

Algoritmo Q-Learning na Aprendizagem por Reforço

Adaptado por: Dr. Arnaldo de Carvalho Junior – Dezembro 2025.

1. INTRODUÇÃO

A aprendizagem de reforço (*reinforcement learning* - RL) é a parte do ecossistema de aprendizado de máquina (*machine learning* - ML), onde o agente aprende interagindo com o ambiente para obter a estratégia ideal a fim de se alcançar as metas. É bem diferente dos algoritmos supervisionados de ML, onde se faz necessário ingerir e processar esses dados. A RL não requer dados. Ao contrário, aprende com o meio ambiente e o sistema de recompensa para tomar melhores decisões [1].

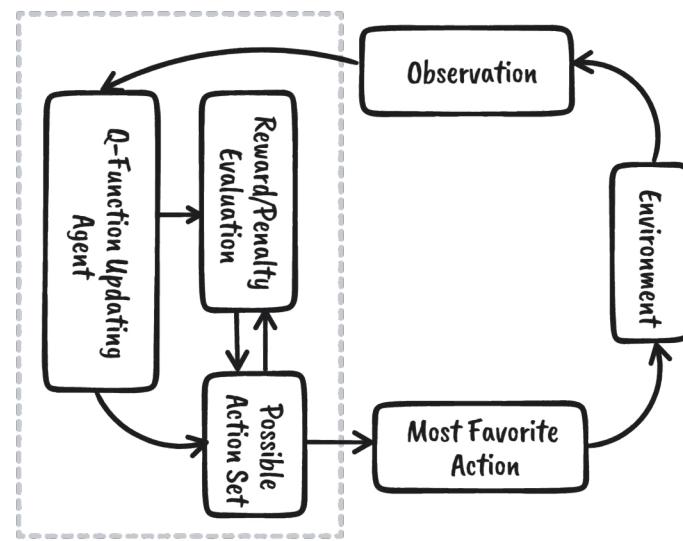


Figura 1 – Fluxo de Aprendizagem por Reforço (RL).

Fonte: Adaptado de [1].

Por exemplo, no *videogame* Mario, se um personagem tomar uma ação aleatória (por exemplo, mover para a esquerda), com base nessa ação, pode receber uma recompensa. Depois de tomar a ação, o agente (Mario) está em um novo estado, e o processo se repete até que o personagem do jogo chegue ao final do palco ou morra. Este episódio se repetirá várias vezes até que Mario aprenda a navegar no meio ambiente, maximizando as recompensas. A Figura 2 apresenta o fluxo do *videogame* Mario [1].

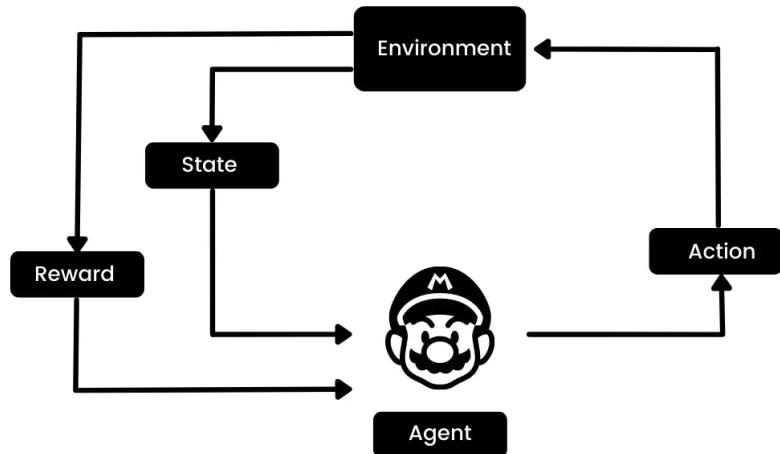


Figura 2 – Fluxo de RL do *videogame* Mario.

Fonte: Adaptado de [1].

Pode-se dividir o aprendizado de reforço em cinco etapas simples:

1. O agente está no estado zero em um ambiente.
2. Serão necessárias uma ação com base em uma estratégia específica.
3. O agente receberá uma recompensa ou punição com base nessa ação.
4. Aprendendo com movimentos anteriores e otimizando a estratégia.
5. O processo será repetido até que uma estratégia ideal seja encontrada.

1.1. Q-Learning

A técnica de aprendizagem por reforço chamada de aprendizagem de qualidade (*quality (Q) learning*) é um tipo de paradigma de aprendizado de máquina em que um algoritmo de aprendizado é treinado não em dados predefinidos, mas sim com base em um sistema de *feedback*. Esses algoritmos são apontados como o futuro do aprendizado de máquinas (Machine Learning - ML), pois eliminam o custo de coleta e limpeza dos dados [2].

O *Q-learning* é um algoritmo de aprendizagem por reforço sem modelo usado para treinar agentes (programas de computador) para tomar decisões ideais interagindo com um ambiente. Ajuda o agente a explorar diferentes ações e aprender quais levam a melhores resultados. O agente usa tentativa e erro para determinar quais ações resultam em recompensas (*good outcomes*) ou penalidades (*bad outcomes*) [2]. A Figura 3 apresenta o fluxograma básico do algoritmo *Q-learning*.

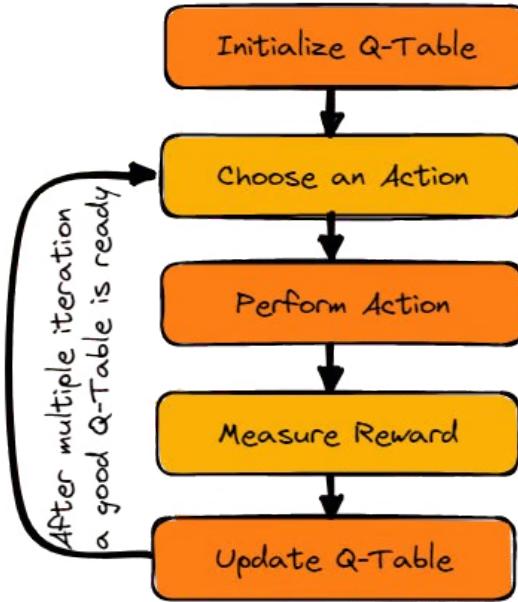


Figura 3 – Algoritmo do Q-Learning.
Fonte: Adaptado de [1].

Com o tempo, melhora sua tomada de decisão atualizando a Tabela Q , que armazena Valores Q que representam as recompensas esperadas para a realização de ações específicas em determinados estados [2].

1.2. Componentes-chave do Q-learning

1. Q-Valores ou Valores de Ação: Os valores Q representam as recompensas esperadas para realizar uma ação em um estado específico. Esses valores são atualizados ao longo do tempo usando a regra de atualização Diferença Temporal (*temporal difference* – TD).
2. Recompensas e Episódios: O agente se move por diferentes estados tomando ações e recebendo recompensas. O processo continua até que o agente atinja um estado terminal, que encerra o episódio.
3. Diferença Temporal ou Atualização TD: O agente atualiza os valores Q usando a fórmula dada pela Equação 1:

$$Q(S,A) \leftarrow Q(S,A) + \alpha(R + \gamma Q(S',A') - Q(S,A)) \quad (1)$$

Onde,

- S é o estado atual.
- A é a ação tomada pelo agente.
- S' é o próximo estado para o qual o agente se move.
- A' é a melhor próxima ação do estado S' .
- R é a recompensa recebida por agir A em estado S .
- γ (gama) é o fator desconto, que equilibra recompensas imediatas com recompensas futuras.
- α (alfa) é a taxa de aprendizagem, determinando quanta informação nova afeta os valores Q antigos.

4. Política ambiciosa (Exploração Negativa vs. Exploração Positiva): A política ε -greedy ajuda o agente a decidir qual ação tomar com base nas estimativas atuais do valor Q :

- Exploração Negativa (*exploitation*): O agente escolhe a ação com o maior valor Q com probabilidade $1 - \varepsilon$. Isso significa que o agente usa seu conhecimento atual para maximizar as recompensas.
- Exploração positiva (*exploration*): Com probabilidade ε , o agente escolhe uma ação aleatória, explorando novas possibilidades para aprender se existem maneiras melhores de obter recompensas. Isso permite que o agente descubra novas estratégias e melhore sua tomada de decisão ao longo do tempo.

1.3. Como funciona o Q-Learning?

Os modelos de Q-learning seguem um processo iterativo, onde diferentes componentes trabalham juntos para treinar o agente [2]:

1. Agente: A entidade que toma decisões e toma ações dentro do ambiente.
2. Estados: As variáveis que definem a posição atual do agente no ambiente.
3. Ações: As operações que o agente realiza quando em um estado específico.
4. Recompensas: O *feedback* que o agente recebe depois de tomar uma ação.
5. Episódios: Uma sequência de ações que termina quando o agente atinge um estado terminal.
6. Valores Q: As recompensas estimadas para cada par estado-ação (*state-action*).

1.4. Etapas do Q-learning:

1. Inicialização: O agente começa com uma tabela Q inicial, onde os valores Q são normalmente inicializados como zero.
2. Exploração: O agente escolhe uma ação com base na política gananciosa (explorando ou explorando).
3. Ação e Atualização: O agente toma a ação, observa o próximo estado e recebe uma recompensa. O valor Q para o par estado-ação é atualizado usando a regra de atualização TD.
4. Iteração: O processo se repete por vários episódios até que o agente aprenda a política ideal.

1.4. Métodos para determinação de valores Q :

1. Diferença Temporal (TD): Diferença Temporal é calculado comparando os valores atuais de estado e ação com os anteriores. Proporciona uma forma de aprender diretamente com a experiência, sem precisar de um modelo de ambiente.
2. Equação de Bellman [3]: é uma fórmula recursiva usada para calcular o valor de um determinado estado e determinar a ação ótima. É fundamental no contexto do Q-learning e se expressa conforme a Equação 2:

$$Q(s,a) = R(s,a) + \gamma \max_a Q(s',a) \quad (2)$$

Onde:

- $Q(s, a)$ é o valor Q para um determinado par estado-ação.
- $R(s, a)$ é a recompensa imediata por agir a em estado s .
- γ é o fator desconto, representando a importância das recompensas futuras.
- $\max_a Q(s',a)$ é o valor Q máximo para o próximo estado s' e todas as ações possíveis.

A Figura 4 apresenta a Equação de Bellman para o Q-learning com as indicações temporais:

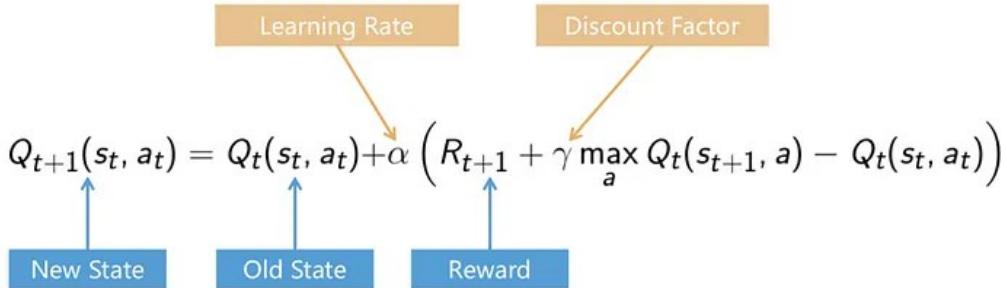


Figura 4 – Equação de Bellman para o Q-learning.
Fonte: Adaptado de [4].

1.5. O que é uma tabela Q ?

O Tabela Q é essencialmente um estrutura memória onde o agente armazena informações sobre quais ações produzem as melhores recompensas em cada estado. É uma tabela de valores Q que representa a compreensão do agente sobre o ambiente. À medida que o agente explora e aprende com suas interações com o ambiente, ele atualiza a tabela Q . A tabela Q ajuda o agente a tomar decisões informadas, mostrando quais ações provavelmente levarão a melhores recompensas.

1.6. Estrutura de uma tabela Q :

- As linhas representam os estados.
- As colunas representam as ações possíveis.
- Cada entrada na tabela corresponde ao valor Q para um par estado-ação.

Com o tempo, à medida que o agente aprende e refina os seus valores Q através da exploração e exploração, a tabela Q evolui para refletir as melhores ações para cada estado, levando a uma tomada de decisão ideal.

2. IMPLEMENTAÇÃO DO Q-LEARNING

Neste capítulo é apresentado um algoritmo básico de *Q-learning* onde o agente aprende a estratégia ideal de seleção de ação para atingir um estado objetivo em um ambiente semelhante a uma grade.

2.1. Definir o Ambiente

Configure os parâmetros do ambiente, incluindo o número de estados e ações, e initialize a tabela Q . Neste cada estado representa uma posição e as ações movem o agente dentro deste ambiente.

```
import numpy as np

n_states = 16
n_actions = 4
goal_state = 15

Q_table = np.zeros((n_states, n_actions))
```

2.2. Definir hiperparâmetros

Defina os parâmetros para o algoritmo *Q-learning* que incluem a taxa de aprendizagem, fator de desconto, probabilidade de exploração e o número de épocas de treinamento.

```
learning_rate = 0.8
discount_factor = 0.95
exploration_prob = 0.2
epochs = 1000
```

2.3. Implemente o Algoritmo *Q-Learning*

Execute o algoritmo *Q-learning* em várias épocas. Cada época envolve a seleção de ações com base em uma estratégia gananciosa epsilon atualizando os valores Q com base nas recompensas recebidas e fazendo a transição para o próximo estado.

```
for epoch in range(epochs):
    current_state = np.random.randint(0, n_states)
    while current_state != goal_state:
        if np.random.rand() < exploration_prob:
            action = np.random.randint(0, n_actions)
        else:
            action = np.argmax(Q_table[current_state])

        next_state = (current_state + 1) % n_states

        reward = 1 if next_state == goal_state else 0

        Q_table[current_state, action] += learning_rate * \
            (reward + discount_factor *
             np.max(Q_table[next_state])) - Q_table[current_state, action])
```

```
current_state = next_state
```

2.4. Produza a Tabela Q aprendida

Após o treinamento, imprima a tabela Q para examinar os valores Q aprendidos que representam as recompensas esperadas para a realização de ações específicas em cada estado. A Figura 5 apresenta a saída da tabela Q aprendida.

```
print("Learned Q-table:")
print(Q_table)
```

```
Learned Q-table:
[[0.48764377 0.39013998 0.48377033 0.48767498]
 [0.51334208 0.51317781 0.51333517 0.51333551]
 [0.54036003 0.54035981 0.54035317 0.54036009]
 [0.56880009 0.56880009 0.56880009 0.56880009]
 [0.59873694 0.59873694 0.59873694 0.59873694]
 [0.63024941 0.63024941 0.63024941 0.6302494 ]
 [0.66342043 0.66342043 0.66342043 0.66342043]
 [0.6983373 0.6983373 0.6983373 0.6983373 ]
 [0.73509189 0.73509189 0.73509189 0.73509189]
 [0.77378094 0.77378094 0.77378094 0.77378094]
 [0.81450625 0.81450625 0.81450625 0.81450625]
 [0.857375 0.857375 0.857375 0.857375 ]
 [0.9025 0.9025 0.9025 0.9025 ]
 [0.95 0.95 0.95 0.95 ]
 [1. 1. 1. 1. ]
 [0. 0. 0. 0. ]]
```

Figura 5 – Tabela Q aprendida.

Fonte: Adaptado de [1].

2.5. Implementação Completa do Algoritmo Q

Segue o código completo. A Figura 6 apresenta a tabela Q contendo os valores Q aprendidos em cada estado.

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

# Parameters
n_states = 16
n_actions = 4
goal_state = 15
```

```

Q_table = np.zeros((n_states, n_actions))

learning_rate = 0.8
discount_factor = 0.95
exploration_prob = 0.2
epochs = 1000

# Q-learning process
for epoch in range(epochs):
    current_state = np.random.randint(0, n_states)

    while current_state != goal_state:

        # Exploration vs. Exploitation ( $\epsilon$ -greedy policy)
        if np.random.rand() < exploration_prob:
            action = np.random.randint(0, n_actions)
        else:
            action = np.argmax(Q_table[current_state])

        # Transition to the next state (circular movement for simplicity)
        next_state = (current_state + 1) % n_states

        # Reward function (1 if goal_state reached, 0 otherwise)
        reward = 1 if next_state == goal_state else 0

        # Q-value update rule (TD update)
        Q_table[current_state, action] += learning_rate * \
            (reward + discount_factor * np.max(Q_table[next_state]) - \
            Q_table[current_state, action])

        current_state = next_state # Update current state

# Visualization of the Q-table in a grid format
q_values_grid = np.max(Q_table, axis=1).reshape((4, 4))

# Plot the grid of Q-values
plt.figure(figsize=(6, 6))
plt.imshow(q_values_grid, cmap='coolwarm', interpolation='nearest')
plt.colorbar(label='Q-value')
plt.title('Learned Q-values for each state')
plt.xticks(np.arange(4), ['0', '1', '2', '3'])
plt.yticks(np.arange(4), ['0', '1', '2', '3'])
plt.gca().invert_yaxis() # To match grid layout
plt.grid(True)

# Annotating the Q-values on the grid
for i in range(4):
    for j in range(4):

```

```

        plt.text(j, i, f'{q_values_grid[i, j]:.2f}', ha='center', va='center',
color='black')

plt.show()

# Print learned Q-table
print("Learned Q-table:")
print(Q_table)

```

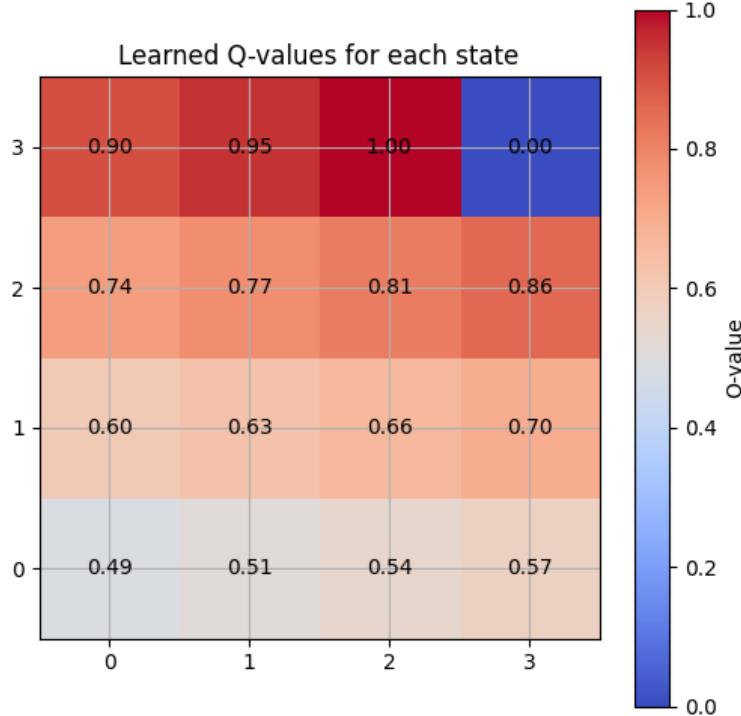


Figura 6 – Tabela Q com valores Q aprendidos em cada estado.
Fonte: Adaptado de [1].

A tabela Q aprendida mostra as recompensas esperadas para cada par estado-ação, com valores Q mais altos próximos ao estado objetivo (state 15), indicando as ações ótimas que levam a atingir a meta. As ações do agente melhoraram gradualmente ao longo do tempo, conforme refletido no aumento dos valores Q entre os estados que levam ao objetivo.

3. APLICAÇÃO

Este capítulo apresenta um algoritmo ML de aprendizagem por reforço *Q-Learning*, implementado em Python [5].

1. Importando as bibliotecas necessárias

```
import numpy as np  
import pylab as pl  
import networkx as nx
```

2. Definir e visualizar o gráfico (*graph*)

```
edges = [(0, 1), (1, 5), (5, 6), (5, 4), (1, 2),  
         (1, 3), (9, 10), (2, 4), (0, 6), (6, 7),  
         (8, 9), (7, 8), (1, 7), (3, 9)]  
  
goal = 10  
G = nx.Graph()  
G.add_edges_from(edges)  
pos = nx.spring_layout(G)  
nx.draw_networkx_nodes(G, pos)  
nx.draw_networkx_edges(G, pos)  
nx.draw_networkx_labels(G, pos)  
pl.show()
```

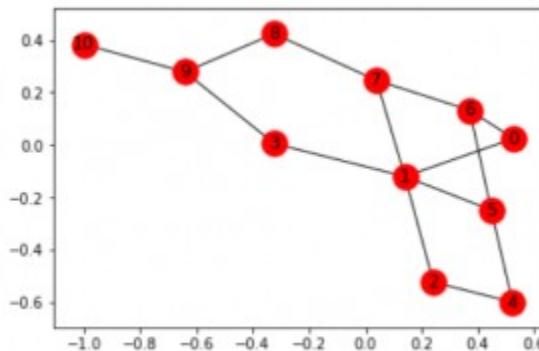


Figura 7 – Visualização do *graph*.
Fonte: Adaptado de [4].

O gráfico da Figura 7 acima pode não ter a mesma aparência na reprodução do código porque a biblioteca networkx [6] em python produz um gráfico aleatório a partir de determinadas arestas.

3. Definindo a recompensa do sistema para o bot

```
MATRIX_SIZE = 11  
M = np.matrix(np.ones(shape =(MATRIX_SIZE, MATRIX_SIZE)))  
M *= -1
```

```

for point in edges:
    print(point)
    if point[1] == goal:
        M[point] = 100
    else:
        M[point] = 0

    if point[0] == goal:
        M[point[::-1]] = 100
    else:
        M[point[::-1]]= 0
        # reverse of point

M[goal, goal]= 100
print(M)
# add goal point round trip

```

```

[[ -1.  0. -1. -1. -1. -1.  0. -1. -1. -1. -1.]
 [ 0. -1.  0.  0. -1.  0. -1.  0. -1. -1. -1.]
 [-1.  0. -1. -1.  0. -1. -1. -1. -1. -1. -1.]
 [-1.  0. -1. -1. -1. -1. -1. -1. -1.  0. -1.]
 [-1. -1.  0. -1. -1.  0. -1. -1. -1. -1. -1.]
 [-1.  0. -1. -1.  0. -1.  0. -1. -1. -1. -1.]
 [-1. -1. -1. -1.  0. -1. -1.  0. -1. -1. -1.]
 [ 0. -1. -1. -1.  0. -1.  0. -1. -1. -1. -1.]
 [-1.  0. -1. -1. -1.  0. -1.  0. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1.  0. -1.  0. -1. -1.]
 [-1. -1. -1.  0. -1. -1. -1. -1.  0. -1. 100.]
 [-1. -1. -1. -1. -1. -1. -1. -1.  0.  0. 100.]]

```

Figura 8 – Imprimindo a matriz M.

Fonte: Adaptado de [4].

4. Definindo algumas funções utilitárias a serem utilizadas no treinamento

```

Q = np.matrix(np.zeros([MATRIX_SIZE, MATRIX_SIZE]))

gamma = 0.75
# learning parameter
initial_state = 1

# Determines the available actions for a given state
def available_actions(state):
    current_state_row = M[state, ]
    available_action = np.where(current_state_row >= 0)[1]
    return available_action

available_action = available_actions(initial_state)

# Chooses one of the available actions at random
def sample_next_action(available_actions_range):
    next_action = int(np.random.choice(available_action, 1))
    return next_action

```

```

action = sample_next_action(available_action)

def update(current_state, action, gamma):

    max_index = np.where(Q[action, ] == np.max(Q[action, ]))[1]
    if max_index.shape[0] > 1:
        max_index = int(np.random.choice(max_index, size = 1))
    else:
        max_index = int(max_index)
    max_value = Q[action, max_index]
    Q[current_state, action] = M[current_state, action] + gamma * max_value
    if (np.max(Q) > 0):
        return(np.sum(Q / np.max(Q)*100))
    else:
        return (0)
# Updates the Q-Matrix according to the path chosen

update(initial_state, action, gamma)

```

5. Treinar e avaliar o *bot* usando a matriz Q

```

scores = []
for i in range(1000):
    current_state = np.random.randint(0, int(Q.shape[0]))
    available_action = available_actions(current_state)
    action = sample_next_action(available_action)
    score = update(current_state, action, gamma)
    scores.append(score)

# print("Trained Q matrix:")
# print(Q / np.max(Q)*100)
# You can uncomment the above two lines to view the trained Q matrix

# Testing
current_state = 0
steps = [current_state]

while current_state != 10:

    next_step_index = np.where(Q[current_state, ] == np.max(Q[current_state, ]))[1]
    if next_step_index.shape[0] > 1:
        next_step_index = int(np.random.choice(next_step_index, size = 1))
    else:
        next_step_index = int(next_step_index)
    steps.append(next_step_index)
    current_state = next_step_index

print("Most efficient path:")
print(steps)

pl.plot(scores)
pl.xlabel('No of iterations')

```

```

pl.ylabel('Reward gained')
pl.show()

```

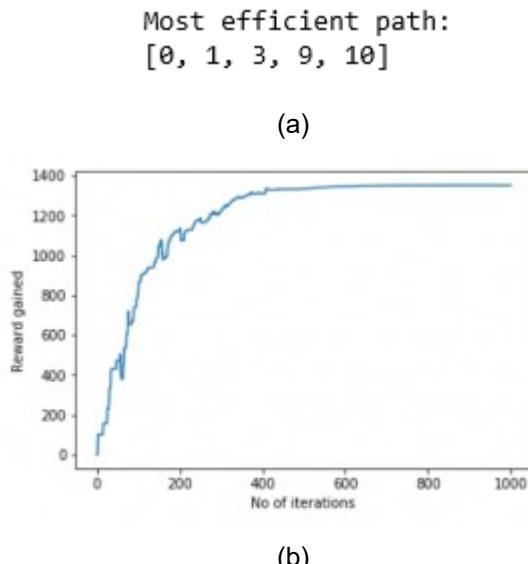


Figura 9 – Caminho mais efetivo (a) e gráfico da pontuação (scores) (b).
Fonte: Adaptado de [4].

O Anexo 1 apresenta o código completo desta aplicação. O código completo pode ser executado via Google Colab pelo link: <https://colab.research.google.com/drive/1DV4faltQYDC28ORqbkypfxGCniLp7GSo>.

O Anexo 2 apresenta o exemplo de código de Q-Learning do lago congelado, adaptado de [7], conforme Figura 10. Execute cada bloco de programação passo a passo para acompanhar o projeto. Este código está disponível no Google Colab pelo link: [clicando aqui](#).

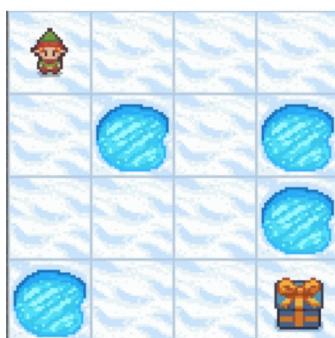


Figura 10 – Exemplo Q-Learning do Lago Congelado.
Fonte: Adaptado de [7].

4. PRÓS E CONTRAS DO ALGORITMO Q-LEARNING DE ML

4.1. Vantagens do Q-learning

- a) Aprendizagem de tentativa e erro: O Q-learning melhora com o tempo, tentando diferentes ações e aprendendo com a experiência.
- b) Auto-Melhoria (*self-improvement*): Erros levam ao aprendizado, ajudando o agente a evitar repeti-los.
- c) Melhor tomada de decisões: Armazena ações bem-sucedidas para evitar más escolhas em situações futuras.
- d) Aprendizagem Autônoma: Aprende sem supervisão externa, puramente através da exploração.

4.2. Desvantagens do Q-learning

- a) Aprendizagem lenta: Requer muitos exemplos, tornando-o demorado para problemas complexos.
- b) Caro em alguns ambientes: Na robótica, as ações de teste podem ser dispendiosas devido a limitações físicas.
- c) Maldição da Dimensionalidade: Grandes espaços de estado e ação tornam a tabela Q muito grande para ser manuseada com eficiência.
- d) Limitado a Ações Discretas: Ele luta com ações contínuas, como ajustar a velocidade, tornando-o menos adequado para aplicações do mundo real que envolvem decisões contínuas.

5. APLICAÇÕES DO Q-LEARNING

As aplicações para Q-learning, um algoritmo de aprendizado por reforço, podem ser encontradas em muitos campos diferentes, tais como [2]:

1. **Jogos Atari:** Os jogos clássicos do Atari 2600 agora podem ser jogados com Q-learning. Em jogos como Space Invaders e Breakout, Deep Q Networks (DQN), uma extensão do Q-learning que faz uso de redes neurais profundas, demonstrou desempenho sobre-humano.

2. **Controle de robô:** Q-learning é usado em robótica para realizar tarefas como navegação e controle de robôs. Com algoritmos de Q-*learning*, os robôs podem aprender a navegar pelos ambientes, evitar obstáculos e maximizar seus movimentos.
3. **Gestão de Tráfego:** Os sistemas autônomos de gerenciamento de tráfego de veículos usam Q-*learning*. Diminui o congestionamento e melhora o fluxo geral do tráfego, otimizando o planejamento das rotas e os horários dos semáforos.
4. **Negociação Algorítmica:** O uso do Q-*learning* para tomar decisões de negociação foi investigado em negociações algorítmicas. Ele torna possível que os agentes automatizados peguem as melhores estratégias de dados de mercado anteriores e se ajustem às mudanças nas condições do mercado.
5. **Planos de Tratamento Personalizados:** Para tornar os planos de tratamento mais exclusivos, o Q-*learning* é usado na área médica. Através do uso de dados de pacientes, os agentes são capazes de recomendar intervenções personalizadas que levam em conta as respostas individuais a vários tratamentos.

REFERÊNCIAS

- [1] AWAN, A. A. An Introduction to Q-Learning: A Tutorial For Beginners, Datacamp, 2022. Disponível em: <<https://www.datacamp.com/tutorial/introduction-q-learning-beginner-tutorial>>. Acesso em Maio 16, 2025.
- [2] GEEKSFORGEEKS, Q-Learning Reinforcement Learning, Geeksforgeeks, 2025. Disponível em: <<https://www.geeksforgeeks.org/q-learning-in-python/>>. Acesso em Maio 15, 2025.
- [3] GEEKSFORGEEKS, Bellman Equation, Geeksforgeeks, 2025. Disponível em: <<https://www.geeksforgeeks.org/bellman-equation/>>. Acesso em Maio 15, 2025.
- [4] JEMIMAH, N. Level up — Understanding Q learning, Medium, 2020. Disponível em:<<https://medium.com/@nancyjemi/level-up-understanding-q-learning-cf739867eb1d>>. Acesso em Maio 15, 2025.
- [5] GEEKSFORGEEKS, ML | Reinforcement Learning Algorithm : Python Implementation using Q-learning, Geeksforgeeks, 2019. Disponível em: <<https://www.geeksforgeeks.org/ml-reinforcement-learning-algorithm-python-implementation-using-q-learning/>>. Acesso em Maio 15, 2025.

- [6] NetworkX, Software for Complex Networks, NetworkX.org, 2024. Disponível em: <<https://networkx.org/documentation/stable/>>. Acesso em Maio 15, 2025.
- [7] SIMONINI, T. Q-Learning with FrozenLake-v1 and Taxi-v3, 2025. Disponível em: <[#scrollTo=KASNViqL4tZn>". Acesso em Maio 16, 2025.](https://colab.research.google.com/drive/1Ur_pYvL_ImgAttMBSZIBRwMNnpzQuY)

ANEXO 1

```
# Fonte: https://www.geeksforgeeks.org/ml-reinforcement-learning-algorithm-python-implementation-using-q-learning/

import numpy as np
import pylab as pl
import networkx as nx
edges = [(0, 1), (1, 5), (5, 6), (5, 4), (1, 2),
          (1, 3), (9, 10), (2, 4), (0, 6), (6, 7),
          (8, 9), (7, 8), (1, 7), (3, 9)]

goal = 10
G = nx.Graph()
G.add_edges_from(edges)
pos = nx.spring_layout(G)
nx.draw_networkx_nodes(G, pos)
nx.draw_networkx_edges(G, pos)
nx.draw_networkx_labels(G, pos)
pl.show()

MATRIX_SIZE = 11
M = np.matrix(np.ones(shape =(MATRIX_SIZE, MATRIX_SIZE)))
M *= -1

for point in edges:
    print(point)
    if point[1] == goal:
        M[point] = 100
    else:
        M[point] = 0

    if point[0] == goal:
        M[point[::-1]] = 100
    else:
        M[point[::-1]]= 0
        # reverse of point

M[goal, goal]= 100
print(M)
# add goal point round trip

Q = np.matrix(np.zeros([MATRIX_SIZE, MATRIX_SIZE]))

gamma = 0.75
# learning parameter
initial_state = 1

# Determines the available actions for a given state
def available_actions(state):
    current_state_row = M[state, ]
    available_action = np.where(current_state_row >= 0)[1]
    return available_action

available_action = available_actions(initial_state)
```

```

# Chooses one of the available actions at random
def sample_next_action(available_actions_range):
    next_action = int(np.random.choice(available_action, 1))
    return next_action

action = sample_next_action(available_action)

def update(current_state, action, gamma):

    max_index = np.where(Q[action, ] == np.max(Q[action, ]))[1]
    if max_index.shape[0] > 1:
        max_index = int(np.random.choice(max_index, size = 1))
    else:
        max_index = int(max_index)
    max_value = Q[action, max_index]
    Q[current_state, action] = M[current_state, action] + gamma * max_value
    if (np.max(Q) > 0):
        return(np.sum(Q / np.max(Q)*100))
    else:
        return (0)
# Updates the Q-Matrix according to the path chosen

update(initial_state, action, gamma)

scores = []
for i in range(1000):
    current_state = np.random.randint(0, int(Q.shape[0]))
    available_action = available_actions(current_state)
    action = sample_next_action(available_action)
    score = update(current_state, action, gamma)
    scores.append(score)

print("Trained Q matrix:")
print(Q / np.max(Q)*100)
# You can uncomment the above two lines to view the trained Q matrix

# Testing
current_state = 0
steps = [current_state]

while current_state != 10:

    next_step_index = np.where(Q[current_state, ] == np.max(Q[current_state, ]))[1]
    if next_step_index.shape[0] > 1:
        next_step_index = int(np.random.choice(next_step_index, size = 1))
    else:
        next_step_index = int(next_step_index)
    steps.append(next_step_index)
    current_state = next_step_index

print("Most efficient path:")
print(steps)

pl.plot(scores)

```

```
pl.xlabel('No of iterations')
pl.ylabel('Reward gained')
pl.show()
```

ANEXO 2

```
# Referência:  
https://colab.research.google.com/drive/1IdsdHZ9Q8pAhmPfX4hHed150IG4YtnaX?usp=sharing  
# Tutorial: https://www.datacamp.com/tutorial/introduction-q-learning-beginner-tutorial  
# Adaptado por: Dr. Arnaldo de Carvalho Junior - Maio 2015  
  
# Configurando o Display Virtual  
%%capture  
!pip install pyglet==1.5.1  
!apt install python-opengl  
!apt install ffmpeg  
!apt install xvfb  
!pip3 install pyvirtualdisplay  
  
# Virtual display  
from pyvirtualdisplay import Display  
  
virtual_display = Display(visible=0, size=(1400, 900))  
virtual_display.start()  
  
# Instalando Dependências  
%%capture  
!pip install gym==0.24  
!pip install pygame  
!pip install numpy  
  
!pip install imageio imageio_ffmpeg  
  
# Importando os Pacotes (bibliotecas)  
import numpy as np  
import gym  
import random  
import imageio  
from tqdm.notebook import trange  
  
# Lago Congelado  
# Cria o ambiente Lago Congelado (FrozenLake-v1) usando um mapa 4x4 e versão não escorregadia (non-slippery)  
env = gym.make("FrozenLake-v1", map_name="4x4", is_slippery=False)  
  
# Compreendendo o ambiente do Lago Congelado  
print("____ESPAÇO DE OBSERVAÇÃO____ \n")  
print("Espaço de Observação", env.observation_space)  
print("Amostra de Observação", env.observation_space.sample()) # display a random observation  
  
# Imprimindo o espaço de ação  
print("\n ____ESPAÇO DE AÇÃO____ \n")
```

```

print("Forma do Espaço de Ação", env.action_space.n)
print("Amostra do Espaço de Ação", env.action_space.sample())

# Criando e Inicializando a tabela-Q
state_space = env.observation_space.n
print("Existem ", state_space, " possíveis estados")

action_space = env.action_space.n
print("Existem ", action_space, " possíveis ações")

# Criar tabela-Q de tamanho (State_space, Action_space) e inicializar cada valores em
# 0 usando np.zeros
def initialize_q_table(state_space, action_space):
    Qtable = np.zeros((state_space, action_space))
    return Qtable

# Definir política y ganancioso (epsilon_greedy)
def epsilon_greedy_policy(Qtable, state, epsilon):
    # Número gerado aleatoriamente entre 0 e 1
    random_int = random.uniform(0,1)
    # Se random_int > maior que y => exploração
    if random_int > epsilon:
        # Tomar a ação com o maior valor dado um estado
        # np.max poder ser útil aqui
        action = np.argmax(Qtable[state])
    # else => exploração
    else:
        action = env.action_space.sample()
    return action

# Definir a política gananciosa (greedy_policy)
# Recuperação corrigida para a função de política gananciosa
def greedy_policy(Qtable, state):
    # Exploração: Tome a ação com o estado mais alto, valor de ação
    action = np.argmax(Qtable[state])
    return action

# Treinando Parâmetros
n_training_episodes = 10000 # Total de episódios de treinamento
learning_rate = 0.7 # Taxa de aprendizagem

# Avaliação de Parâmetros
n_eval_episodes = 100 # Número total de episódios de teste

# Parâmetros do Ambiente
env_id = "FrozenLake-v1" # Nome do ambiente
max_steps = 99 # Máximos passos por episódio
gamma = 0.95 # Taxa de Desconto

```

```

eval_seed = []                      # A semente de avaliação do ambiente

# Parâmetros de Exploração
max_epsilon = 1.0                  # Probabilidade de exploração ao iniciar
min_epsilon = 0.05                  # Probabilidade mínima de exploração
decay_rate = 0.0005                # Taxa de decaimento exponencial para a probabilidade de
exploração

# Treinando o modelo
def train(n_training_episodes, min_epsilon, max_epsilon, decay_rate, env, max_steps,
Qtable):
    for episode in range(n_training_episodes):
        # Reduza o epsilon (porque se precisa de cada vez menos exploração)
        epsilon = min_epsilon + (max_epsilon - min_epsilon)*np.exp(-decay_rate*episode)
        # Reinicialize o ambiente
        state = env.reset()
        step = 0
        done = False

        # Repetir
        for step in range(max_steps):
            # Escolha a ação At usando a política Epsilon Gananciosa
            action = epsilon_greedy_policy(Qtable, state, epsilon)
            # Tome a ação At e observe Rt+1 e St+1
            # Tome a ação (a) e observe a saída estado (s') e recompensa (r)
            new_state, reward, done, info = env.step(action)

            # Atualize Q(s,a):= Q(s,a) + lr [R(s,a) + gamma * max Q(s',a') - Q(s,a)]
            Qtable[state][action] = Qtable[state][action] + learning_rate * (reward + gamma *
            np.max(Qtable[new_state]) - Qtable[state][action])

            # Se feito, finalize o episódio
            if done:
                break

            # Nossa estado é o novo estado
            state = new_state
        return Qtable

Qtable_frozenlake = train(n_training_episodes, min_epsilon, max_epsilon, decay_rate,
env, max_steps, Qtable_frozenlake)

Qtable_frozenlake

# Avaliação do Modelo
def evaluate_agent(env, max_steps, n_eval_episodes, Q, seed):

```

```

episode_rewards = []
for episode in range(n_eval_episodes):
    if seed:
        state = env.reset(seed=seed[episode])
    else:
        state = env.reset()
    step = 0
    done = False
    total_rewards_ep = 0

    for step in range(max_steps):
        # Tome a ação (índice) que têm a recompensa máxima
        action = np.argmax(Q[state][:])
        new_state, reward, done, info = env.step(action)
        total_rewards_ep += reward

        if done:
            break
        state = new_state
    episode_rewards.append(total_rewards_ep)
mean_reward = np.mean(episode_rewards)
std_reward = np.std(episode_rewards)

return mean_reward, std_reward

# Avaliando o Agente
mean_reward, std_reward = evaluate_agent(env, max_steps, n_eval_episodes,
Qtable_frozenlake, eval_seed)
print(f"Mean_reward={mean_reward:.2f} +/- {std_reward:.2f}")

# Visualizando os Resultados
def record_video(env, Qtable, out_directory, fps=1):
    images = []
    done = False
    state = env.reset(seed=random.randint(0,500))
    img = env.render(mode='rgb_array')
    images.append(img)
    while not done:
        # Tome a ação (índice) que têm a recompensa futura máxima esperada, dado aquele
        estado
        action = np.argmax(Qtable[state][:])
        state, reward, done, info = env.step(action) # Colocar diretamente next_state =
        estado para gravar lógica
        img = env.render(mode='rgb_array')
        images.append(img)

```

```
imageio.mimsave(out_directory, [np.array(img) for i, img in enumerate(images)],  
fps=fps)  
  
# Salvando arquivo animado omo gif com 1 quadro por segundo  
video_path="/content/replay.gif"  
video_fps=1  
  
record_video(env, Qtable_frozenlake, video_path, video_fps)  
  
from IPython.display import Image  
Image('./replay.gif')
```