados e ações pode ser muito grande ou contínuo, tornando inviável aprender e armazenar um valor Q para cada par estado-ação. Para superar isso, o Q-Learning Aproximado usa uma função de aproximação para estimar os valores Q. Em vez de aprender um valor Q para cada par estado-ação, o agente aprende os pesos para os recursos que descrevem os pares estado-ação.

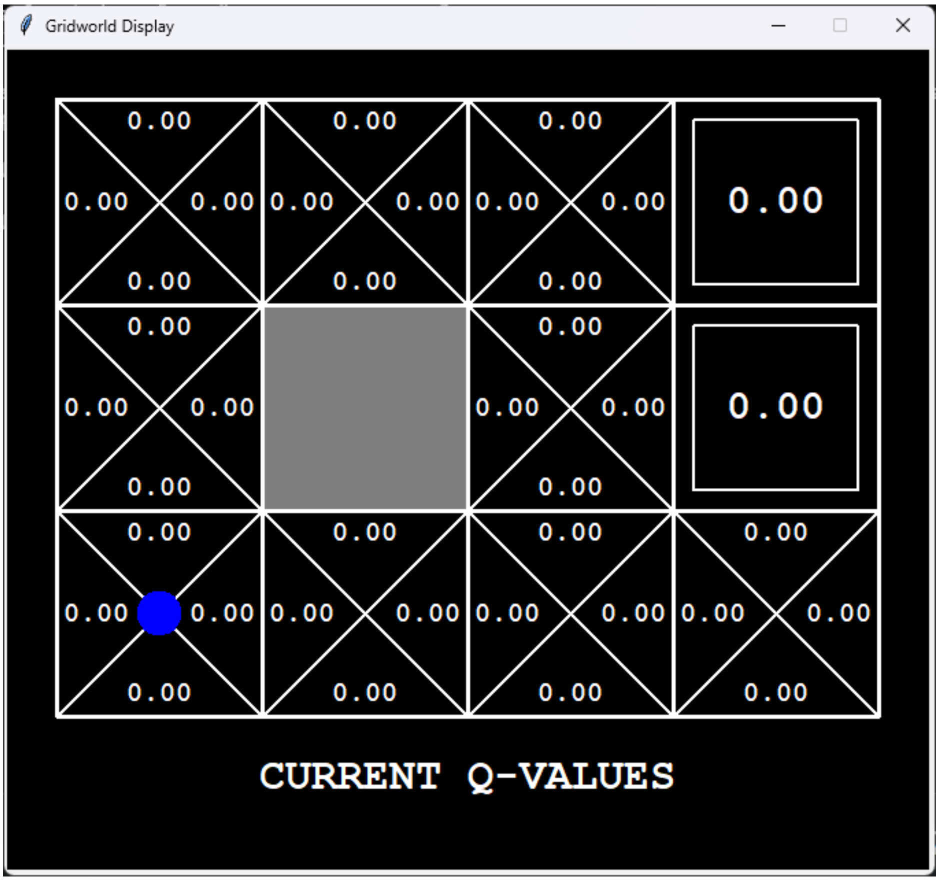

Figura 5: Current Q-values

4.2 Questão 7 - Deep Q-Learning

Esta tarefa aborda a implementação do Deep Q-Learning, uma extensão do Q-Learning que emprega redes neurais como função de aproximação. Aqui, a rede neural recebe a representação do estado como entrada e gera os valores Q para todas as ações possíveis. O objetivo é treinar a rede para se aproximar da função Q ideal. Ao resolver o problema, podemos avaliar o desempenho do algoritmo de Q-Learning na escolha da melhor rota.

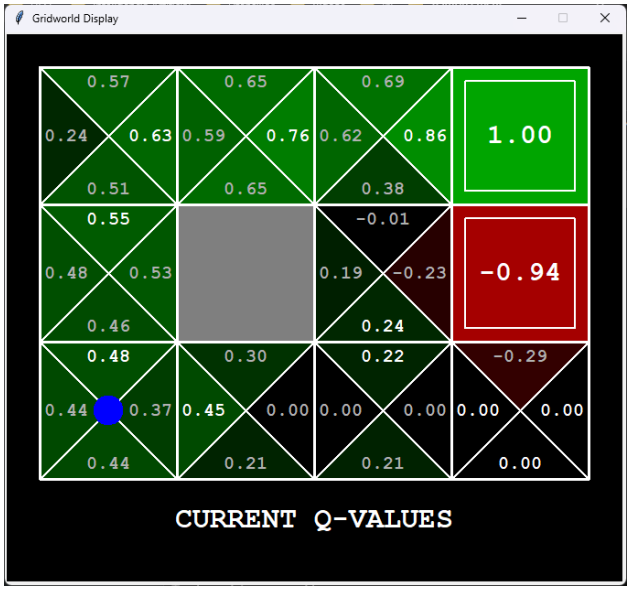

Figura 6: Policy Optimization.png

4.3 Questão 9 - Q-Learning e Pacman

A questão envolve a aplicação do algoritmo Q-Learning no jogo Pacman. O Pacman joga em duas fases: treinamento e teste. Durante o treinamento, o Pacman começa a aprender sobre os valores das posições e ações. Uma vez que o treinamento está completo, ele entra no modo de teste, onde a exploração é desativada para permitir que o Pacman explore a política aprendida.

O problema surge quando o Pacman é treinado em grades maiores, como a ‘mediumGrid‘. Cada configuração do tabuleiro é um estado separado com Q-valores separados. O Pacman não tem como generalizar que correr para um fantasma é ruim para todas as posições. Isso resulta em uma incapacidade de escalar para problemas maiores.

Além disso, o problema pode surgir se os métodos ‘getAction‘ e/ou ‘com-

puteActionFromQValues‘ não considerarem adequadamente as ações não vistas. Como as ações não vistas têm por definição um Q-valor de zero, se todas as ações que foram vistas têm Q-valores negativos, uma ação não vista pode ser ótima.

Portanto, o desafio aqui é implementar um agente Q-Learning que possa aprender efetivamente uma política para o Pacman, mesmo em grades maiores, e que considere adequadamente as ações não vistas ao selecionar ações. Isso envolve a escolha apropriada dos parâmetros de aprendizado e a implementação correta dos métodos de aprendizado por reforço. Além disso, é importante entender que a configuração exata do tabuleiro que o Pacman está enfrentando é o estado do Processo de Decisão de Markov (MDP), e as transições complexas descrevem uma mudança completa nesse estado. As configurações intermediárias do jogo, nas quais o Pacman se moveu, mas os fantasmas não responderam, não são estados do MDP, mas estão incluídas nas transições.

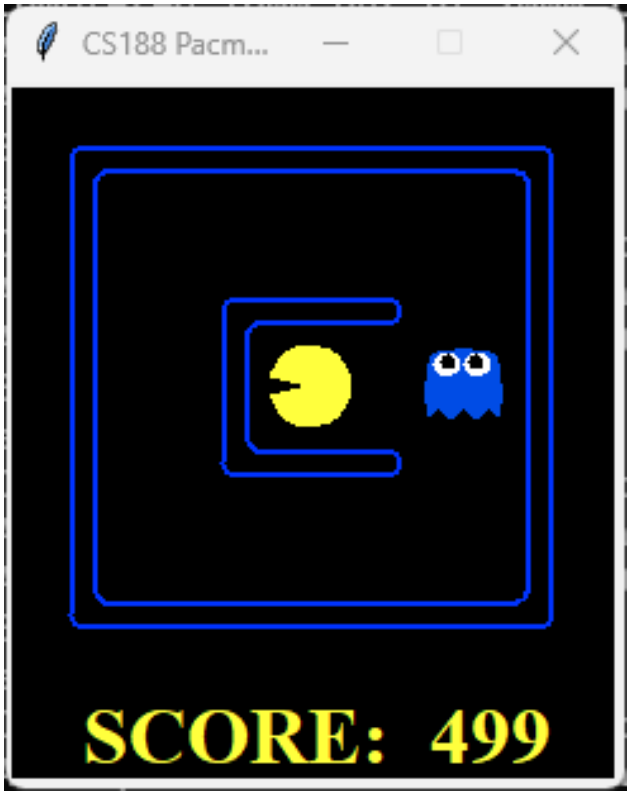

Figura 7: PacMan