

Avaliação de Algoritmos de Aprendizado por Reforço para o Jogo BlackJack

Lucas Silva Lima, Lucca Rangel Haddad e Antonio Gustavo Silveira Dantas

Instituto Tecnológico de Aeronáutica, São José dos Campos/SP

16 de julho de 2024

Resumo—Este trabalho avalia diferentes algoritmos de aprendizado por reforço na resolução do jogo de cartas *BlackJack*. Os algoritmos analisados incluem *Monte Carlo*, *SARSA*, *Q-Learning* e *Deep Q-Learning*. A avaliação foi realizada utilizando o ambiente *gym* da OpenAI®, com o objetivo de fornecer uma visão detalhada sobre o desempenho de cada algoritmo em termos de taxa de vitórias. Os resultados mostraram que o agente baseado em *Deep Q-Learning* apresentou a maior taxa de vitórias, enquanto os demais se destacaram pelo reduzido tempo de treinamento frente ao agente mais eficaz. Este estudo busca oferecer *insights* sobre a eficácia de cada abordagem e sua aplicabilidade em um cenário de tomada de decisão sequencial.

Index Terms—Aprendizado por Reforço, BlackJack, Monte Carlo, SARSA, Q-Learning, Deep Q-Learning, OpenAI® Gym.

I. INTRODUÇÃO

O Aprendizado por reforço (*Reinforcement Learning - RL*) tem demonstrado ser uma técnica eficaz para resolução de problemas complexos onde a tomada de decisão sequencial é essencial. O *RL* é uma área do aprendizado de máquina que se concentra em como agentes devem tomar ações em um ambiente específico para maximizar uma noção de recompensa cumulativa. Entre os diversos desafios que podem ser abordados com *RL*, o jogo de cartas *BlackJack* se destaca por sua simplicidade estrutural e por oferecer uma plataforma acessível para testar e comparar diferentes algoritmos de aprendizado por reforço.

A. Descrição do Jogo

BlackJack é um jogo de cartas popularmente jogado em cassinos ao redor do mundo. O objetivo principal é alcançar a soma de cartas o mais próximo possível de 21, sem ultrapassá-lo. O jogo é jogado contra o *dealer* (banca) e envolve as seguintes regras básicas:

- Cada jogador recebe duas cartas no início, assim como o *dealer*, com uma das cartas do *dealer* virada para cima.
- As cartas de 2 a 10 têm valor nominal, as figuras (valete, dama e rei) valem 10, e o Ás pode valer 1 ou 11.
- Os jogadores podem optar por “pedir” (*hit*) uma carta adicional ou “parar” (*stand*) a qualquer momento.
- Se a soma das cartas de um jogador ultrapassar 21, ele “estoura” (*bust*) e perde a rodada (episódio).
- O *dealer* deve continuar “pedindo” cartas até que a soma das suas cartas seja pelo menos 17.
- Ganha quem tiver a soma de cartas mais próxima de 21, sem ultrapassá-lo. Em caso de empate, o *dealer* vence.



Figura 1: Jogo de cartas *BlackJack*.

Neste estudo, utilizamos o ambiente de simulação de *BlackJack* fornecido pelo *gym* da OpenAI® para treinar e avaliar os algoritmos de *RL*.

II. REVISÃO TEÓRICA

A. Dinâmica de Markov

Para modelar o problema do *BlackJack* como um Processo de Decisão de Markov (*Markov Decision Process - MDP*), definimos os seguintes componentes:

- **Estados (*S*):** Cada estado é definido pela soma das cartas do jogador, a carta visível do *dealer* e a presença de um Ás utilizável. Um estado pode ser representado como (s, d, a) , onde s é a soma das cartas do jogador, d é o valor da carta visível do *dealer* e a indica a presença de um Ás utilizável (1 se presente, 0 se ausente).
- **Ações (*A*):** As ações possíveis são “pedir” (*hit*) ou “parar” (*stand*).
- **Recompensas (*R*):** A recompensa é recebida ao final de cada episódio: +1 se o jogador vencer e -1 se o jogador empatar ou perder.
- **Política (π):** A política define a probabilidade de tomar uma ação em cada estado. Uma política pode ser determinística ou estocástica.
- **Transições:** As transições entre estados são determinadas pelas regras do jogo e pela política seguida.
- **Ambiente:** O ambiente consiste nas regras do jogo de *BlackJack* e nas cartas distribuídas durante o jogo.

A função ação-valor $Q(s, a)$ em um *MDP* é definida como a expectativa do retorno futuro a partir do estado s e ação a , seguindo uma política π [1]:

$$Q_\pi(s, a) = E_\pi[G_t \mid S_t = s, A_t = a] \quad (1)$$

Onde G_t é o retorno acumulado a partir do tempo t , dado por:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (2)$$

A função ação-valor pode ser estimada iterativamente usando a equação de Bellman:

$$Q_\pi(s, a) = \sum_{s' \in S} p(s'|s, a) [r(s, a, s') + \gamma \sum_{a' \in A} \pi(a'|s') Q_\pi(s', a')] \quad (3)$$

B. Monte Carlo

A predição de *Monte Carlo* (*MC*) envolve resolver problemas estatísticos complexos por meio de amostragem, realizando múltiplas simulações e avaliando a média dos resultados. Este algoritmo aprende diretamente de episódios de experiência, é livre de modelo e utiliza a média dos retornos para atualizar os valores da função ação-valor.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{T-t-1} R_T \quad (4)$$

Onde G_t é o retorno total acumulado a partir do tempo t , e γ é o fator de desconto.

$$Q_\pi(s, a) = E_\pi[G_t \mid S_t = s, A_t = a] \quad (5)$$

A função de valor $G_\pi(s, a)$ é a média dos retornos recebidos a partir do par ação-valor (s, a) , seguindo a política π .

C. Sarsa

O *SARSA* (*State-Action-Reward-State-Action*) é um algoritmo de RL *on-policy*, que atualiza a função de valor Q com base na ação tomada e na recompensa recebida, utilizando a política seguida pelo agente.

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \quad (6)$$

Onde α é a taxa de aprendizado e γ é o fator de desconto.

D. *Q*-learning

O *Q*-learning é um algoritmo *off-policy* que busca encontrar a política ótima independentemente da política seguida pelo agente, atualizando a função de valor Q com base na ação que maximiza a função de valor para o próximo estado.

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \quad (7)$$

Aqui, a atualização é baseada na ação que maximiza a função de valor Q para o próximo estado, em vez da ação realmente tomada.

E. Deep *Q*-learning

O Deep *Q*-learning (DQN) estende o *Q*-learning ao utilizar redes neurais profundas para aproximar a função de valor Q , sendo útil quando o espaço de estados é muito grande para ser representado de forma tabular. A atualização da rede neural é feita minimizando o erro quadrático entre a predição atual e o alvo Q .

$$L(\theta) = E \left[\left(R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a'; \theta^-) - Q(S_t, A_t; \theta) \right)^2 \right] \quad (8)$$

Onde θ são os parâmetros da rede neural e θ^- são os parâmetros da rede alvo, os quais são atualizados periodicamente para estabilizar o treinamento.

III. DESENVOLVIMENTO

A. Implementação dos Algoritmos

Para a implementação dos algoritmos de RL, utilizamos a biblioteca *gym* da OpenAI® para criar o ambiente de simulação do jogo *BlackJack*. Adicionalmente, utilizamos a biblioteca *numpy* para operações matemáticas e, no caso do Deep *Q-Learning*, a biblioteca *tensorflow* para construção e treinamento da rede neural.

Implementou-se os algoritmos *Monte Carlo*, *SARSA*, *Q-Learning* e *Deep Q-Learning*. Os códigos completos das implementações estão disponíveis no [Apêndice](#).

B. Quasi-ótimo Monte Carlo

Para melhorar o desempenho da implementação com o algoritmo *Monte Carlo*, desenvolvemos um novo agente que foi treinado por um número significativamente maior de episódios (5 milhões). Dado que o algoritmo *Monte Carlo* garante a convergência quando o número de episódios tende ao infinito, assumimos que, para fins práticos, o desempenho desse agente é ótimo. Assim, denominamos essa abordagem de Quasi-ótimo *Monte Carlo*. Esta implementação serve como um comparativo com as demais, que foram treinadas por apenas 5 mil episódios, proporcionando uma referência de desempenho ideal.

C. Detalhes da Implementação

Os algoritmos foram implementados com a linguagem de programação *Python*. A seguir, descrevemos brevemente os passos principais e arbitrações de cada implementação:

1) Monte Carlo:

- Inicialização dos valores $Q(s, a)$ e contadores $N(s, a)$.
- Execução da política ϵ -greedy para gerar episódios.
- Atualização dos valores $Q(s, a)$ com a média incremental dos retornos.
- Atualização de ϵ com dacaimento exponencial, tendendo à 0.01 ao término do aprendizado.
- $\gamma = 1$.

2) SARSA:

- Inicialização dos valores $Q(s, a)$.
- Execução da política ϵ -greedy para selecionar ações.
- Atualização dos valores $Q(s, a)$ baseando-se na ação realmente tomada.
- Atualização de ϵ com dacaimento exponencial, tendendo à 0.01 ao término do aprendizado.
- $\gamma = 1$.

3) Q-Learning:

- Inicialização dos valores $Q(s, a)$.
- Execução da política ϵ -greedy para selecionar ações.
- Atualização dos valores $Q(s, a)$ baseando-se na ação que maximiza a função de valor para o próximo estado.
- Atualização de ϵ com dacaimento exponencial, tendendo à 0.01 ao término do aprendizado.
- $\gamma = 1$.

4) Deep Q-Learning:

- Utilização de uma rede neural para aproximar a função de valor $Q(s, a)$.
- Treinamento da rede neural utilizando mini-lotes de experiências armazenadas em um buffer de repetição.
- Atualização dos pesos da rede para minimizar a diferença entre a predição atual e o alvo Q .
- Atualização de ϵ com dacaimento exponencial, tendendo à 0.01 ao término do aprendizado.
- $\gamma = 1$.
- tamanho do mini-lote = 32
- tamanho do *buffer replay*: 4.096

Os códigos dos métodos de aprendizado de todas as implementações estão apresentados no [Apêndice](#) (Seções A.1 a A.4).

IV. RESULTADOS

Cada algoritmo foi avaliado em termos de taxa de vitórias após 5 mil episódios (partidas).

A evolução da taxa de vitórias e as políticas aprendidas pelos agentes são apresentadas nas Figuras 2 a 11. A Figura 12 mostra a comparação da evolução das taxas de vitórias entre as diferentes implementações. A Tabela I resume as taxas de vitórias finais para cada agente, enquanto a Tabela II apresenta os tempos dispendidos para treinamento e inferência.

Tabela I: TAXA DE VITÓRIAS PARA CADA IMPLEMENTAÇÃO

Algoritmo	Taxa de Vitória
Deep Q-Learning	42.4%
Quasi-ótimo Monte Carlo	42.2%
Monte Carlo	40.3%
Q-Learning	37.8%
SARSA	37.0%

V. DISCUSSÃO E CONCLUSÃO

Os resultados deste estudo confirmam que os algoritmos de RL, incluindo *Monte Carlo*, SARSA, *Q-Learning* e *Deep Q-Learning*, podem melhorar significativamente o desempenho em jogos de *BlackJack* em comparação a uma política aleatória. Com somente 5 mil episódios, a implementação com

Tabela II: TEMPO DE PROCESSAMENTO PARA CADA IMPLEMENTAÇÃO

Algoritmo	T. de treino (min)	T. de Inferência (s)
Deep Q-Learning	720	0.3
Quasi-ótimo Monte Carlo	20	0.2
Monte Carlo	4	0.2
Q-Learning	4	0.2
SARSA	4	0.2

Obs.: O processamento foi realizado em um computador com a seguinte especificação: SO Microsoft® Windows® 11 Pro for Workstations SMP x86_64, CPU Intel® Xeon® W-2123 CPU 3.60 GHz 8 Cores, SDRAM 32 GB.

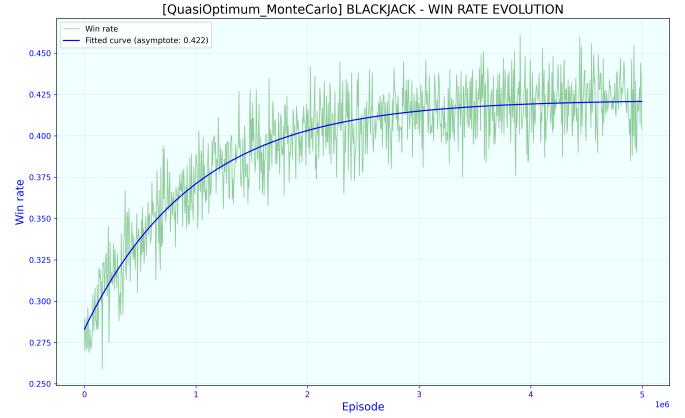


Figura 2: Evolução da taxa de vitórias com o algoritmo *Quasi-ótimo Monte Carlo*.

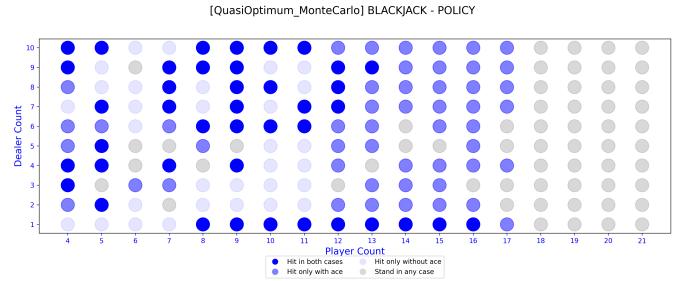


Figura 3: Política do algoritmo *Quasi-ótimo Monte Carlo*.

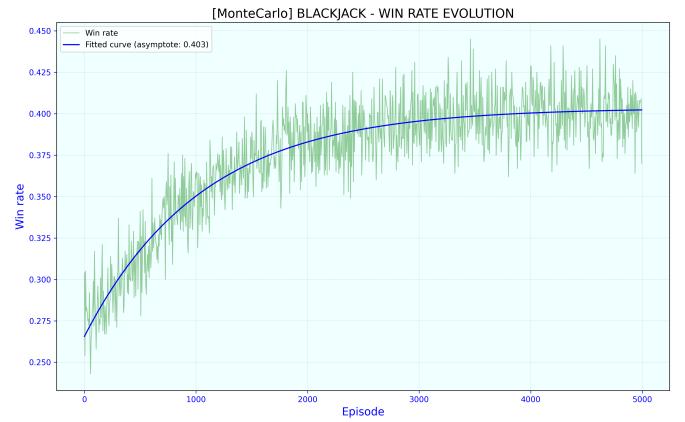


Figura 4: Evolução da taxa de vitórias com o algoritmo *Monte Carlo*.

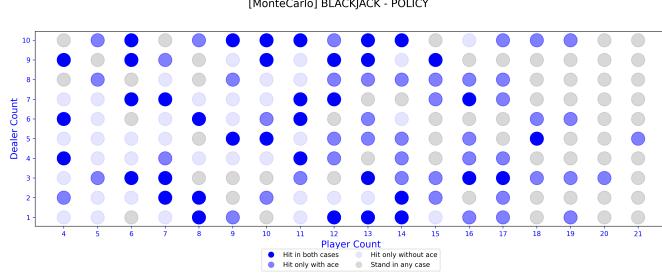


Figura 5: Política do algoritmo *Monte Carlo*.

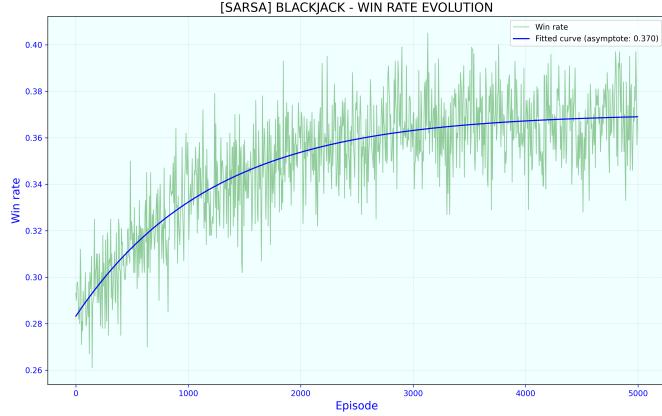


Figura 6: Evolução da taxa de vitórias com o algoritmo SARSA.

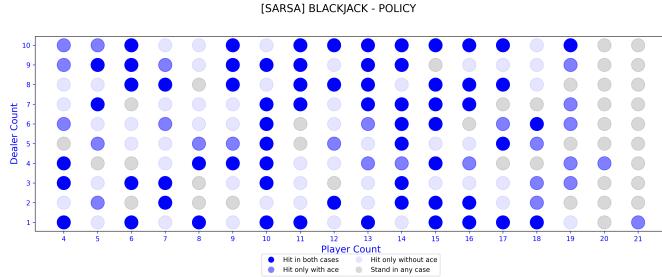


Figura 7: Política do algoritmo SARSA.

Deep Q-Learning se destacou como o algoritmo aprendizado mais eficaz, apresentando a maior taxa de vitórias.

Um fator limitante importante no ambiente utilizado para este estudo é a representação dos estados, onde só temos acesso à soma das cartas do jogador e não às cartas individuais que compõem essa soma. Isso impede a implementação de técnicas de contagem de cartas, frequentemente usadas por jogadores humanos profissionais para melhorar seu desempenho nesse jogo. Jogadores humanos experientes conseguem taxas de vitórias de aproximadamente 51.5 %, significativamente superiores àquelas obtidas pelos algoritmos aqui implementados, devido à capacidade de obter observações mais detalhadas do estado do ambiente (cartas que compõem a contagem).

Para problemas mais desafiadores, como jogos com maior complexidade estratégica ou ambientes com estados contínuos e de alta dimensionalidade, é necessário explorar abordagens

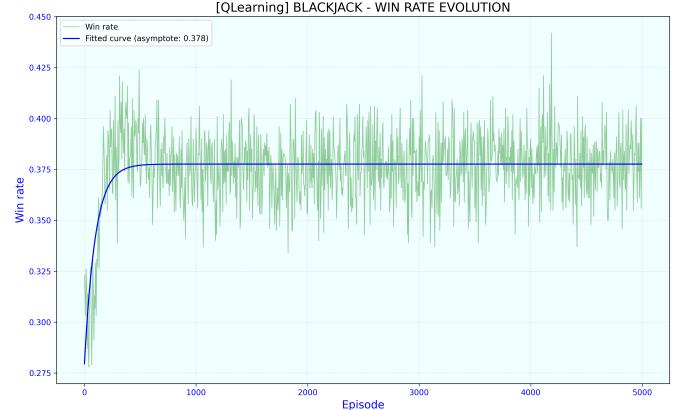


Figura 8: Evolução da taxa de vitórias com o algoritmo *Q-Learning*.

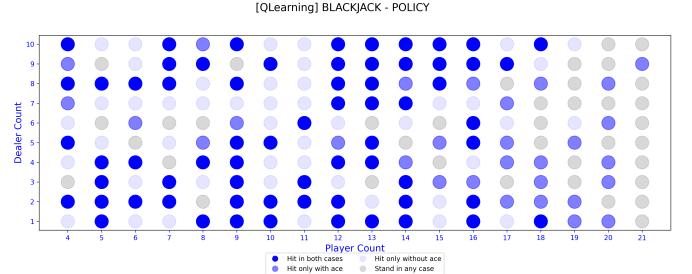


Figura 9: Política do algoritmo *Q-Learning*.

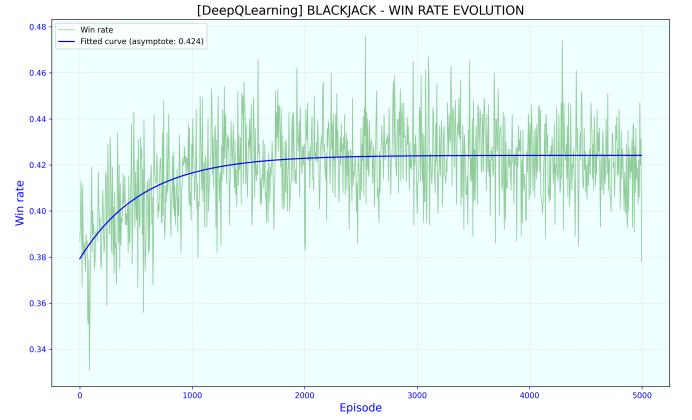


Figura 10: Evolução da taxa de vitórias com o algoritmo *Deep Q-Learning*.

mais avançadas. Algoritmos como *Proximal Policy Optimization* (PPO), *Trust Region Policy Optimization* (TRPO) e *Actor-Critic* podem oferecer melhorias substanciais em desempenho e eficiência de aprendizado.

Além disso, a combinação de técnicas, como aprendizado por reforço profundo (*Deep Reinforcement Learning - DRL*) com redes neurais convolucionais (CNNs) para processamento de entrada visual, pode ser vantajosa para problemas que envolvem reconhecimento de padrões ou processamento de imagens, como jogos de tabuleiro complexos (por exemplo,

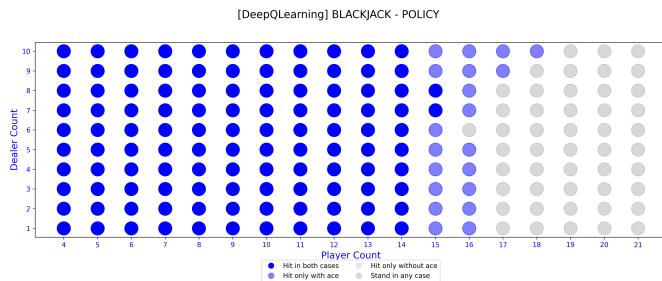


Figura 11: Política do algoritmo *Deep Q-Learning*

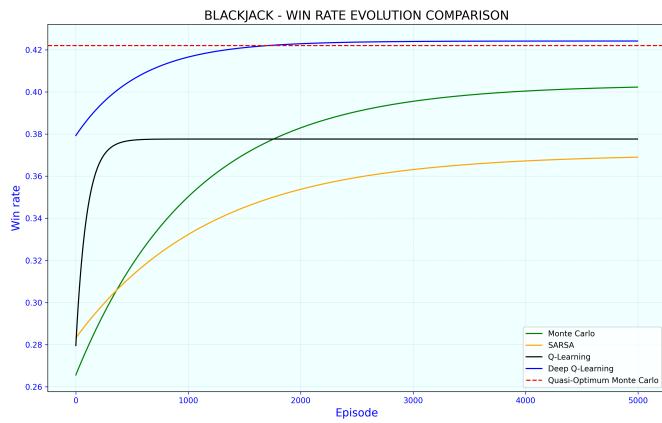


Figura 12: Comparação da evolução da taxa de vitórias dentre os agentes implementados.

Go ou xadrez) e simulações de robótica.

Explorar técnicas de transferência de aprendizado também pode ser útil para adaptar políticas aprendidas em um domínio para resolver problemas em outro domínio relacionado, acelerando o processo de aprendizado e melhorando a robustez do agente.

Uma continuação natural deste trabalho é investigar ambientes mais desafiadores, validando a aplicabilidade dos algoritmos discutidos em problemas mais complexos, como navegação autônoma, controle de sistemas dinâmicos não lineares e jogos multiagente. A inclusão de técnicas de aprendizado por imitação, onde o agente aprende observando demonstrações de especialistas, pode proporcionar uma vantagem adicional em cenários onde a exploração segura é crítica.

Em resumo, este estudo mostrou a eficácia dos algoritmos clássicos de RL na resolução de problemas simples. No entanto, desafios mais complexos exigirão a adoção e adaptação de técnicas de RL mais avançadas, além da combinação de diferentes abordagens para alcançar um desempenho ideal.

REFERÊNCIAS

- [1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998.
- [2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.
- [3] C. J. C. H. Watkins and P. Dayan, “Q-learning,” *Machine Learning*, vol. 8, no. 3-4, pp. 279–292, 1992.
- [4] G. A. Rummery and M. Niranjan, “On-line Q-learning using connectionist systems,” University of Cambridge, Department of Engineering, Tech. Rep., 1994.
- [5] R. Bellman, *Dynamic Programming*. Princeton University Press, 1957.
- [6] A. L. Samuel, “Some studies in machine learning using the game of checkers,” *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, 1959.
- [7] J. Geiser and T. Hasseler, “Beating Blackjack - A Reinforcement Learning Approach,” Stanford University, 2020.

VI. REPOSITÓRIO

Os códigos-fonte desenvolvidos para este estudo, incluindo as implementações dos algoritmos de RL e os *scripts* de treinamento e de simulação, estão disponíveis no repositório público:

- <https://github.com/lucasll37/RL-BlackJack>


```

67     G = self.gamma * G + reward
68     self.N[self.map_state_Q[state]] += 1
69     old_value = self.Q[self.map_state_Q[state], action]
70     new_value = old_value + (1 / self.N[self.map_state_Q[state]]) * (
71         G - old_value
72     )
73     self.Q[self.map_state_Q[state], action] = new_value
74
75     if (
76         isinstance(validate_each_episodes, int)
77         and episode % validate_each_episodes == 0
78     ):
79         self.delta = np.linalg.norm(self.Q - self.last_Q)
80         self.last_Q = self.Q.copy()
81         self.validation(episode, episodes, epsilon_val, verbose, save)
82
83     elif verbose:
84         print(f"Episode: {episode:7d}/{episodes}, epsilon: {self.epsilon:.5f}")
85
86     self.epsilon *= epsilon_decay_factor
87
88
89 if __name__ == "__main__":
90
91     agent = BJAgent_MonteCarlo()
92     agent.learn(
93         episodes=5_000,
94         final_epsilon=1e-2,
95         validate_each_episodes=5,
96         verbose=True,
97     )
98
99     history = agent.plot_history(return_fig=True)
100    policy = agent.plot_policy(return_fig=True)
101
102    history.savefig(f"./images/{agent.name}_history.png", dpi=300, format="png")
103    policy.savefig(f"./images/{agent.name}_policy.png", dpi=300, format="png")

```

A.2 SARSA

```

1  from base import BJAgent
2  from utils import reward_engineering
3  import numpy as np
4
5
6
7  class BJAgent_Sarsa(BJAgent):
8
9      def __init__(
10          self,
11          render_mode=None,
12          gamma=1,
13          initial_epsilon=1,
14          natural=False,
15          sab=False,
16          max_iteration=500,
17      ):
18          super().__init__(
19              render_mode=render_mode,
20              gamma=gamma,
21              initial_epsilon=initial_epsilon,
22              natural=natural,
23              sab=sab,
24              max_iteration=max_iteration,
25          )
26          self.name = "SARSA"
27
28      def learn(
29          self,
30          episodes=10_000,
31          final_epsilon=0.01,
32          epsilon_decay=None,
33          epsilon_val=None,
34          validate_each_episodes=None,
35          verbose=True,
36          save=True,
37      ):
38
39          epsilon_decay_factor = self.epsilon_update(
40              episodes, validate_each_episodes, final_epsilon, epsilon_decay
41          )
42
43          for episode in range(1, episodes + 1):
44
45              state, _ = self.env.reset()
46              done = False
47              iteration = 0
48              self.last_Q = self.Q.copy()
49
50              while not done and iteration < self.max_iteration:
51
52                  iteration += 1
53
54                  action = self.epsilon_greedy_policy(state, epsilon=self.epsilon)
55                  next_state, reward, done, truncated, _ = self.env.step(action)
56
57                  done = done or truncated
58                  reward = reward_engineering(state, action, reward)
59
60                  next_action = self.epsilon_greedy_policy(
61                      next_state, epsilon=self.epsilon
62                  ) # epsilon-greedy policy
63                  self.N[self.map_state_Q[state]] += 1
64                  old_value = self.Q[self.map_state_Q[state], action]
65                  new_value = old_value + (1 / self.N[self.map_state_Q[state]]) * (
66                      reward
67                      + self.gamma * self.Q[self.map_state_Q[next_state], next_action]
68                      - old_value

```

```

69     )
70     self.Q[self.map_state_Q[state], action] = new_value
71     state = next_state
72
73     if (
74         isinstance(validate_each_episodes, int)
75         and episode % validate_each_episodes == 0
76     ):
77         self.delta = np.linalg.norm(self.Q - self.last_Q)
78         self.last_Q = self.Q.copy()
79         self.validation(episode, episodes, epsilon_val, verbose, save)
80
81     elif verbose:
82         print(f"Episode: {episode:7d}/{episodes}, epsilon: {self.epsilon:.5f}")
83
84     self.epsilon *= epsilon_decay_factor
85
86
87 if __name__ == "__main__":
88
89     agent = BJAgent_Sarsa()
90     agent.learn(
91         episodes=5_000, final_epsilon=1e-2, validate_each_episodes=5, verbose=True
92     )
93
94     history = agent.plot_history(return_fig=True)
95     policy = agent.plot_policy(return_fig=True)
96
97     history.savefig(f"./images/{agent.name}_history.png", dpi=300, format="png")
98     policy.savefig(f"./images/{agent.name}_policy.png", dpi=300, format="png")

```

A.3 Q-Learning

```

1 import numpy as np
2 from base import BJAgent
3 from utils import reward_engineering
4
5
6
7 class BJAgent_QLearning(BJAgent):
8
9     def __init__(
10         self,
11         render_mode=None,
12         gamma=1,
13         initial_epsilon=1,
14         natural=False,
15         sab=False,
16         max_iteration=500,
17     ):
18         super().__init__(
19             render_mode=render_mode,
20             gamma=gamma,
21             initial_epsilon=initial_epsilon,
22             natural=natural,
23             sab=sab,
24             max_iteration=max_iteration,
25         )
26         self.name = "QLearning"
27
28     def learn(
29         self,
30         episodes=10_000,
31         final_epsilon=0.01,
32         epsilon_decay=None,
33         epsilon_val=None,
34         validate_each_episodes=None,
35         verbose=True,
36         save=True,
37     ):
38
39         epsilon_decay_factor = self.epsilon_update(
40             episodes, validate_each_episodes, final_epsilon, epsilon_decay
41         )
42         done = True
43
44         for episode in range(1, episodes + 1):
45
46             state, _ = self.env.reset()
47             done = False
48             iteration = 0
49             self.last_Q = self.Q.copy()
50
51             while not done and iteration < self.max_iteration:
52
53                 iteration += 1
54
55                 action = self.epsilon_greedy_policy(state, epsilon=self.epsilon)
56                 next_state, reward, done, truncated, _ = self.env.step(action)
57
58                 done = done or truncated
59                 reward = reward_engineering(state, action, reward)
60
61                 next_action = self.epsilon_greedy_policy(
62                     next_state, epsilon=0
63                 ) # greedy policy
64                 self.N[self.map_state_Q[state]] += 1
65                 old_value = self.Q[self.map_state_Q[state], action]
66                 new_value = old_value + (1 / self.N[self.map_state_Q[state]]) * (
67                     reward
68                     + self.gamma * self.Q[self.map_state_Q[next_state], next_action]

```

```

69             - old_value
70         )
71         self.Q[self.map_state_Q[state], action] = new_value
72         state = next_state
73
74     if (
75         isinstance(validate_each_episodes, int)
76         and episode % validate_each_episodes == 0
77     ):
78         self.delta = np.linalg.norm(self.Q - self.last_Q)
79         self.last_Q = self.Q.copy()
80         self.validation(episode, episodes, epsilon_val, verbose, save)
81
82     elif verbose:
83         print(f"Episode: {episode:7d}/{episodes}, epsilon: {self.epsilon:.5f}")
84
85     self.epsilon *= epsilon_decay_factor
86
87
88 if __name__ == "__main__":
89
90     agent = BJAgent_QLearning()
91     agent.learn(
92         episodes=5_000,
93         final_epsilon=1e-2,
94         epsilon_val=0,
95         validate_each_episodes=5,
96         verbose=True,
97     )
98
99     history = agent.plot_history(return_fig=True)
100    policy = agent.plot_policy(return_fig=True)
101
102    history.savefig(f"./images/{agent.name}_history.png", dpi=300, format="png")
103    policy.savefig(f"./images/{agent.name}_policy.png", dpi=300, format="png")

```

A.4 Deep Q-Learning

```

1 import os
2
3 os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"
4
5
6 import numpy as np
7 import random
8 import tensorflow as tf
9 from keras.models import Sequential
10 from keras.layers import Dense, Input
11 from keras.optimizers import Adam
12 from keras.initializers import RandomUniform
13 from collections import deque
14 from base import BJAgent
15 from utils import reward_engineering
16 import gymnasium as gym
17
18
19 class BJAgent_DeepQLearning(BJAgent):
20
21     def __init__(
22         self,
23         render_mode=None,
24         gamma=1,
25         initial_epsilon=1,
26         max_iteration=500,
27         maxlen_deque=4_096,
28     ):
29         super().__init__(
30             render_mode=render_mode,
31             gamma=gamma,
32             initial_epsilon=initial_epsilon,
33             max_iteration=max_iteration,
34         )
35         self.name = "DeepQLearning"
36         self.model = self._build_model()
37         self.replay_buffer = deque(maxlen=maxlen_deque)
38
39     def _build_model(self):
40         initializer = RandomUniform(minval=-0.1, maxval=0.1)
41
42         model = Sequential()
43         model.add(Input(shape=(self.dimensions_states,)))
44         model.add(Dense(16, activation="relu", kernel_initializer=initializer))
45         model.add(Dense(8, activation="relu", kernel_initializer=initializer))
46         model.add(
47             Dense(self.n_actions, activation="linear", kernel_initializer=initializer)
48         )
49
50         model.compile(optimizer=Adam(learning_rate=0.001), loss="mse")
51
52         return model
53
54     def epsilon_greedy_policy(self, state, epsilon):
55         if np.random.rand() < epsilon:
56             return np.random.choice(self.n_actions)
57
58         else:
59             state_array = np.array(state).reshape(1, 3)
60             q_values = self.model.predict(state_array, verbose=0)[0]
61
62             return np.argmax(q_values)
63
64     def update_policy(self, batch_size=32):
65
66         if len(self.replay_buffer) < batch_size:
67             return

```

```

69 minibatch = random.sample(self.replay_buffer, batch_size)
70 states = list()
71 targets = list()
72
73 for state_array, action, reward, next_state_array, done in minibatch:
74     target = self.model.predict(state_array, verbose=0)
75     next_action = self.epsilon_greedy_policy(next_state_array, epsilon=0)
76
77     if not done:
78         target[0][action] = (
79             reward
80             + self.gamma
81             * self.model.predict(next_state_array, verbose=0)[0][next_action]
82         )
83
84     else:
85         target[0][action] = reward
86
87     states.append(state_array[0])
88     targets.append(target[0])
89
90 _ = self.model.fit(np.array(states), np.array(targets), epochs=1, verbose=0)
91
92 return
93
94 def learn(
95     self,
96     episodes=100,
97     final_epsilon=0.01,
98     epsilon_decay=None,
99     epsilon_val=None,
100    validate_each_episodes=None,
101    verbose=True,
102    save=True,
103):
104
105     epsilon_decay_factor = self.epsilon_update(
106         episodes, validate_each_episodes, final_epsilon, epsilon_decay
107     )
108     done = True
109
110     for episode in range(1, episodes + 1):
111
112         state, _ = self.env.reset()
113         done = False
114         iteration = 0
115         self.last_Q = self.Q.copy()
116
117         while not done and iteration < self.max_iteration:
118
119             iteration += 1
120
121             action = self.epsilon_greedy_policy(state, epsilon=self.epsilon)
122             next_state, reward, done, truncated, _ = self.env.step(action)
123
124             done = done or truncated
125             reward = reward_engineering(state, action, reward)
126             state_array = np.array(state, dtype=np.float32).reshape(1, -1)
127             next_state_array = np.array(next_state, dtype=np.float32).reshape(1, -1)
128             self.replay_buffer.append(
129                 (state_array, action, reward, next_state_array, done)
130             )
131             self.update_policy()
132             state = next_state
133
134             if (
135                 isinstance(validate_each_episodes, int)
136                 and episode % validate_each_episodes == 0
137             ):

```

```
138         self.validation(episode, episodes, epsilon_val, verbose, save)
139
140     elif verbose:
141         print(f"Episode: {episode:7d}/{episodes}, epsilon: {self.epsilon:.5f}")
142
143     self.epsilon *= epsilon_decay_factor
144
145
146 if __name__ == "__main__":
147
148     agent = BJAgent_DeepQLearning()
149     agent.learn(
150         episodes=5_000,
151         final_epsilon=1e-2,
152         epsilon_val=0,
153         validate_each_episodes=5,
154         verbose=True,
155     )
156
157     history = agent.plot_history(return_fig=True)
158     policy = agent.plot_policy(return_fig=True)
159
160     history.savefig(f"./images/{agent.name}_history.png", dpi=300, format="png")
161     policy.savefig(f"./images/{agent.name}_policy.png", dpi=300, format="png")
```