SCIENTIA PLENA VOL. X, NUM. X 202X
www.scientiaplena.org.br doi: 10.14808/sci.plena.202X.XXXXXX

Análise comparativa no jogo Pong: DQN e MCTS

Comparative analysis in the game Pong: DQN and MCTS

E. C. Rodrigues[1]; F. P. S. Sá[1]; J. V. Santos[1] L. A. Viana[1]; S. A. Araujo[1]

[1]*Departamento de Computação, Universidade Federal de Sergipe, 49100-000, São Cristóvão-Sergipe, Brasil*

sauloalmeida@academico.ufs.br
(Recebido em 24 de Fevereiro de 2025; aceito em dia de mês de ano)

Este estudo buscou desenvolver agentes de Inteligência Artificial (IA) capazes de jogar o clássico jogo do Atari Pong, para competir com a máquina e entre os mesmos como forma de verificar qual abordagem obteve melhores resultados. Os agentes criados foram baseados em um modelo Deep Q-Learning (DQN) e um outra abordagem que envolve o modelo DQN integrado a um algoritmo de Monte Carlo Tree Search (MCTS). Os resultados obtidos comprovam a promissora utilização de algoritmos de Aprendizagem por Reforço (RL) para enfrentar situações complexas, como os jogos Atari 2600. Conclui-se que essas abordagens não apenas podem ser úteis, como também fornecem insights valiosos para futuras pesquisas e aplicações em outras áreas da Inteligência Artificial.
Palavras-chave: IA, DQN, MCTS.

This study aimed to develop Artificial Intelligence (AI) agents capable of playing the classic Atari Pong game, to compete against the machine and among themselves in order to determine which approach achieved better results. The agents created were based on a Deep Q-Learning (DQN) model and another approach that involves the DQN model integrated with a Monte Carlo Tree Search (MCTS) algorithm. The results obtained demonstrate the promising use of Reinforcement Learning (RL) algorithms to tackle complex situations, such as Atari 2600 games. It is concluded that these approaches can not only be useful but also provide valuable insights for future research and applications in other areas of Artificial Intelligence.
Keywords: AI, DQN, MCTS.

1. INTRODUÇÃO

Jogos do Atari têm sido um meio de experimentação de algoritmos capazes de realizar comportamentos humanos, destacando-se com uma alta precisão em seus objetivos. Para este trabalho, foi escolhido o jogo Pong através do framework da *Arcade Learning Environment* (ALE). Atualmente há diversas abordagens para lidar com esse tipo de problema, desde estratégias que visam ensinar o agente a agir como um humano a algoritmos que buscam fazer o agente aprender uma política com base em recompensas.

O uso de algoritmos de aprendizado por reforço trouxe resultados superiores aos resultados obtidos por aprendizado supervisionado em humanos, o que demonstra o potencial de técnicas mais precisas e que não precisam de uma base de conhecimento, senão uma política de recompensas positivas e negativas [1].

Segundo Haddad et al. (2021) [2] o uso de agente baseado em DQN superou os resultados obtidos por um agente baseado apenas no algoritmo do Q-Learning, além de apresentar um tempo 10 vezes mais rápido para obter esses resultados, em jogos da plataforma Unity. Esse estudo comparou os resultados obtidos para as taxas de vitórias para ambos modelos, e constatou que o Q-Learning conseguiu uma taxa de vitórias de 70% contra 90% do algoritmo DQN.

De acordo com Fu et al. (2016) [3], o uso de MCTS em regiões que possuem altas recompensas, contribuem para que o agente tome decisões que são mais promissoras, fazendo com que os parâmetros sejam ajustados para que haja mais *exploitation* do que *exploration*.

Sendo assim, este trabalha busca fazer uma análise comparativa entre os algoritmos de um agente apenas baseado em DQN com uma base de treinamento do dobro da base de treinamento de um segundo treinado com um DQN integrado com uma MCTS, ambos aplicados ao jogo Pong e verificar qual dos agentes consegue ter uma maior taxa de vitórias, bem como uma convergência mais rápida e robusta.

2. METODOLOGIA

O estudo foi conduzido na plataforma *Colab* do Google, usando recursos de computação em nuvem. Os códigos foram desenvolvidos na linguagem de programação Python, para isso foram utilizadas bibliotecas como *Gymnasium*, *Torch*, *Numpy* e *Stable Baselines*.

Foram treinados dois agentes, o DQN puro e o DQN com MCTS (para aprimorar a seleção de ações). O treinamento do DQN puro utilizou 2 milhões (2M) de passos, enquanto o DQN com MCTS utilizou metade desse número, apenas 1 milhão (1M). Além disso, cada modelo DQN foi criado com a política *CnnPolicy* e com um *Buffer Size* de 100.000.

Após o treinamento, os dois modelos foram salvos para, posteriormente, serem avaliados em 100 partidas. Cada partida (episódio) é vencida por quem atingir 7 pontos primeiro. Ao final, a taxa de vitória é calculada para cada agente.

2.1 Frameworks e Bibliotecas

2.1.1 ALE

Para o desenvolvimento e teste de agentes de aprendizado por reforço, foi utilizado o framework ALE que possibilita o suporte a jogos do Atari 2600, extração automática de pontuações e sinais de fim de jogo, e ferramentas de visualização [4].

2.1.2 Gymnasium

Já na realização da avaliação e testes dos algoritmos de aprendizado por reforço desenvolvidos, foi empregado a biblioteca *Gymnasium* que facilita a criação de benchmarks para análise da performance dos algoritmos [5].

2.1.3 PyTorch

O framework *PyTorch* foi aplicado para o aprendizado profundo, proporcionando um conjunto de ambientes vasto para a construção e treinamento de redes neurais [6].

2.1.3 *Stable Baselines3*

Stable Baselines3 (SB3) é um framework de aprendizado por reforço em *Python*, que fornece implementações eficientes e robustas de algoritmos populares, como DQN, PPO e A2C [7].

2.2 Arquitetura DQN

2.2.1 Descrição

A arquitetura do DQN utilizado segue o modelo clássico introduzido por DeepMind, composta por três camadas convolucionais para extração de características dos frames do jogo (32 filtros de 8×8, 64 filtros de 4×4 e 64 filtros de 3×3, todas com ativação ReLU), seguidas por uma camada totalmente conectada de 512 neurônios e uma camada de saída correspondente ao número de ações disponíveis. O treinamento utiliza Replay Buffer para armazenar experiências e Target Network para maior estabilidade no aprendizado. A função de perda é baseada na Equação de *Bellman*, e a otimização é feita com *Adam Optimizer* e taxa de aprendizado 1e-4.

2.2.2 Pseudocódigo

```
CLASS DQN:

conv    ' Objeto Sequential para camadas convolucionais

fc      ' Objeto Sequential para camadas totalmente conectadas

FUNCTION DQN(input_shape, n_actions):
    ' Inicializar camadas convolucionais
    conv = SEQUENTIAL(
        CONV2D(input_channels=input_shape[0], out_channels=32,
kernel_size=8, stride=4),
        RELU(),
        CONV2D(input_channels=32, out_channels=64, kernel_size=4,
stride=2),
        RELU(),
        CONV2D(input_channels=64, out_channels=64, kernel_size=3,
stride=1),
        RELU()
    )

    ' Calcular o tamanho da saída após as camadas convolucionais
    conv_out_size = GET_CONV_OUT(input_shape)

    ' Inicializar camadas totalmente conectadas
    fc = SEQUENTIAL(
        LINEAR(input_size=conv_out_size, output_size=512),
        RELU(),
        LINEAR(input_size=512, output_size=n_actions)
    )

END FUNCTION

' Função de forward para processar os dados de entrada

FUNCTION FORWARD(x):

    ' Passar a entrada pelas camadas convolucionais

    conv_out = RESHAPE(conv(x), batch_size=x.size[0], -1)  ' Achatar a
saída

    ' Passar pelas camadas totalmente conectadas e retornar o resultado

    RETURN fc(conv_out)

END FUNCTION


END CLASS
```

2.3 Arquitetura DQN + MCTS

2.3.1 Descrição

DQN com MCTS mantém a mesma arquitetura de rede neural do DQN puro, mas aprimora a seleção de ações incorporando uma árvore de busca Monte Carlo (MCTS). Nesse modelo, o DQN calcula os Q-values do estado atual, e o MCTS expande uma árvore de busca simulando múltiplas sequências de ações, usando a estratégia UCT (Upper Confidence Bound for Trees) para balancear exploração e exploração. Após várias simulações, os valores dos nós são atualizados com base nos Q-values e nas recompensas obtidas, permitindo que a ação com maior valor esperado seja escolhida. Essa combinação torna o agente mais eficiente ao avaliar futuras decisões antes de agir.

2.3.2 Pseudocódigo

```
CLASS DQN_MCTS:

' Seleciona uma ação usando MCTS

FUNCTION SELECIONAR_ACAO_COM_MCTS(env, num_simulations):

    RETURN MCTS(env, self, num_simulations).ESCOLHER_MELHOR_ACAO()

END FUNCTION


END CLASS

CLASS NO_MCTS:

' Inicializa um novo nó

FUNCTION NO_MCTS(state, parent, action):

    INICIALIZAR_ESTADO(state, parent, action)

END FUNCTION

' Verifica se o nó está totalmente expandido

FUNCTION ESTA_EXPANDIDO(action_space):

    RETURN NUMERO_DE_FILHOS == action_space

END FUNCTION

' Seleciona o melhor filho com base no UCT

FUNCTION MELHOR_FILHO():

    RETURN FILHO_COM_MELHOR_BALANCEAMENTO_DE_VALOR_E_EXPLORACAO()

END FUNCTION
```

```
END CLASS

FUNCTION MCTS(env, agent, simulations):
RETURN raiz.MELHOR_FILHO()

END FUNCTION
```

2.4 Ambiente de Hardware

2.4.1 CPU

2 vCPUs (virtual CPUs) Intel Xeon @ 2.20GHz, que é consideravelmente razoável para muitas tarefas de análise de dados e aprendizado de máquina.

2.4.2 GPU

Nvidia Tesla P100, oferecendo até 4.7 teraflops de precisão dupla e 9.3 teraflops de precisão simples, garantindo o treinamento dos agentes e a execução de múltiplas simulações em paralelo.

2.4.3 RAM

Para garantir o desempenho das operações a serem feitas, foi utilizado uma máquina virtual com 12.7GB de memória RAM.

3. EXPERIMENTOS

Nesta seção, são apresentados como foram feitos os experimentos realizados para avaliar o desempenho dos algoritmos de aprendizado por reforço, especificamente os agentes DQN e MCTS, no ambiente de jogo do Pong. Os experimentos têm como objetivo verificar a eficácia dos algoritmos propostos na aprendizagem e otimização de estratégias de jogo, bem como comparar seu desempenho em diferentes cenários.

Para garantir a integridade e validade dos experimentos, foram usados a configuração de ambientes de simulação padronizados e a utilização de métricas de avaliação específicas, de modo que ambos experimentos foram expostos em condições iguais.

3.1 Configuração de Testes

Os experimentos foram conduzidos utilizando o ambiente Pong do ALE, configurado através da biblioteca *Gymnasium*. Utilizamos a biblioteca *Stable Baselines3* para a implementação do agente DQN. Os experimentos foram executados em uma máquina com 2 vCPUs Intel Xeon @ 2.20GHz, uma GPU da Nvidia Tesla P100 e 12,7GB de memória RAM e o sistema operacional Ubuntu 20.04 LTS.

3.2 Métricas de Desempenho

Para avaliação do desempenho e performance dos agentes, foi usado como critério de análise a taxa de vitórias, que é a proporção de episódios em que o agente venceu o jogo. Essa métrica é de suma importância para obtenção dos dados e assim fazer inferências estatísticas.

Vale destacar que foram realizadas 100 partidas/episódios para cada agente.

3.3 Comparação de Abordagens

Para comparar a eficácia de cada agente, foram propostos 3 cenários:

- C01: Agente DQN (2M) vs Máquina
- C02: Agente DQN (1M) + MCTS vs Máquina
- C03: Pontuação Total do Agente DQN (2M) vs Agente DQN (1M) + MCTS

Cada cenário tem o objetivo de levantar insights sobre a eficácia e as limitações de cada abordagem, bem como identificar oportunidades para futuras melhorias e otimizações no treinamento dos agentes de aprendizado por reforço.

4. RESULTADOS E DISCUSSÃO

Como pode ser visto na Figura 1, os experimentos realizados no Cenário C01, onde um agente DQN treinado por 2 milhões de episódios enfrentou um adversário controlado pela máquina demonstram que, embora o agente DQN tenha obtido um desempenho razoável, ele foi superado pelo adversário controlado pela máquina.

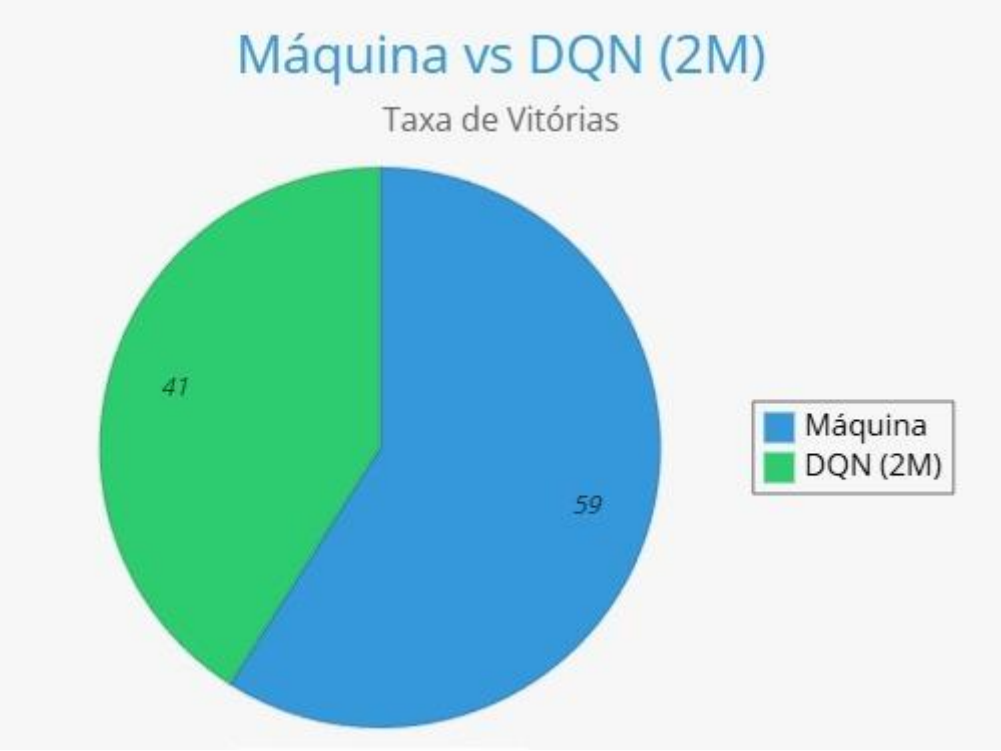

Figura 1: C01: Agente DQN (2M) vs Máquina. Fonte: Autoria Própria

Isso mostra que alguns motivos foram cruciais para que obtivesse esses resultados, como um tempo de treinamento insuficiente para que o agente conseguisse aprender de forma a obter resultados mais equilibrados.

Já a Figura 2 mostra os experimentos realizados no Cenário C02, onde um agente híbrido DQN treinado por 2 milhões de episódios, combinado com o MCTS, enfrentou um adversário controlado pela máquina, mostraram que o agente híbrido DQN + MCTS se mostrou extremamente ineficaz, sendo pior que a implementação do DQN puro.

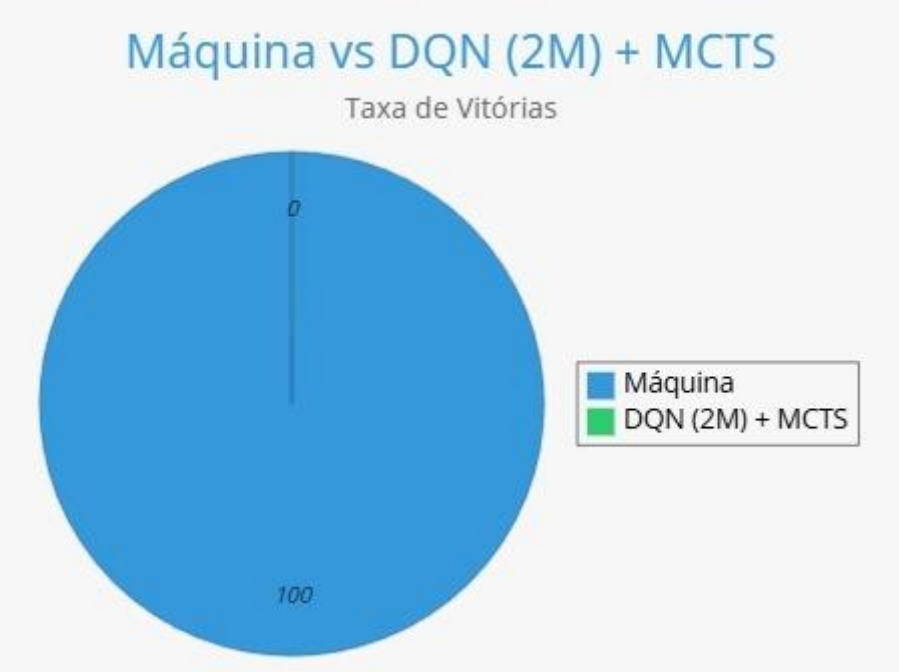

Figura 2: I C01: Agente DQN (2M) + MCTS vs Máquina. Fonte: Autoria Própria

A combinação dos algoritmos DQN e MCTS pode não ter sido plenamente eficaz neste contexto específico. Algumas razões plausíveis para esses números ruins podem ter sido o fato de o MCTS realizar buscas profundas e simulações para prever os resultados de ações futuras, causando uma integração entre essas duas abordagens um conflito na forma como o agente explora e aproveita as ações, resultando em uma ineficiência no aprendizado.

Já a Figura 3 traz uma comparação “ponto a ponto” entre os agentes desenvolvidos em suas partidas contra o adversário controlado pela máquina, como esperado, o agente baseado apenas no DQN pontuou 40 vezes mais que o agente combinado de DQN + MCTS.

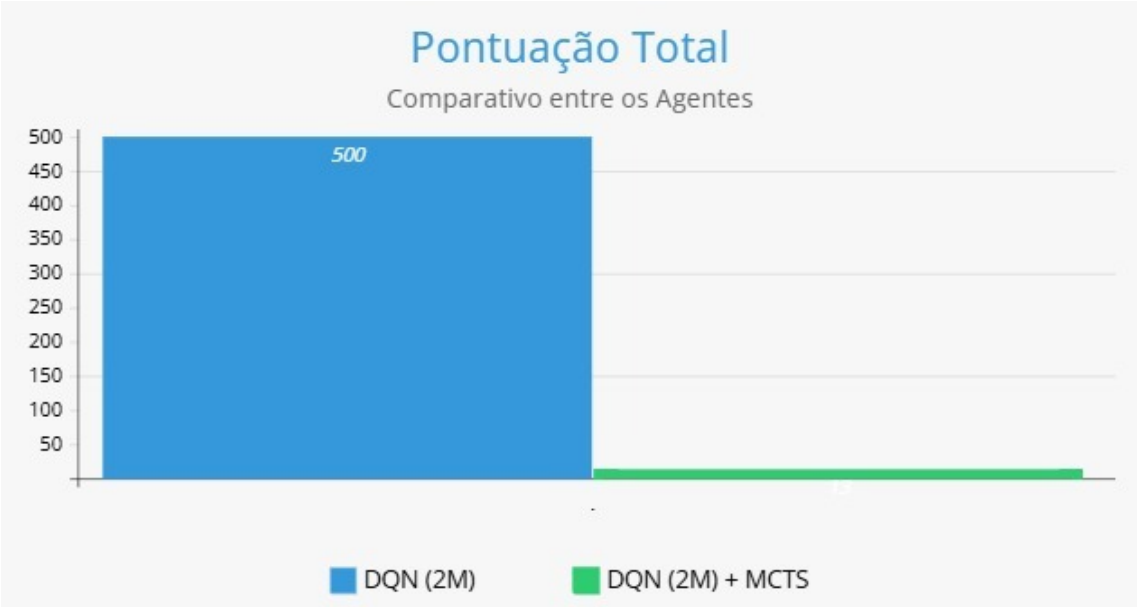

Figura 3: C03: Comparativo de Pontuação Total.Fonte: Autoria Própria

Os resultados destacam a disparidade de precisão entre os algoritmos doa agentes de DQN e DQN + MCTS, sobretudo a falta de calibragem do segundo agente, que reflete o quão ineficaz se tornou a integração do MCTS ao agente, embora o MCTS seja um método robusto para a tomada de decisão em árvores de busca, sua integração com o DQN pode ter encontrado desafios específicos, como o conflito de políticas, a latência que o algoritmo do MCTS pode ter provocado para tomada de decisões rápidas, algo que faz total diferença em jogos de tempo real. Por fim, a modelagem do ambiente aliado a uma política não tão estável pode não ter sido bom para essa estratégia, visto que provocaria a tomada de decisões precipitadas, e geraria uma política subótima.

5. CONCLUSÃO

A incorporação do MCTS no processo de decisão do agente DQN resultou em uma piora significativa do desempenho, evidenciada pela inferioridade do DQN+MCTS (0 vitórias) sobre o DQN Puro (41 vitórias) em 100 episódios respectivamente. A abordagem híbrida ainda pode ser promissora, sugerindo que a integração de métodos de busca pode fornecer uma vantagem estratégica, que está sujeita a alguns fatores, como modelagem do ambiente, poder computacional e a dinamicidade do ambiente. Vale ressaltar que o agente com MCTS estava realizando apenas uma simulação por estado, ou seja, ao aumentar o número de simulações, o agente pode ter um aproveitamento maior.

Em síntese, pode ser inferido que um treinamento com um número insuficiente de treinamento provoca agentes ineficazes quando se analisa os dados estatisticamente, mesmo que em algumas exceções de episódios ele tenha se saído bem. Apenas com um treinamento suficiente o agente pode convergir a uma boa política, onde assim ele teria capacidade de desenvolver estratégias robustas e consistentes.

Dessa forma, fica como sugestões para trabalhos futuros a implementação de um treinamento com no mínimo uns 10.000.000 de passos, o que levaria o agente a ter uma política mais robusta, a depender dos resultados obtidos realizar alterações nos hiper parâmetros de forma empírica para garantir que o agente não sofra com *overfitting*.

Ademais, com um agente mais eficaz, seria interessante a implementação do MCTS como forma de averiguar se o desempenho dele também iria piorar, melhorar ou não afetaria significativamente, visto que ela incrementaria em um agente mais consistente.

6. REFERÊNCIAS BIBLIOGRÁFICAS

1. Silver, David, et al. "Mastering the game of go without human knowledge." nature 550.7676 (2017): 354-359. https://tinyurl.com/mastering-go
2. HADDAD, Gabriel Prudencio; JULIA, Rita Maria Silva; FARIA, Matheus Prado Prandini. Técnicas de Aprendizado por Reforço Aplicadas em Jogos Eletrônicos na Plataforma Geral do Unity. In: ENCONTRO NACIONAL DE INTELIGÊNCIA ARTIFICIAL E COMPUTACIONAL (ENIAC), 18. , 2021, Evento Online. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2021 . p. 571-582. ISSN 2763-9061. DOI: https://doi.org/10.5753/eniac.2021.18285.
3. Fu, M. C. 2016. “AlphaGo and Monte Carlo Tree Search: The Simulation Optimization Perspective”. In Proceedings of the 2016 Winter Simulation Conference, edited by T. M. K. Roeder et al., 659–670. Piscataway, NJ: IEEE.
4. Arcade Learning Environment (ALE). Farama Foundation. Disponível em: https://ale.farama.org/. Acesso em: 24 fev. 2025.
5. Gymnasium Documentation. Farama Foundation. Disponível em: https://gymnasium.farama.org/. Acesso em: 24 fev. 2025.
6. PyTorch Documentation. PyTorch. Disponível em: https://pytorch.org/docs/stable/index.html. Acesso em: 24 fev. 2025.
7. Stable-Baselines3 Documentation. Stable-Baselines3. Disponível em: https://stable-baselines3.readthedocs.io/en/master/. Acesso em: 24 fev. 2025.