UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Lucas Marques Pinho Tiago

Identificação de Padrões de Software para Instanciação de Agentes DQN em Jogos 2D

Uberlândia, Brasil

2025

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Lucas Marques Pinho Tiago

Identificação de Padrões de Software para Instanciação de Agentes DQN em Jogos 2D

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Orientador: Daniel Duarte Abdala

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2025

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

T551 2025

Tiago, Lucas Marques Pinho, 2004-
Identificação de Padrões de Software para Instanciação de Agentes DQN em Jogos 2D [recurso eletrônico] / Lucas Marques Pinho Tiago. - 2025.

Orientador: Daniel Duarte Abdala.
Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Uberlândia, Graduação em Ciência da Computação.
Modo de acesso: Internet.
Inclui bibliografia.
Inclui ilustrações.

1. Computação. I. Abdala, Daniel Duarte,1979-, (Orient.). II. Universidade Federal de Uberlândia. Graduação em Ciência da Computação. III. Título.

CDU: 681.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:
Gizele Cristine Nunes do Couto - CRB6/2091
Nelson Marcos Ferreira - CRB6/3074

Lucas Marques Pinho Tiago

Identificação de Padrões de Software para Instanciação de Agentes DQN em Jogos 2D

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Trabalho aprovado. Uberlândia, Brasil, 19 de setembro de 2025:

Daniel Duarte Abdala
Orientador

Márcia Aparecida Fernandes

Humberto Luiz Razente

Uberlândia, Brasil
2025

Resumo

O desenvolvimento de agentes inteligentes para NPCs em jogos de computador tem atraído crescente interesse, sobretudo devido aos recentes avanços em inteligência artificial. Nesse contexto, este trabalho aborda a criação e o treinamento de agentes utilizando DQNs, devido ao notável desempenho demonstrado por essas redes em pesquisas recentes. Paralelamente, realiza-se uma análise do framework de criação e treinamento de tais agentes, distinguindo os componentes genéricos daqueles que demandam um ajuste fino para cada um. Para validar essa análise, três agentes distintos foram criados e treinados, cujos resultados e particularidades são detalhadamente exemplificados e analisados neste estudo.

Palavras-chave: Agentes, DQN, jogos 2D.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ALE	Arcade Learning Environment
ANN	Artificial Neural Network
CNN	Rede Neural Convolucional
DDQN	Double Deep Q-Network
DL	Deep Learning
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
DSL	Domain-Specific Language (Linguagem de Domínio Específico)
Dueling DQN	Dueling Q-Network
FIFO	First-In, First-Out (Primeiro a Entrar, Primeiro a Sair)
FPS	Frames Per Second (Quadros por Segundo)
GPU	Unidade de Processamento Gráfico
ML	Machine Learning
NES	Nintendo Entertainment System
NPC	Personagem Não Jogável (Non-Playable Character)
PCA	Análise de Componentes Principais
POMDP	Processos de Decisão de Markov Parcialmente Observáveis
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
RNA	Rede Neural Artificial
ROM	Memória Somente de Leitura (Read-Only Memory)
TDL	Aprendizado por Diferença Temporal

Sumário

1 Introdução

O aprendizado por reforço (Reinforcement Learning - RL) tem se tornado uma abordagem proeminente para se treinar modelos de inteligência artificial capazes de jogar jogos eletrônicos, diferenciando-se do aprendizado supervisionado e não supervisionado, visto que possui uma interação constante entre o agente e o ambiente. Ao explorar e interagir com o ambiente de forma livre, os agentes recebem constantemente recompensas ou penalidades, dependendo das ações que tomam. Essa técnica teve grandes avanços com o advento de DQN (Deep Q-Network), um algoritmo de RL capaz de guardar informações de experiências passadas, minimizando a instabilidade no treinamento.

Jogos são usualmente utilizados como campos de teste (benchmarks) para validar a eficiência e o desempenho de diversos algoritmos de inteligência artificial (HU et al., 2024). Isso se deve ao fato de que, muitas vezes, os jogos representam um ambiente complexo, com um objetivo claro e, normalmente, um conjunto de regras bem definidas, o que facilita a modelagem para um agente de inteligência artificial. Contudo, a utilização dos conceitos de inteligência artificial em jogos vai muito além de benchmarks, já que essas técnicas são muito utilizadas na indústria para criar experiências de jogo mais ricas, como o desenvolvimento de NPCs (personagens não jogáveis) e a geração procedural de conteúdo, fatores que favorecem a imersão e a rejogabilidade para o jogador (SHARIFI; ZHAO; SZAFRON, 2010).

Muitos estudos, como por exemplo o realizado por Fu (FU, 2024), mostram que os algoritmos que apresentam o melhor desempenho para treinar agentes para jogar são o DQN e suas variações, sendo o Dueling DQN o implementado neste estudo. O trabalho de Mnih (MNIH et al., 2013) também demonstra como foi possível treinar o mesmo modelo de rede neural profunda para jogar diversos jogos diferentes, de forma a superar o desempenho humano.

1.1 Objetivos

O objetivo principal deste trabalho é treinar modelos de inteligência artificial que sejam capazes de, a partir da análise da tela do jogo, inferir as sequências de comandos adequadas para avançar com sucesso nas fases. Para alcançar tal objetivo, foram necessários alguns passos, que estão elencados como objetivos específicos a seguir:

- Definir o conjunto de infraestrutura de software necessário para treinar um modelo de forma genérica;

- Definir o ambiente de teste para treinamento e validação;
- Estudar os conceitos necessários para implementar um protótipo;
- Avaliar a viabilidade de implementar um protótipo genérico para diferentes ambientes;
- Iterar sobre o protótipo para melhorá-lo e otimizar o treinamento do modelo;
- Testar os modelos;
- Avaliar os resultados e verificar se os objetivos foram atingidos.

1.2 Justificativa

A indústria de jogos eletrônicos consolidou-se como uma das maiores do mundo, movimentando centenas de bilhões de dólares anualmente (PERI, 2024). Com isso, cresce o interesse no uso de agentes inteligentes para criar experiências mais personalizadas e, principalmente, para dar vida a NPCs (Personagens Não Jogáveis) que interagem com o ambiente de forma dinâmica. NPCs mais inteligentes impactam diretamente indicadores cruciais como a imersão e a rejogabilidade. A importância de ensinar uma IA a jogar reside no fato de que esses modelos podem ser adaptados para otimizar o comportamento dos NPCs, elevando a qualidade da experiência do jogador.

A imersão, por exemplo, está diretamente ligada à naturalidade com que um NPC interage com o ambiente e com o próprio jogador (SHARIFI; ZHAO; SZAFRON, 2010). O mesmo vale para a rejogabilidade, um aspecto importante que motiva o jogador a continuar no jogo após terminar a história principal. Uma boa rejogabilidade não só melhora a opinião dos jogadores, mas também os mantém ativos na comunidade, um fator que diversos modelos de negócio exploram para fins econômicos. Antigamente, as IAs eram desenvolvidas com comportamentos pseudo-inteligentes baseados em scripts de ações; no entanto, esses scripts tornam-se repetitivos rapidamente, o que impacta de forma negativa tanto a imersão quanto a rejogabilidade. NPCs operados por script seguem um conjunto de ações predeterminadas que, uma vez aprendidas pelo jogador, diminuem a diversão e a imersão, o que não ocorre com NPCs inteligentes, que podem apresentar padrões muito mais complexos ou que se adaptam com base nas ações do jogador.

1.3 Hipótese

Em que medida é possível utilizar as tecnologias disponíveis para automatizar o treinamento de agentes em diversos jogos 2D e, a partir desse processo, identificar os padrões de software comuns que constituem a infraestrutura necessária?

A estrutura deste trabalho é a seguinte: o Capítulo 2 inicia-se com a abordagem dos trabalhos relacionados e as justificativas para a seleção dos algoritmos e ferramentas. Em seguida, o Capítulo 3 detalha os conceitos, algoritmos e ferramentas fundamentais utilizados, com exemplos para facilitar a compreensão. O Capítulo 4 explora o funcionamento geral do código, suas estruturas, funções e um diagrama de execução. No Capítulo 5, são apresentados os experimentos, seus parâmetros e os resultados obtidos. O Capítulo 6 dedica-se a identificar os componentes de software genéricos, aplicáveis a qualquer jogo, e aqueles que são específicos para cada implementação. Por fim, o Capítulo 7 traz as conclusões do estudo.

2 Levantamento do estado da arte

A etapa inicial deste trabalho consistiu em uma revisão bibliográfica, ou levantamento do estado da arte, com o objetivo de avaliar a viabilidade do projeto e identificar estudos correlatos. Este processo foi fundamental, pois revelou diversos trabalhos que não apenas corroboram a viabilidade da hipótese deste trabalho, mas também fundamentam a escolha das ferramentas e arquiteturas aqui adotadas.

Playing Atari with Deep Reinforcement Learning, 2013, Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller, relatório técnico - O trabalho de (MNIH et al., 2013) foi um marco no aprendizado por reforço no contexto de jogos eletrônicos, já que os autores treinaram uma rede neural convolucional capaz de jogar sete jogos diferentes de Atari 2600. Para isso, combinaram Q-Learning com redes neurais convolucionais para criar um modelo que recebe como entrada os pixels brutos da tela do jogo e retorna, como saída, os valores estimados para cada ação possível. Os resultados indicam que o agente obteve um bom desempenho, superando inclusive o desempenho humano em vários jogos, o que evidencia a viabilidade dessa técnica para o treinamento de agentes em ambientes iterativos.

Além disso, esse trabalho serviu como uma prova das capacidades de autonomia e flexibilidade do DQN, já que comprovou que a mesma arquitetura, sem ajustes, poderia ser aplicada com sucesso a uma variedade de jogos de Atari distintos. Os experimentos deste relatório foram feitos utilizando o Arcade Learning Environment (ALE), um framework que fornece uma interface para diversos jogos de Atari 2600 e que, posteriormente, se tornou um benchmark para muitos projetos de Deep Reinforcement Learning (DRL). Este trabalho se relaciona com o presente TCC por ser uma ótima referência para a aplicação de redes neurais profundas em jogos 2D, além de comprovar que é possível projetar uma arquitetura DRL genérica que funcione para uma grande gama de jogos.

A Comparison of DQN and Dueling DQN in A Super Mario Environment, 2024, Xiangwei Fu, anais de conferência - Após a consolidação do DQN como uma base estável para pesquisas em DRL, o trabalho de (FU, 2024) teve como objetivo comparar o desempenho dos algoritmos DQN e Dueling DQN no ambiente do jogo Super Mario Bros. O autor compara ambos os algoritmos utilizando o ambiente da biblioteca Gym-Super-Mario-Bros e várias métricas para avaliar o desempenho das duas arquiteturas, como: taxa de sucesso, número de episódios para convergência, estabilidade durante o treinamento e sensibilidade à variação da taxa de aprendizado e do tamanho do

lote (batch size). Os testes foram feitos em níveis de jogo idênticos, com cada treinamento durando um total de 3.000 iterações (epochs). Os resultados mostram que o Dueling DQN teve um menor tempo de convergência na maior parte dos casos, uma menor sensibilidade à taxa de aprendizado e uma melhora constante no treinamento conforme o tamanho do lote foi incrementado.

Por fim, o autor indica que ambos os algoritmos são similares em desempenho, mas que o Dueling DQN pode ter uma vantagem nesse tipo de ambiente específico, e que trabalhos futuros são necessários para explorar sistematicamente os hiperparâmetros a fim de comprovar, de fato, a superioridade do Dueling DQN nesta tarefa. Essa comparação é importante, já que serviu de base para a escolha do Dueling DQN como algoritmo principal deste trabalho.

A Survey of Deep Reinforcement Learning in Video Games, 2019, Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, Dongbin Zhao, revista científica. - Este artigo (SHAO et al., 2019) apresenta uma coleção de abordagens e aplicações de DRL no contexto de jogos eletrônicos. Para isso, os métodos de DRL são organizados e analisados em três categorias principais:

- **Métodos baseados em Valor (Value-Based)**: Buscam aprender uma função que estima a recompensa esperada para cada ação a partir de cada estado. A política consiste em escolher, a cada estado, a ação com o maior valor esperado. O DQN é o principal exemplo dessa categoria.
- **Métodos baseados em Gradiente de Política (Policy-Based)**: Aprendem uma política que mapeia estados para uma distribuição de probabilidade sobre ações. Os parâmetros da política são otimizados a cada iteração, buscando atingir o gradiente esperado. O artigo destaca o Proximal Policy Optimization (PPO) como um exemplo.
- **Métodos baseados em Modelo (Model-Based)**: Criam um modelo do ambiente que simula sua dinâmica, buscando prever o próximo estado e a recompensa gerada, dado o estado e a ação atuais. O agente pode treinar sua política internamente nesse modelo, reduzindo a interação com o ambiente real, para que futuramente seja testado no ambiente real. Um dos principais exemplos atuais é o MuZero.

Cada um desses métodos é comparado e analisado pelos autores em diferentes gêneros de jogos, para mostrar as vantagens e limitações de cada um. Por fim, os autores destacam os grandes progressos recentes do DRL e apontam alguns desafios, como a generalização e a estabilidade do treinamento.

Esse artigo se relaciona com este projeto devido à sua análise sobre o DQN, caracterizando-o como um algoritmo robusto, com alta capacidade de generalização e adaptação, reforçada pelo fato de ter conseguido um bom desempenho contra oponentes nunca antes vistos.

Application of Reinforcement Learning in Games, Lunyuan Hu, Ruike Peng, Junpeng Yang, anais de conferência - O trabalho de (HU; PENG; YANG, 2024) explora os avanços recentes na área de inteligência artificial no contexto de jogos, destacando principalmente três algoritmos de RL que foram implementados e avaliados para diferentes jogos: Q-Learning, Policy Gradient e Actor-Critic. Cada um dos algoritmos foi avaliado, destacando seus principais problemas e indicando, quando aplicável, possíveis soluções.

O Q-learning, um algoritmo baseado em valor, busca a função de valor de ação $Q(s, a)$, a qual estima o valor de cada ação a em um dado estado s, e uma função Alvo, que calcula o valor do estado s' após a ação a ser tomada. Esse algoritmo apresentou dois problemas principais: a alta correlação entre quadros consecutivos viola o princípio de que os dados de treinamento devem ser independentes, e a instabilidade do treinamento quando a mesma rede neural é usada para calcular os valores Q e Alvo. Porém, esses problemas podem ser resolvidos, respectivamente, ao armazenar os quadros em lotes aleatórios e ao criar uma rede auxiliar para calcular o valor Alvo.

O Policy Gradient, algoritmo baseado em gradiente de política, utiliza o gradiente para atualizar a política que mapeia todos os estados para uma distribuição probabilística das ações. O principal problema desse algoritmo ocorre quando há muitos agentes, já que a coordenação e a cooperação desses agentes são difíceis de serem implementadas de forma eficaz, além de a complexidade aumentar significativamente com o número de agentes (WAFA; SANTOSO, 2020).

O Actor-Critic combina os métodos baseados em gradiente e em valor, buscando unir as principais vantagens dos dois algoritmos, onde o "Ator" busca uma política ótima para a escolha das ações, enquanto o "Crítico" avalia a ação escolhida para guiar o "Ator" a essa política ótima. O principal problema do algoritmo reside na instabilidade do treinamento e no alto custo para a coleta de dados de amostra em ambientes complexos. Adicionar uma rede para o "Crítico" e utilizar otimizadores mais sofisticados são as principais soluções para contornar esses problemas.

Por fim, o autor conclui que o melhor algoritmo depende das características do problema escolhido. Isto justifica a implementação de algoritmos baseados em valor neste TCC, já que o Q-learning se destacou em problemas onde o conjunto de ações é conhecido, o que ocorre em todos os experimentos desta monografia.

Maintain Agent Consistency in Surakarta Chess Using Dueling Deep Network with Increasing Batch, 2022, Rian Adam Rajagede, artigo científico. - Este artigo (RAJAGEDE, 2022) tem como principal funcionalidade explicitar um dos maiores problemas dos algoritmos DQN: a inconsistência em ambientes competitivos multiagentes. Isso significa que um agente treinado para enfrentar um único tipo de oponente pode obter um desempenho muito inferior contra um novo tipo de adversário, mesmo que este seja considerado mais fraco. Esse problema é uma grande falha na generalização do DQN, já que o agente treinado aprende uma estratégia ótima apenas contra um tipo de oponente, e não uma política ótima genérica.

Para resolver esse problema, o autor compara variações do algoritmo DQN com uma solução proposta, utilizando como ambiente o jogo Surakarta Chess. A solução foi dividida em duas etapas:

1. Utilizar o Dueling Q-Network (Dueling DQN), uma variação do DQN que calcula o valor $A(s, a)$, a recompensa esperada de cada ação a num estado s, e o valor $V(s)$, que representa a qualidade de um estado s independentemente das ações.

2. Tamanho de lote crescente, que tem como objetivo criar atualizações mais frequentes no começo do treinamento, incentivando a exploração. À medida que o treino avança, as atualizações se tornam menos frequentes, impedindo variações muito grandes nos pesos da rede ao final do treinamento.

Por fim, o autor conclui que os experimentos validaram a hipótese proposta, mas que esses resultados só podem ser considerados para o ambiente específico do jogo Surakarta Chess, e que mais testes devem ser realizados para comprovar essa hipótese em ambientes mais complexos. Com isso, fica evidente que este trabalho se alinha com a sugestão proposta pelo autor, já que o Dueling DQN foi implementado em ambientes consideravelmente mais complexos que o Surakarta Chess.

Improving Bidding and Playing Strategies in the Trick-Taking game Wizard using Deep Q-Networks, 2022, Jonas Schumacher, Marco Pleines, anais de conferência. - Este estudo (SCHUMACHER; PLEINES, 2022) tem como foco principal testar e avaliar a capacidade de aprendizado do algoritmo DQN em um ambiente de jogo complexo, nesse caso, o jogo de cartas Wizard. Para que um algoritmo de RL pudesse ser aplicado a esse problema, o conjunto de ações possíveis foi modelado utilizando dois Processos de Decisão de Markov Parcialmente Observáveis (POMDPs) interligados. O primeiro desses POMDPs serve para o agente decidir qual será a sua aposta, dada a sua mão inicial, enquanto o segundo serve para controlar as jogadas das cartas após o início do jogo.

A fase de treinamento foi realizada utilizando autojogos (self-play), ou seja, o agente DQN aprende a melhorar sua política jogando exclusivamente contra cópias de si mesmo. Já para a parte de avaliação do modelo, o agente jogou um total de 10.000 partidas contra dois tipos diferentes de adversários: o aleatório, que sempre toma decisões aleatórias, e o baseado em regras, que joga com base em um conjunto de regras e heurísticas pré-determinadas.

A conclusão final desse experimento é que o agente DQN é robusto e capaz de generalizar sua estratégia. Mesmo sendo treinado apenas em autojogos, ele conseguiu superar o agente baseado em regras, um tipo de oponente que nunca havia enfrentado antes. Isso indica que o agente DQN aprendeu a utilizar estratégias fundamentalmente sólidas e válidas para o jogo Wizard, sendo mais um dos fatores que reforçam a escolha do DQN como principal algoritmo deste projeto, já que ele consegue aprender por conta própria políticas genéricas e efetivas.

Com base neste levantamento do estado da arte, parâmetros relevantes da proposta inicial puderam ser revistos e refinados, culminando na elaboração do método que será detalhado no Capítulo 4. A seguir, o capítulo 3 apresenta os conceitos que fundamentam o desenvolvimento deste trabalho.

3 Fundamentação Teórica

Para compreender o funcionamento dos algoritmos DQN e Dueling DQN, é importante entender suas bases: o Aprendizado por Reforço (Reinforcement Learning - RL) e as Redes Neurais Profundas (Deep Neural Networks - DNN), tecnologias fundamentais no campo do Aprendizado Profundo (Deep Learning - DL), conceitos estes, discutidos a seguir.

3.1 Reinforcement Learning

O campo de Aprendizado por Reforço é um dos paradigmas fundamentais do Aprendizado de Máquina (Machine Learning - ML) e uma das principais subáreas da Inteligência Artificial. A origem dessa área de estudo não possui uma data definida, porém o RL foi consolidado e formalizado na obra de Sutton e Barto, publicada originalmente em 1998 (SUTTON; BARTO, 2018). Antes de explicar o funcionamento do RL, é importante detalhar os componentes-chave utilizados nessa técnica, a Figura 1 representa o funcionamento geral da técnica.

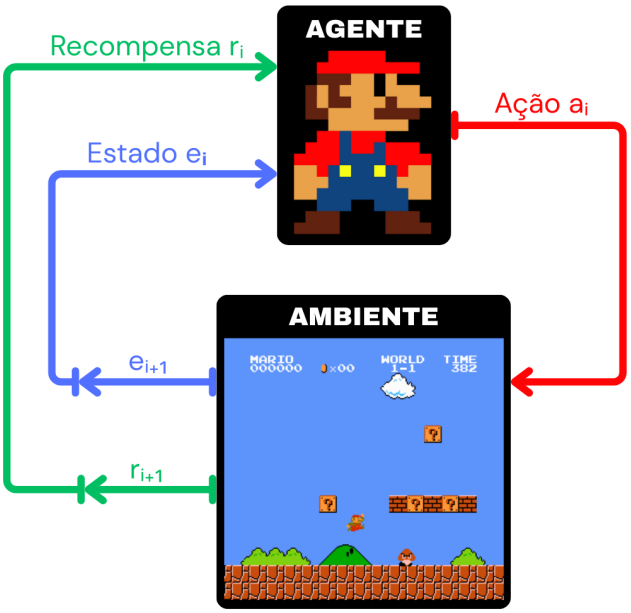

Figura 1 – Funcionamento do RL no contexto do jogo Super Mario Bros.

- **Agente**: Representa a entidade que aprende e toma as decisões;

- **Ambiente**: O mundo externo com o qual o agente interage, na maior parte dos casos o jogo;
- **Estado**: A situação atual do ambiente em um determinado momento;
- **Ação**: Movimento realizado pelo agente em um determinado estado;
- **Recompensa**: Valor numérico (positivo ou negativo) recebido pelo agente como resposta à sua ação;
- **Política**: Estratégia que o agente adota para definir qual ação escolher em um determinado estado.

O RL é uma técnica utilizada para treinar agentes de software em ambientes dinâmicos, buscando imitar o processo de aprendizado humano de tentativa e erro. O cerne dessa técnica está no sistema de recompensa e punição, no qual ações que contribuem para alcançar o objetivo esperado são recompensadas, enquanto ações que prejudicam esse objetivo são penalizadas. O processo de aprendizado do RL pode ser entendido como uma interação contínua entre o agente e o ambiente: a cada passo, o agente analisa o ambiente em que está inserido e escolhe uma ação dentre as possíveis.

O aprendizado em si ocorre depois que o agente recebe um feedback (resposta), que pode ser positivo ou negativo, da ação escolhida. Esse comportamento faz com que os algoritmos de RL aprendam por conta própria qual é a estratégia ótima para alcançar o objetivo em questão, sem que seja necessário um direcionamento para uma estratégia correta. Já que o RL aprende de forma autônoma através da experiência, a interação humana é consequentemente menor, permitindo que o viés (bias) desses algoritmos seja consideravelmente menor se comparado a outras técnicas de ML, como o aprendizado supervisionado.

A escolha de qual ação o agente irá tomar é governada por uma política; essencialmente, a política é a estratégia adotada pelo agente para definir a melhor ação a ser escolhida. As políticas podem ser divididas em duas categorias: determinísticas, onde para cada estado há somente uma ação a ser tomada, ou probabilísticas, que definem uma distribuição de probabilidade para cada uma das ações possíveis em um determinado estado. Portanto, o objetivo final do RL é encontrar uma política ótima, denotada por π^*, a qual se caracteriza por maximizar a recompensa total, denotada retorno (G_t), a partir de um instante t. A formulação mais comum é o retorno descontado:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.1}$$

Onde R_{t+1} é a recompensa recebida no passo $t+1$ e γ é o fator de desconto, um valor entre 0 e 1 que pondera a importância da recompensa futura.

O verdadeiro potencial do RL se destaca em ambientes complexos, com inúmeras regras ou estratégias ótimas, já que é nessas situações que o RL pode encontrar soluções que, de outra forma, não seriam exploradas por seres humanos. Por isso, a escolha de usar esse algoritmo para treinar o agente em um ambiente de jogo é uma opção totalmente viável, visto que um dos primeiros projetos para treinar um agente a jogar foram feitos utilizando conceitos de RL. O principal exemplo é o artigo de Arthur Samuel, de 1959, no qual os princípios do aprendizado por reforço foram usados para ensinar um agente a jogar damas (SAMUEL, 1959).

Para facilitar a compreensão dos conceitos de RL, pode-se tomar um exemplo prático: o processo de ensinar um cachorro a sentar.

- **Agente**: o cachorro, que aprende com suas ações;
- **Ambiente**: o local onde o adestramento acontece;
- **Estado**: a situação do ambiente, por exemplo, o tutor segurando um petisco;
- **Ação**: o que o cachorro faz, dado o estado atual;
- **Recompensa**: se ele sentar, ganha um petisco (recompensa positiva); se fizer algo diferente, não ganha nada (ausência de recompensa);
- **Política**: estratégia que o cachorro aprende a adotar. Ele entende que a melhor ação para receber a recompensa é sentar quando o tutor segura o petisco.

Como RL e DRL são assuntos muito extensos, com várias aplicações e variações, seguem dois dos principais livros utilizados para estudar o tema: o livro de (SUTTON; BARTO, 2018) oferece uma ótima base para aprender sobre RL, enquanto o livro de (MORALES, 2020) foca no entendimento do DRL.

3.2 Deep Neural Networks

As DNNs representam um avanço significativo em relação às Redes Neurais Artificiais (RNAs). Em suas concepções originais, as RNAs eram frequentemente estruturadas com apenas três camadas: uma de entrada (input layer), uma intermediária (hidden layer) e uma de saída (output layer). Conforme demonstrado por Minsky e Papert em seu livro "Perceptrons", essas redes mais simples não conseguiam resolver problemas não linearmente separáveis. Consequentemente, para um vasto conjunto de tarefas, seu poder de decisão era limitado, mostrando-se pouco superior ao de métodos mais simples de análise de dados, como a Análise de Componentes Principais (PCA) em certos contextos de classificação.

A introdução das DNNs superou essa barreira, sendo a principal inovação a adição de múltiplas camadas ocultas, uma característica que se tornou computacionalmente viável graças aos avanços no hardware (especialmente GPUs). A adição dessas camadas intermediárias removeu a limitação da separabilidade linear, permitindo que as DNNs fossem aplicadas com sucesso na resolução dos mais diversos e complexos problemas. Um bom exemplo é o AlphaGo (SILVER et al., 2016), uma DNN criada para jogar o complexo jogo de tabuleiro Go, que posteriormente foi generalizada no AlphaZero, um programa capaz de aprender diversos jogos de tabuleiro através de autojogos (self-play).

Além disso, a estrutura de conexão dos neurônios também evoluiu. Em arquiteturas iniciais, como o Perceptron de Múltiplas Camadas como pode ser observado na Figura 2, era comum um grafo completamente conectado (onde cada neurônio de uma camada se conecta a todos da seguinte); a adição de novas camadas nas DNNs incentivou a experimentação com novas arquiteturas e formas de conectar os neurônios. Por isso, DNN tornou-se um termo genérico que abrange várias arquiteturas distintas, projetadas para resolver problemas específicos. Como exemplos notáveis, podem-se citar as Redes Neurais Convolucionais (CNNs) e as Redes Q Profundas (DQNs).

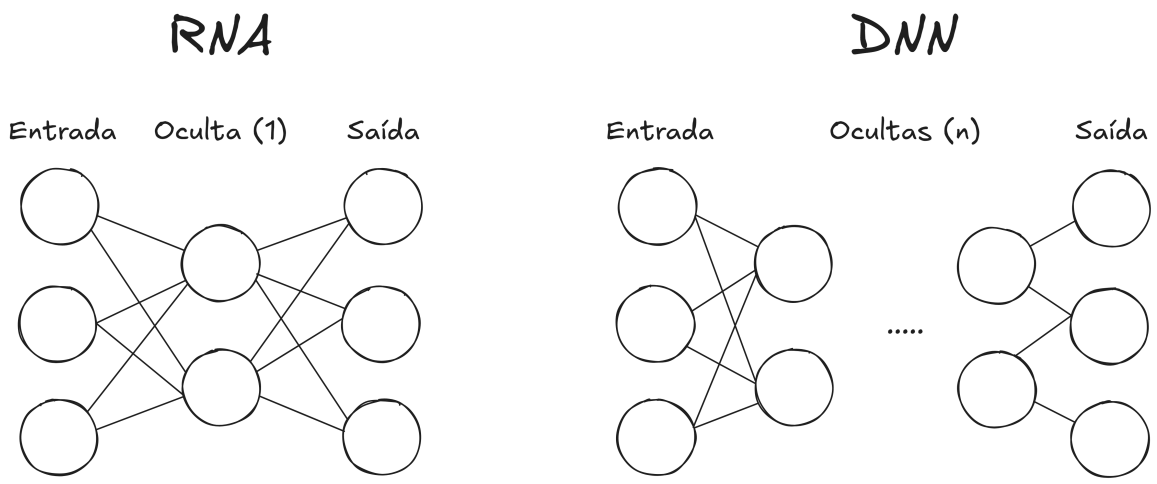

Figura 2 – Comparação de camadas em RNAs e DNNs.

3.3 Q-learning e DQN

O algoritmo Q-Learning (WATKINS, 1989) é um dos principais algoritmos de RL baseados em valor. Ele se resume a aprender a função Q-valor, denotada como $Q(s, a)$, que representa a recompensa esperada por um agente ao realizar uma ação a em um dado estado s. Se o agente conseguir aprender a função Q ótima, denotada por $Q^*(s, a)$, significa que o algoritmo descobriu uma política ótima π^*.

A política ótima $Q^*(s, a)$, adotada pelo algoritmo Q-Learning, é governada pela Equação de Otimalidade de Bellman, ou seja, essa equação é o alvo que o algoritmo de Q-Learning tenta alcançar. Ela afirma que, em uma política ótima, o valor de tomar uma ação a no estado s é igual à recompensa imediata r, somada à recompensa máxima esperada pela próxima ação a' no estado futuro s', descontada pelo fator γ (gamma).

$$Q^*(s, a) = r + \gamma \max_{a'} Q^*(s', a') \tag{3.2}$$

Na sua implementação fundamental (SUTTON; BARTO, 2018), o Q-learning utiliza uma matriz como base de dados, chamada Tabela-Q, onde as colunas representam os estados possíveis e as linhas, todas as ações possíveis. Cada célula, representada pelo par (s, a), é inicializada com 0 e seus valores são atualizados iterativamente. Cada par representa o valor retornado pela função $Q(s, a)$. Esses valores são atualizados utilizando o Aprendizado por Diferença Temporal (TDL), método que atualiza as estimativas da Tabela-Q com base em outras estimativas aprendidas (Bootstrapping), utilizando uma taxa de aprendizado α que varia de 0 a 1 para a seguinte regra:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)] \tag{3.3}$$

O uso deste algoritmo com a Tabela-Q é viável para problemas cujos espaços de estados e ações são discretos. No contexto de jogos, uma implementação clássica é o jogo da velha. Neste jogo, existem 9 quadrados, e cada um pode assumir um de três valores: X, O ou vazio. Isso resulta em um total teórico de $3^9 = 19683$ estados. Porém o jogo tem algumas restrições — por exemplo, um jogo com 9 quadrados X é impossível — , por isso esse total de estados pode ser reduzido para 5478 (GAOL; MUNIR, 2021). Para cada estado, existem até 9 ações possíveis, representando a escolha de marcar com X ou O algum dos quadrados. Portanto, a Tabela-Q possui um total de 5478 estados x 9 ações, resultando em uma matriz de $5478x9 = 49032$ células.

Definida a estrutura de dados e a regra de atualização, é necessário estabelecer um método efetivo para a escolha das ações do agente. Um dos principais desafios do Q-learning é o dilema entre Investigação e Exploração (Exploration vs. Exploitation), ou seja, escolher sempre a melhor ação possível com base no conhecimento atual ou escolher uma possível solução subótima para adquirir mais informação. Para resolver esse dilema, foi proposta a estratégia Epsilon-Guloso (ϵ-Greedy) (WATKINS, 1989), onde se define um valor ϵ entre 0 e 1, e então o agente escolhe a próxima ação com probabilidade ϵ de ser aleatória. Para garantir que o agente explore muito no começo e diminua com o tempo, é comum que o Decaimento-Epsilon (ϵ-Decay) seja utilizado, onde ϵ começa em um valor alto (próximo de 1) e é gradualmente reduzido com o avanço do treinamento.

O Q-Learning foi um marco no aprendizado por reforço, visto que conseguia treinar agentes para aprenderem políticas eficazes em ambientes desconhecidos. Porém, o uso desse algoritmo era muito limitado, já que só funcionava para problemas de baixa dimensionalidade (WATKINS; DAYAN, 1992). Isso ocorre pois, em problemas de alta dimensão, torna-se inviável usar uma Tabela-Q. Para exemplificar, suponha o seguinte jogo:

- **Resolução**: Uma resolução pequena de 50x50 pixels, resultando em um total de 2.500 pixels;
- **Cores**: Uma única camada de cor, utilizando uma escala de cinza em que a intensidade varia entre 0 e 255 (2^8);
- **Ações**: Um total de 6 ações, duas para se movimentar em cada um dos eixos X e Y, e duas ações auxiliares.

Nesse exemplo simples, teríamos um total de 256^{2500} estados e 6 ações possíveis, resultando em uma Q-table de tamanho $(256^{2500}) \times 6$, o que é impraticável computacionalmente. Foi com o avanço das redes neurais profundas que surgiu o algoritmo DQN, que usa redes neurais para se aproximar da função de valor Q, permitindo o aprendizado em ambientes de grande dimensionalidade, como videojogos.

O Deep Q-Network (DQN) (MNIH et al., 2013) é um algoritmo de aprendizado por reforço que combina os conceitos do Q-Learning e das Redes Neurais Profundas para resolver problemas de alta dimensionalidade. O objetivo desse algoritmo é descobrir a função $Q(s, a)$, que resulta no valor da recompensa futura ao realizar uma ação a em um estado s. Em vez de armazenar um valor para cada par (s,a), a rede neural profunda ajusta seus pesos θ a cada iteração para estimar o valor da função $Q(s, a; \theta)$.

Porém, com a substituição da Tabela-Q por uma DNN, uma nova instabilidade no treinamento surgiu. Isso se deve à combinação de três fatores: a rede neural sendo utilizada como um aproximador de função não-linear; o Bootstrapping, onde a rede tenta aprender com um alvo que ela mesma gera constantemente; e a correlação dos dados, já que normalmente as experiências são sequenciais e muito similares. Diante disso, duas técnicas essenciais foram implementadas para resolver esse problema:

1. **Replay de Experiência (Experience Replay)**: Em vez de treinar a rede apenas com os dados mais recentes, as experiências são armazenadas em um buffer de tamanho fixo (Replay Buffer) que utiliza uma estrutura FIFO (First-In, First-Out). Então, o algoritmo seleciona de forma aleatória um pequeno lote (batch) das experiências armazenadas no buffer. Esse batch é usado para realizar a atualização da rede neural, garantindo que as experiências sejam recentes e não correlacionadas.

2. **Redes Alvo (Target Networks)**: Inicialmente, a mesma rede neural era usada tanto para calcular a previsão atual $Q(s, a; \theta)$ quanto para o valor alvo $r+\gamma \max_{a'} Q(s', a'; \theta)$, o que causa instabilidade, já que a cada alteração dos pesos θ, ambas as funções precisam ser recalculadas. Para resolver isso, cria-se a rede-alvo, uma cópia da rede principal que utiliza pesos θ', os quais não são atualizados com tanta frequência quanto os pesos θ. Assim, o valor alvo é calculado utilizando como base os pesos da rede alvo:

$$y = r + \gamma \max_{a'} Q(s', a'; \theta^-) \tag{3.4}$$

Como os pesos θ^- permanecem constantes por um longo período, consequentemente o valor de y também permanece estável. Periodicamente, os pesos θ da rede principal são copiados para a rede alvo ($\theta^- \leftarrow \theta$), o que reduz drasticamente a instabilidade do treinamento. Juntos, o Replay de Experiência e a Rede Alvo tornam possível a utilização do Q-learning com redes neurais profundas, quebrando as limitações do Q-learning a problemas de baixa dimensionalidade (MNIH et al., 2015).

3.3.1 Dueling e Double DQN

Desde a introdução do algoritmo DQN por Mnih et al. (MNIH et al., 2013), algumas variações surgiram, sendo as principais o Dueling DQN e o Double DQN. O algoritmo Double Deep Q-Network (DDQN) (HASSELT; GUEZ; SILVER, 2016) busca resolver um dos principais problemas do DQN padrão: o viés de superestimação (maximization bias). O operador de maximização (max) na equação de atualização da rede alvo no DQN tende a selecionar valores que podem ter sido superestimados devido a erros de aproximação ou ruídos no ambiente. Para resolver esse problema, o DDQN propôs dividir o trabalho entre as duas redes: a seleção da melhor ação futura a' no estado s' é feita pela rede principal, porém a rede alvo é quem calcula o valor previsto dessa ação futura e avalia a escolha.

Já o Dueling DQN (WANG et al., 2016)busca separar a função Q-valor em duas outras funções: a função de valor do estado atual $V(s)$ e a função de vantagem da ação $A(s, a)$. A relação entre essas três funções é dada da seguinte forma:

$$Q(s, a) = V(s) + A(s, a) \tag{3.5}$$

- **State-Value Function, V(s)**: Representa a qualidade de um estado s, calculada a partir do valor médio de todas as ações possíveis naquele estado.
- **Action-Advantage Function, A(s,a)**: Vantagem de escolher uma ação a em um estado s. Esse valor é estimado pela rede neural, não sendo calculado diretamente.

Isso serve para que a rede neural profunda determine se as diferentes ações em um estado são significativas ou não, melhorando a eficiência do treino. Porém, essa fórmula sofre de um grave problema de identificabilidade, já que, dado um valor Q, a rede neural não consegue recuperar os valores de V e A quando for atualizar seus pesos. Para resolver esse problema, o artigo original propõe a seguinte fórmula, considerando $\mathcal{A}$ como o conjunto de todas ações possíveis:

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}} A(s,a') \right) \tag{3.6}$$

Esse cálculo permite à rede neural descobrir o valor de $V(s)$ apenas conhecendo os valores $Q(s,a)$, já que ao calcularmos o Q-médio ($\bar{Q}$), ele sempre será igual a $V(s)$. Isso pode ser comprovado manipulando a equação:

$$\bar{Q}(s,a) = \frac{1}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} Q(s,a) \tag{3.7}$$

Substituímos a fórmula do Dueling DQN para $Q(s,a)$ dentro do somatório:

$$\bar{Q}(s,a) = \frac{1}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} \left[V(s) + \left(A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}} A(s,a') \right) \right] \tag{3.8}$$

Podemos separar o somatório:

$$\bar{Q}(s,a) = \frac{1}{|\mathcal{A}|} \left[\sum_{a \in \mathcal{A}} V(s) + \sum_{a \in \mathcal{A}} A(s,a) - \sum_{a \in \mathcal{A}} \left(\frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}} A(s,a') \right) \right] \tag{3.9}$$

Como $V(s)$ e a média de $A(s,a')$ são constantes em relação à somatória sobre a, somá-los $|\mathcal{A}|$ vezes é o mesmo que multiplicar por $|\mathcal{A}|$:

$$\bar{Q}(s,a) = \frac{1}{|\mathcal{A}|} \left[|\mathcal{A}| V(s) + \sum_{a \in \mathcal{A}} A(s,a) - |\mathcal{A}| \left(\frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}} A(s,a') \right) \right] \tag{3.10}$$

O termo $|\mathcal{A}|$ se cancela na última parte:

$$\bar{Q}(s,a) = \frac{1}{|\mathcal{A}|} \left[|\mathcal{A}| V(s) + \sum_{a \in \mathcal{A}} A(s,a) - \sum_{a' \in \mathcal{A}} A(s,a') \right] \tag{3.11}$$

A soma de todas as vantagens ($\sum A(s,a)$) subtraída de si mesma é zero:

$$\bar{Q}(s,a) = \frac{1}{|\mathcal{A}|} \left[|\mathcal{A}| V(s) + 0 \right] \tag{3.12}$$

Finalmente, simplificamos para obter o resultado final:

$$\bar{Q}(s,a) = V(s) \tag{3.13}$$

Na prática, isso permite que o algoritmo Dueling DQN aprenda os valores $A(s, a)$ e $V(s)$ sem gastar armazenamento adicional, ancorando as estimativas V e A e eliminando a ambiguidade. A principal vantagem desse processo é que a rede neural aprende o valor estimado de cada estado sem depender do valor de cada ação individual, podendo avaliar se as escolhas das ações são irrelevantes ou não em um determinado estado s.

O trabalho de (FU, 2024) compara o aprendizado do agente Dueling DQN com o DQN padrão, utilizando como base o ambiente do jogo Super Mario Bros. Ambos foram treinados ao longo de 2.000 épocas com diferentes hiperparâmetros, e foram comparadas as pontuações médias das 100 últimas épocas de cada algoritmo. Ambos os agentes indicaram uma capacidade de aprendizado, mas o Dueling DQN obteve uma taxa de crescimento mais alta e constante, indicando um desempenho potencialmente melhor. Por isso, normalmente não se opta por utilizar Double DQN ou Dueling DQN isoladamente, mas sim as duas técnicas juntas, já que se complementam e servem para resolver problemas distintos do algoritmo DQN original. Na maioria das vezes, essa combinação é chamada de Dueling DQN com otimização Double DQN, explorada com mais profundidade em (SEWAK, 2019). A Figura 3 representa um diagrama do funcionamento geral do Dueling DQN, bem como seus principais componentes.

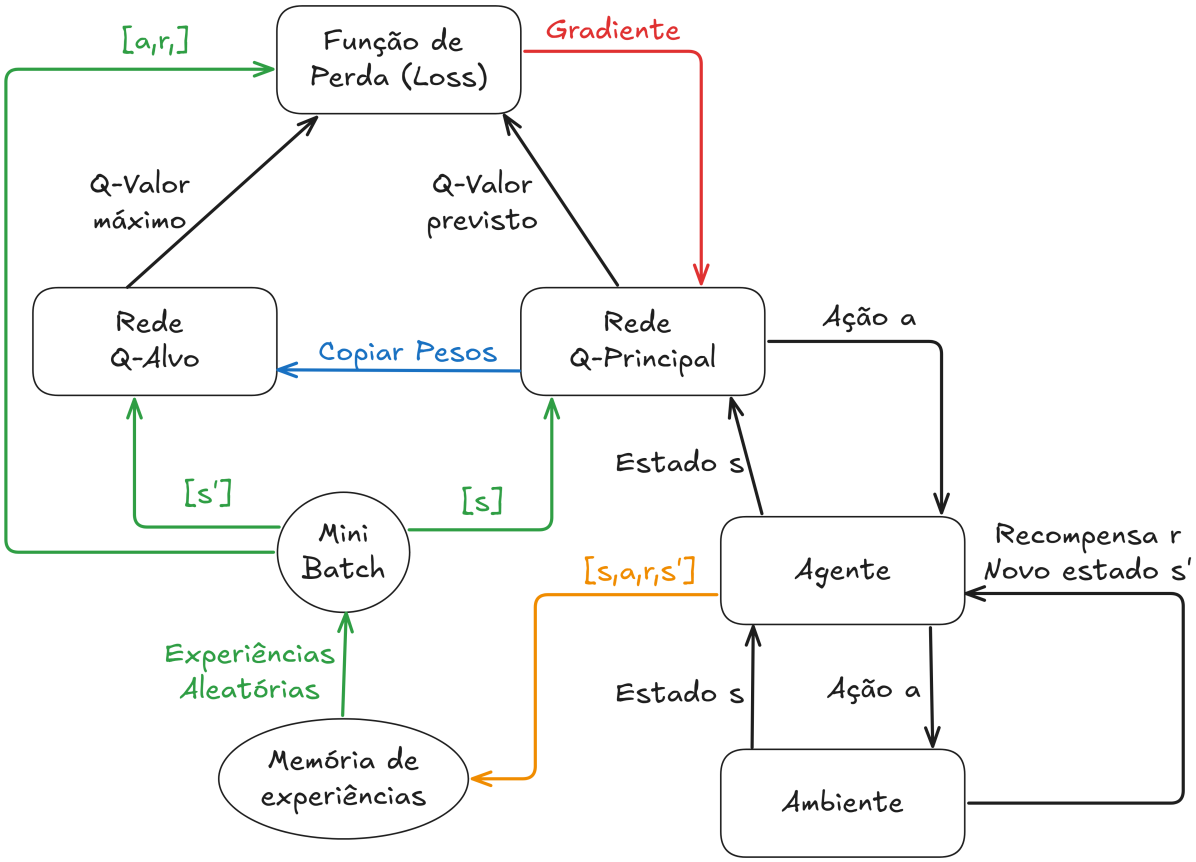

Figura 3 – Funcionamento geral do Dueling DQN.

O Dueling DQN define o ponto de parada necessário para a fundamentação teórica deste trabalho. Para a implementação dos testes, faz-se necessário utilizar um conjunto de ferramentas e bibliotecas, que são apresentadas a seguir.

3.4 PyTorch

O PyTorch (PyTorch Foundation, 2023) é uma biblioteca de Python desenvolvida pela Meta AI com foco em aprendizado de máquina, fornecendo interfaces simples e eficientes para a criação e o treinamento de redes neurais artificiais. Sua principal estrutura de dados são os Tensores, arrays multidimensionais similares aos do NumPy, mas com suporte a computação paralela em Unidades de Processamento Gráfico (GPUs). O PyTorch também possui um módulo específico para redes neurais chamado 'nn' (torch.nn), que oferece um conjunto de blocos de construção, contendo várias camadas e funções de ativação, permitindo a construção de modelos complexos.

Lançado em 2016 e criado pela Meta (anteriormente Facebook), o PyTorch surgiu como um concorrente ao TensorFlow. Sua adoção cresceu rapidamente devido à facilidade de uso e integração com outras bibliotecas, como NumPy e SciPy. Em 2019, o PyTorch 1.0 foi lançado, apresentando várias melhorias de otimização e usabilidade, tornando-se uma alternativa viável ao TensorFlow.

Recentemente, o PyTorch 2.0 foi lançado junto de sua nova funcionalidade, o TorchDynamo, um compilador em nível de Python que torna o código até 2 vezes mais rápido, além de melhorias significativas no desempenho do treinamento. Adicionalmente, nos últimos anos, o PyTorch tem recebido várias melhorias de integração com GPUs, na estrutura de código e na inferência em principais plataformas de nuvem (como Google Cloud e Azure).

Existem outras alternativas ao PyTorch, sendo as principais o TensorFlow e o JAX, ambas da Google. O TensorFlow é amplamente utilizado em produção, devido à sua alta escalabilidade e ótimo sistema de deployment, enquanto o JAX é conhecido por seu alto desempenho e por ser amplamente utilizado em computação de alta performance. Já o PyTorch é muito utilizado em ambientes de pesquisa e estudo, devido à facilidade de uso e reprodução, além de ter uma vasta comunidade com extensas bases de código pré-existentes (NOVAC et al., 2024).

Alguns trabalhos já mostraram a eficiência do PyTorch, como (PASZKE et al., 2019), que explica como o PyTorch é fácil de usar e depurar, além de ser eficiente, principalmente quando usado com GPUs. Esse artigo também utiliza testes de benchmark para avaliar a eficiência de subsistemas individuais do PyTorch, bem como a eficiência geral da biblioteca.

3.5 Biblioteca Gym

O OpenAI Gym (agora Gymnasium) é uma biblioteca de Python amplamente utilizada para o treinamento de agentes através do aprendizado por reforço, devido à sua facilidade de uso e grande quantidade de ambientes de treinamento. A biblioteca fornece um conjunto de ambientes padronizados, permitindo treinar e avaliar diferentes algoritmos para diferentes cenários de forma simples e uniforme. O Gym surgiu como uma solução para facilitar o treinamento por reforço, visto que antes a maioria dos datasets não era padronizada, o que tornava cada implementação de algoritmo única e dificultava a comparação entre diferentes abordagens para o mesmo problema.

Desde seu lançamento, foram criadas diferentes variações e versões do Gym para variados datasets, buscando atender a necessidades específicas. Um dos mais recentes é o Gym Super Mario Bros (Christian Kauten, 2018), um ambiente do OpenAI Gym para os jogos Super Mario Bros, e Super Mario Bros 2 do Nintendo Entertainment System (NES), usando o emulador nes-py. Alguns artigos já mostraram a utilidade do Gym, como o trabalho de (BROCKMAN et al., 2016), que o indica como uma ótima ferramenta para o compartilhamento e a comparação do desempenho de diferentes algoritmos, além de permitir testes de benchmark padronizados.

Baixar e utilizar o Gym é relativamente simples: basta ter uma versão compatível do Python instalada (versões 3.7-3.13) e utilizar o comando `pip install gymnasium` para instalar a versão mais recente. Após baixar os arquivos necessários, para criar e inicializar um ambiente, basta utilizar os seguintes comandos:

```
# Crie um ambiente virtual chamado 'gym-env' no diretório do seu projeto
python -m venv gym-env

# Ative o ambiente virtual criado:

# No Linux ou macOS:
source gym-env/bin/activate

# No Windows (PowerShell):
.\gym-env\Scripts\activate
```

Agora que o básico do Gym foi instalado, o ambiente criado terá apenas alguns cenários básicos, como o CartPole, para evitar a instalação de pacotes desnecessários. Os ambientes mais complexos são organizados em famílias. Para instalar uma família específica, como jogos de Atari, basta executar o seguinte comando:

```
pip install "gymnasium[atari]"
```

Também é possível instalar todos os ambientes de uma só vez, porém isso pode ser demorado devido ao grande número de pacotes. Para tal, deve-se executar o seguinte comando:

```
pip install "gymnasium[all]"
```

Com a instalação completa, já é possível importar os ambientes instalados para o código Python e utilizá-los conforme o necessário. O processo de utilização e o funcionamento completo do Gym são relativamente extensos, por isso a leitura do trabalho (BROCKMAN et al., 2016) é recomendada para uma melhor compreensão da ferramenta.

4 Metodologia de desenvolvimento e pesquisa

Antes de realizar os experimentos, foi preciso primeiramente modelar o código para identificar as principais etapas necessárias para treinar um agente para um problema específico e, a partir disso, definir quais dessas etapas poderiam ser generalizadas e reutilizadas para qualquer outro exemplo.

4.1 Funcionamento geral do código

Para a realização dos experimentos, foi utilizada a linguagem de programação Python, devido à sua ampla utilização na área de inteligência artificial e à grande disponibilidade de bibliotecas úteis para o projeto. Foram testados 3 jogos ao todo; no entanto, para fins de exemplificação e normatização, será considerado apenas o código do jogo Super Mario Bros. Onde for necessário, serão indicadas as mudanças para implementar um jogo diferente. A Figura 4 apresenta as principais etapas do aprendizado de um agente, cada uma descrita nos subcapítulos a seguir:

4.1.1 A - Ambiente do Jogo

O ambiente de treinamento foi construído utilizando a biblioteca Gym. No primeiro treinamento, utilizou-se especificamente a Gym-Super-Mario-Bros, que permite a simulação do jogo em uma interface compatível com algoritmos de aprendizado por reforço. Esse pacote também possui um conjunto de funções prontas que facilitam o retorno das recompensas pelo ambiente, permitindo uma implementação mais eficiente. Como o Gym-Super-Mario-Bros funciona apenas para o jogo Super Mario, seria necessário buscar outra biblioteca para implementar o ambiente de outro jogo, como o Gym-Retro ou o Gym-Atari.

Juntamente com o ambiente Gym, também são inicializados o espaço de ação e o espaço de observação. O espaço de ação representa o conjunto de ações que o agente pode realizar; no exemplo do jogo Mario, esse conjunto já é dado, mas em outros ambientes o conjunto de ações não necessariamente estará completamente definido. O espaço de observação representa as informações que o agente recebe do ambiente, normalmente frames da tela do jogo no estado atual, mas esses frames passarão por um tratamento nos Wrappers de ambiente antes de serem enviados para a rede neural.

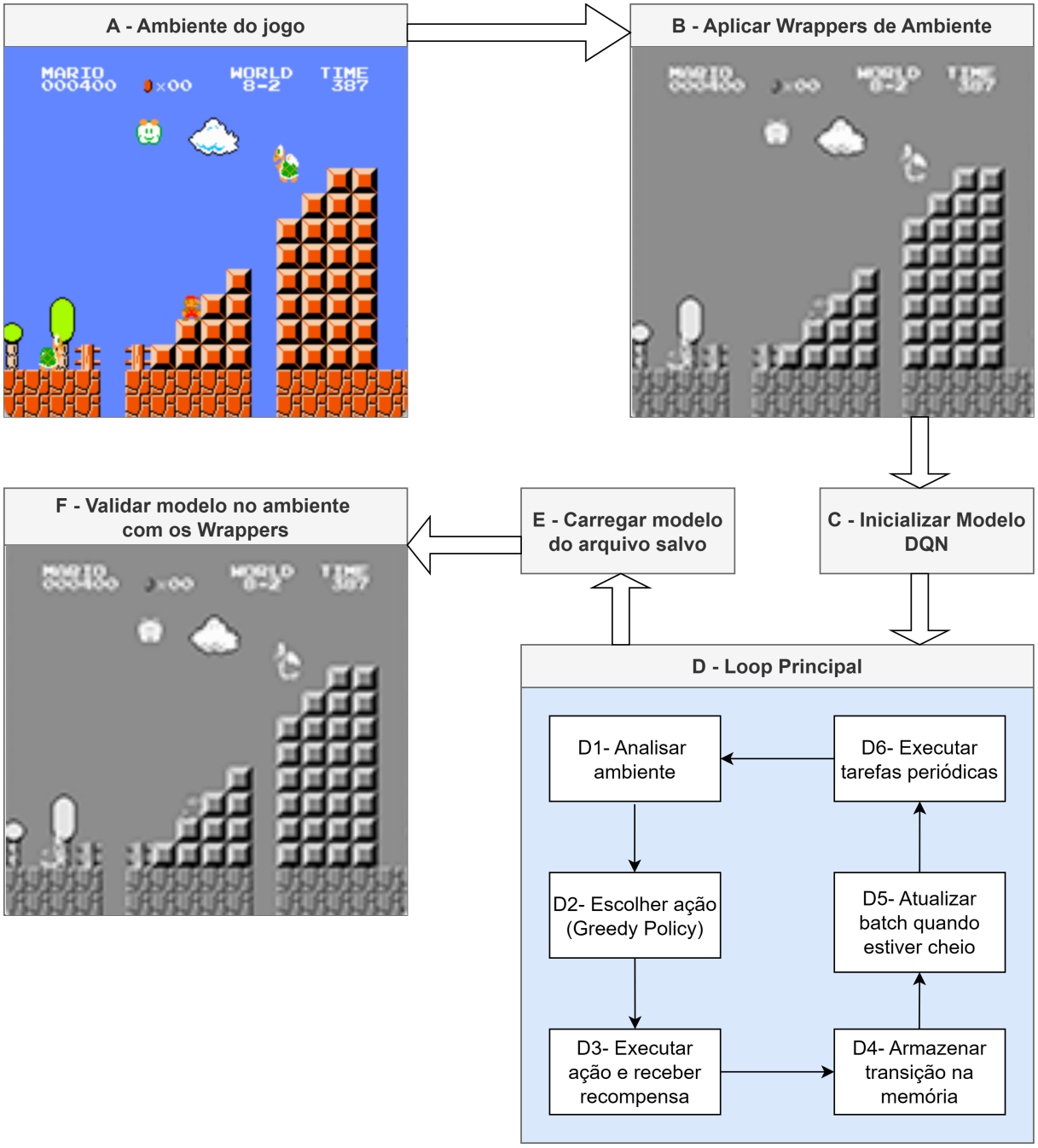

Figura 4 – Diagrama representando a execução do código.

Neste trabalho, optou-se apenas por utilizar ambientes de jogos 2D (duas dimensões). Contudo, é possível utilizar o DQN em ambientes 3D, mas o processo de transição é relativamente complexo e por isso não será abordado. No entanto, trabalhos como o de (CHAPLOT et al., 2016) ensinam com detalhes como implementar e treinar um agente DQN dentro de ambientes 3D.

4.1.2 B - Wrappers de Ambiente

Para acelerar o treinamento e otimizar o código, foram implementados Wrappers no ambiente, os quais redimensionam a tela para uma resolução menor, convertem a imagem para uma escala de cinza e pulam uma pequena porcentagem dos frames. Além disso, os Wrappers também servem para normalizar e ajustar alguns valores retornados pelo ambiente, sendo o principal o valor da recompensa.

Os Wrappers podem ser aplicados em diversos tipos de ambiente, mas precisam ser ligeiramente adaptados para atender às necessidades específicas de cada um. A maior parte dos wrappers é uma adaptação dos códigos disponibilizados pela OpenAI Baselines, que foram feitos para se integrar com o ambiente do Gym. Seguem os principais Wrappers genéricos implementados em cada um dos experimentos, em sua respectiva ordem de ativação:

- **GameActions()**: Alguns ambientes possuem vários botões, permitindo centenas de combinações. Entretanto, na maioria dos jogos, só algumas dessas combinações são relevantes, então torna-se necessário criar uma lista apenas com as combinações necessárias.
- **NoopReset(noop_max)**: Após o reset do ambiente, o personagem não realiza nenhuma operação (noop) por um número aleatório de vezes, variando entre 1 e ‘noop_max‘.
- **MaxAndSkip(n)**: Serve para acelerar o treinamento. Em vez de o agente decidir sua ação a cada frame, a ação escolhida em um frame x é repetida pelos próximos $n - 1$ frames. Suponha $n = 4$ e que o jogo esteja funcionando a 60 quadros por segundo (FPS); então, a cada segundo, o agente terá que escolher $60/4 = 15$ ações.
- **GameReward()**: Alguns ambientes não possuem sistema de recompensa próprio, então torna-se necessário criar uma função que calcula e retorna a recompensa para o agente.
- **WarpFrame(size)**: Converte a escala de três canais de cores (RGB) para um canal (escala de cinza) e redimensiona o ambiente para uma quantidade de pixels igual a ‘size‘ x ‘size‘. Isso é feito para diminuir consideravelmente a quantidade de informação que a rede neural recebe e precisa processar.
- **ScaledFloat()**: Para cada pixel da imagem retornada pelo WarpFrame, transforma o intervalo de 0 a 255 da escala de cinza para um intervalo de 0.0 a 1.0, dividindo cada valor por 255. Isso é feito pois as redes neurais funcionam de forma muito mais estável quando recebem valores normalizados.

- **FrameStack(k)**: Pega os últimos k frames processados e os empilha antes de passar para a rede neural, ou seja, retorna um tensor de tamanho ‘size‘ x ‘size‘ x ‘k‘. Isso é feito para não passar uma única imagem estática para a rede, permitindo que o agente tenha uma noção de movimento, direção e velocidade de cada elemento na tela.

4.1.3 C - Inicialização do Modelo DQN

Nessa etapa, foi instanciada a rede neural DQN que será usada ao longo do código. Para isso, foi necessário definir os parâmetros iniciais do modelo, além da quantidade de neurônios presentes nas camadas ocultas e de saída. Após isso, foram instanciados os hiperparâmetros usados durante o treinamento, como taxa de aprendizado, tamanho de memória, fator de desconto (Gamma), entre outros. Por fim, o ambiente é retornado ao estado inicial, etapa feita para garantir que não haja nenhuma alteração antes do início do treinamento.

É importante notar que, como cada jogo possui um ambiente e um conjunto de ações diferentes, os hiperparâmetros também precisam ser alterados para que se adéquem melhor ao ambiente específico utilizado. A arquitetura escolhida, bem como a estrutura do código de treinamento do agente DQN, foram feitas utilizando como base as principais implementações de Dueling DQN com otimização Double DQN presentes na OpenAI Baselines. Na Tabela 1, está representada a arquitetura utilizada no modelo.

Tabela 1 – Arquitetura da Rede Neural Dueling DQN

Camada	**Tipo**	**Formato da Saída**	**Kernel/Stride**	**Ativação**
Input	Imagem	(4, 84, 84)	-	-
Conv1	Convolucional	(32, 20, 20)	8x8 / 4	ReLU
Conv2	Convolucional	(64, 18, 18)	3x3 / 1	ReLU
FC1	Densa	512	-	ReLU
Fluxos da Dueling DQN				
Value Stream	Densa	1	-	Linear
Advantage Stream	Densa	12 (Número de Ações)	-	Linear

As duas primeiras camadas da rede são convolucionais (Conv1 e Conv2). A função dessas camadas é extrair informações visuais da imagem do jogo, como a posição do personagem e dos inimigos na tela. A camada linear (FC1) recebe as informações visuais fornecidas pelas camadas convolucionais e as transforma em um vetor de características, o que facilita a tomada de decisão do agente.

Então, a rede neural é dividida em dois fluxos, o Fluxo de Valor (Value Stream) e o Fluxo de Vantagem (Advantage Stream), ambos recebendo a mesma entrada: o vetor de 512 valores fornecido pela camada linear. O Fluxo de Valor é responsável por calcular

o valor do estado atual do agente, independentemente da ação, ou seja, o valor $V(s)$. O Fluxo de Vantagem serve para estimar a vantagem de cada ação individualmente em um dado estado s, ou seja, estimar o valor $A(s, a)$ para cada ação. A partir disso, uma operação matemática é realizada para calcular o valor $Q(s, a)$ para cada ação a no estado s, e a ação com o maior Q-value será escolhida.

4.1.4 D - Loop Principal

O treinamento ocorre em um loop iterativo utilizando episódios, onde cada episódio representa uma única tentativa do agente de terminar a fase. A cada iteração, ocorrem as seguintes etapas, respectivamente:

4.1.4.1 D1 - Analisar Ambiente

O ambiente fornece para o agente o estado atual, que já passou por todos os wrappers de pré-processamento, para definir se o agente está em um estado final ou se é necessário escolher a próxima ação.

4.1.4.2 D2 - Escolher Ação

Após analisar o ambiente, uma ação é escolhida, com uma probabilidade ϵ de ser aleatória e $1 - \epsilon$ de ser uma ação escolhida pelo agente DQN. No caso do Mario e de algumas implementações, ϵ não é um valor fixo, mas sim um valor que começa próximo de 1 e decai ao longo das épocas a uma taxa fixa definida pelo ϵ-Decay, até chegar a um valor próximo de zero chamado ϵ-Min. Caso a ação aleatória não seja escolhida, o modelo DQN verifica todas as ações possíveis e escolhe aquela que gerará a melhor recompensa futura esperada, já que a política é determinística. Também é feito o cálculo do valor esperado da posição atual no ambiente, para definir se o agente está em um estado onde as ações têm grande importância ou não, o que é feito para aumentar a exploração em casos onde não existe uma ação com um valor muito maior que as outras.

É importante notar que o conjunto de ações a ser escolhido é feito com base no espaço de ação do ambiente, ou seja, sempre que o ambiente for alterado, o conjunto de ações que a rede neural recebe também mudará.

4.1.4.3 D3 - Executar Ação e Receber Recompensa

Ao escolher a ação que gerará a melhor recompensa esperada, o personagem dentro do jogo realizará essa ação, que em seguida é enviada para o ambiente através da função `env.step()`. Então, o ambiente retorna a recompensa (reward), o próximo estado (next-state) e um sinal de conclusão (done). Esse sinal é necessário para identificar se o ambiente chegou a um estado final ou não, por exemplo, se o agente morreu.

4.1.4.4 D4 - Armazenar Transição

Em seguida, a transição completa (state, action, reward, next-state, done) é armazenada em uma memória temporária, que guarda as experiências passadas mais recentes do agente. O código só inicia o processo de aprendizado quando a memória está suficientemente preenchida, o que é garantido pela condição `if len(agent.memory) > BATCH-SIZE:`.

4.1.4.5 D5 - Atualização do Lote (Batch)

No momento em que a memória temporária atinge um determinado tamanho, o algoritmo atualiza a rede neural do modelo DQN utilizando um pequeno lote (batch) de experiências, obtido a partir de transições aleatórias escolhidas da memória. Essa atualização ocorre a cada certo número de passos (steps) no ambiente. As transições permanecem na memória, e apenas o batch é atualizado com novas transições aleatórias, para garantir maior estabilidade no treinamento. A atualização é feita com base na perda (loss), parâmetro obtido a partir da diferença entre o valor Q-previsto pela rede neural principal (Q-net) e o valor Q-alvo (Q-target). A fórmula está descrita a seguir:

$$L(\theta) = \mathbb{E}\left[(y_i - Q(s, a; \theta, \alpha, \beta))^2\right]$$

onde,

$$y_i = r + \gamma \max_{a'} Q(s', a'; \theta^-)$$

e

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \left(A(s, a; \theta, \alpha) - \frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}} A(s, a'; \theta, \alpha)\right)$$

- $L(\theta)$: A função de perda (Loss) que o algoritmo tenta minimizar.
- θ: Os parâmetros (pesos e vieses) das camadas da rede neural.
- α: Os parâmetros utilizados no cálculo da Vantagem (Advantage stream).
- β: Os parâmetros utilizados no cálculo do Valor do estado (Value stream).
- $\mathbb{E}[\cdot]$: O valor médio esperado. Na prática, é a média do erro sobre um lote (batch) de experiências.
- y_i: O Q-alvo (target), calculado usando a equação de Bellman.
- $Q(s, a; \theta, \alpha, \beta)$: O Q-valor previsto pela rede para o par estado-ação (s, a).
- s: O estado (state) atual do ambiente.

- a: A ação (action) executada no estado s.
- r: A recompensa (reward) recebida após executar a ação a no estado s.
- γ: O fator de desconto (gamma), valor entre 0 e 1 que pondera a importância de recompensas futuras.
- s': O próximo estado do ambiente, após a ação a ser executada no estado s.
- a': Uma ação possível no próximo estado s'.
- θ^-: Os parâmetros da rede alvo (target network), uma cópia congelada dos parâmetros da rede principal.
- $V(s;\theta,\beta)$: A predição da função de Valor para o estado s.
- $A(s,a;\theta,\alpha)$: A predição da função de Vantagem (Advantage) para a ação a no estado s, que mede a vantagem da ação a em comparação com as outras ações possíveis.
- $\mathcal{A}$: O conjunto de todas as ações possíveis no ambiente.
- $|\mathcal{A}|$: O número total de ações possíveis, cardinalidade do conjunto $\mathcal{A}$.

4.1.4.6 D6 - Tarefas Periódicas

Por fim, a cada número periódico de épocas, o algoritmo realiza duas tarefas distintas que ajudam a estabilizar o treinamento.

Primeiro, o algoritmo armazena o modelo dentro de um arquivo a cada número de épocas definido por `save-interval`. Esse arquivo é salvo como um checkpoint (.pth). O salvamento da rede principal é especialmente importante, pois torna possível retomar o treinamento quando necessário, sem grandes perdas no progresso.

Paralelamente, a atualização da rede alvo ocorre periodicamente, de acordo com o hiperparâmetro `update-interval`. A atualização ocorre ao copiar os pesos da rede neural principal para a rede alvo. É importante que essa atualização não seja frequente, pois é ela quem garante a estabilidade do treinamento ao fornecer uma rede alvo fixa e confiável.

4.1.5 E - Carregar Modelo do Arquivo

Após concluir o treinamento, o arquivo do agente gerado é salvo juntamente com seus pesos e a estrutura das camadas. A partir disso, é possível carregar em outro código o mesmo agente treinado, que pode ser usado para retomar o loop de treinamento ou ser carregado em um ambiente de validação sem a necessidade de mais treino.

4.1.6 F - Validar Modelo

Para avaliar o modelo em um código separado, o ambiente do jogo é carregado — o mesmo utilizado durante o treinamento — juntamente com os wrappers, visto que o agente foi treinado para jogar no ambiente modificado, e não no original. Em seguida, o modelo salvo em arquivo é inicializado junto com o ambiente para validar seu desempenho fora do loop de treinamento, até que eventualmente o modelo chegue a um estado final, e então o algoritmo mostrará qual foi a pontuação máxima alcançada.

No caso do jogo Super Mario Bros, a validação foi feita com base na pontuação e na fase máxima alcançada pelo agente, para que fosse possível observar seu desempenho após o final do treinamento. Já para outros exemplos, a validação deve ser feita com base na função de recompensa definida nos wrappers ou com base em parâmetros que sejam relevantes para o ambiente específico.

5 Experimentos e resultados

Para cada um dos experimentos a seguir, o ambiente de desenvolvimento utilizado foi o PyCharm, onde foram utilizados três arquivos de código: ‘wrappers.py‘, que trata do funcionamento do ambiente e seus wrappers; ‘duel_dqn.py‘, no qual a rede neural é instanciada e o treinamento propriamente dito ocorre; e ‘eval.py‘, que carrega o modelo treinado a partir de um arquivo e o avalia. Todos os experimentos foram feitos utilizando o mesmo computador, cujas especificações estão descritas na Tabela 2.

Tabela 2 – Especificações do ambiente computacional utilizado nos experimentos.

Componente	**Especificação**
Processador (CPU)	Intel®Core™i5-12400F, 2.50 GHz
Memória RAM	16 GB DDR4 (2x 8 GB) @ 2400MHz
Placa de Vídeo (GPU)	NVIDIA GeForce RTX 3060
Armazenamento Principal	HDD 1 TB Western Digital (5400 RPM)
Sistema Operacional	Microsoft Windows 10 Pro

Por motivos de padronização e normatização, os mesmos valores de hiperparâmetros foram utilizados em todos os experimentos para os seguintes Wrappers: `NoopReset(noopMax)`, `MaxAndSkip(frameSkip)`, `WarpFrame(size)` e `FrameStack(k)`. A Tabela 3 possui todos os valores utilizados no ambiente.

Tabela 3 – Hiperparâmetros de ambiente utilizados em todos os experimentos.

Hiperparâmetro	**Valor**
Ações Nulas Máximas (noopMax)	30
Pulo de Frames (frameSkip)	4
Redimensionamento de Imagem (size)	84x84 pixels
Empilhamento de Frames (k)	4 frames

As implementações completas para os ambientes Gym-Super-Mario-Bros e Gym-Retro, bem como as bibliotecas e versões necessárias para executá-las, podem ser encontradas nos respectivos repositórios (TIAGO, 2025b; TIAGO, 2025a).

5.1 Super Mario Bros - NES

Nesse primeiro experimento, o ambiente utilizado foi o do jogo Super Mario Bros., um jogo de plataforma de duas dimensões onde o principal objetivo é avançar do começo da fase até o final sem morrer, movimentando-se normalmente da esquerda para a direita. No meio do caminho, existem obstáculos como buracos e inimigos, que devem ser evitados,

além de moedas e itens especiais que podem ser utilizados para aumentar a pontuação ou fortalecer o personagem. O ambiente utilizado durante o treinamento, pode ser observado na Figura 5.

Figura 5 – Tela do ambiente de treinamento do jogo Super Mario

A biblioteca Gym-Super-Mario-Bros foi utilizada para carregar o ambiente do jogo na versão de NES. Essa biblioteca já possui muitas funcionalidades prontas que facilitaram o treinamento, sendo as principais um conjunto de ações definido, uma função de recompensa própria e um dicionário com informações detalhadas sobre o estado do jogo a cada passo. Na Tabela 4, seguem os hiperparâmetros de treinamento do Super Mario Bros.

Neste primeiro experimento, as dificuldades não foram muitas, visto que o Gym-Super-Mario-Bros já lida com muitos dos problemas de implementação e os hiperparâmetros recomendados são conhecidos. Por isso, a maior dificuldade foi encontrar meios para acelerar o treinamento, o que foi possível com uma melhora no funcionamento dos wrappers e a substituição de vetores base do Python por stacks do NumPy e tensores do PyTorch.

Para o treinamento propriamente dito, o agente foi treinado por um total de 10.000 épocas. Sempre que o agente morre ou ultrapassa o tempo limite, o ambiente é reiniciado

Tabela 4 – Hiperparâmetros utilizados no treinamento do jogo Mario.

Hiperparâmetro	**Valor**
Taxa de Aprendizagem (Learning Rate)	0.001
Tamanho do Lote (Batch Size)	64
Número de Épocas (Epochs)	10.000
Otimizador	AdamW
Capacidade da Memória	100.000
Epsilon Inicial (ϵ_{start})	0.85
Epsilon Final (ϵ_{end})	0.001
Decaimento do Epsilon	500
Fator de Desconto (γ)	0.99
Frequência de Atualização da Rede Alvo (por época)	50

e o agente volta para o começo da fase em uma nova época. Ao todo, este treinamento durou 27 horas. Durante esse processo, as seguintes métricas foram armazenadas a cada época: tempo de execução, fase alcançada, score (pontuação), average Loss (perda média) e total steps (ações totais). As métricas de perda média e pontuação podem ser observados na Figura 6, enquanto o nível alcançado e tempo de execução total são apresentados na Figura 7.

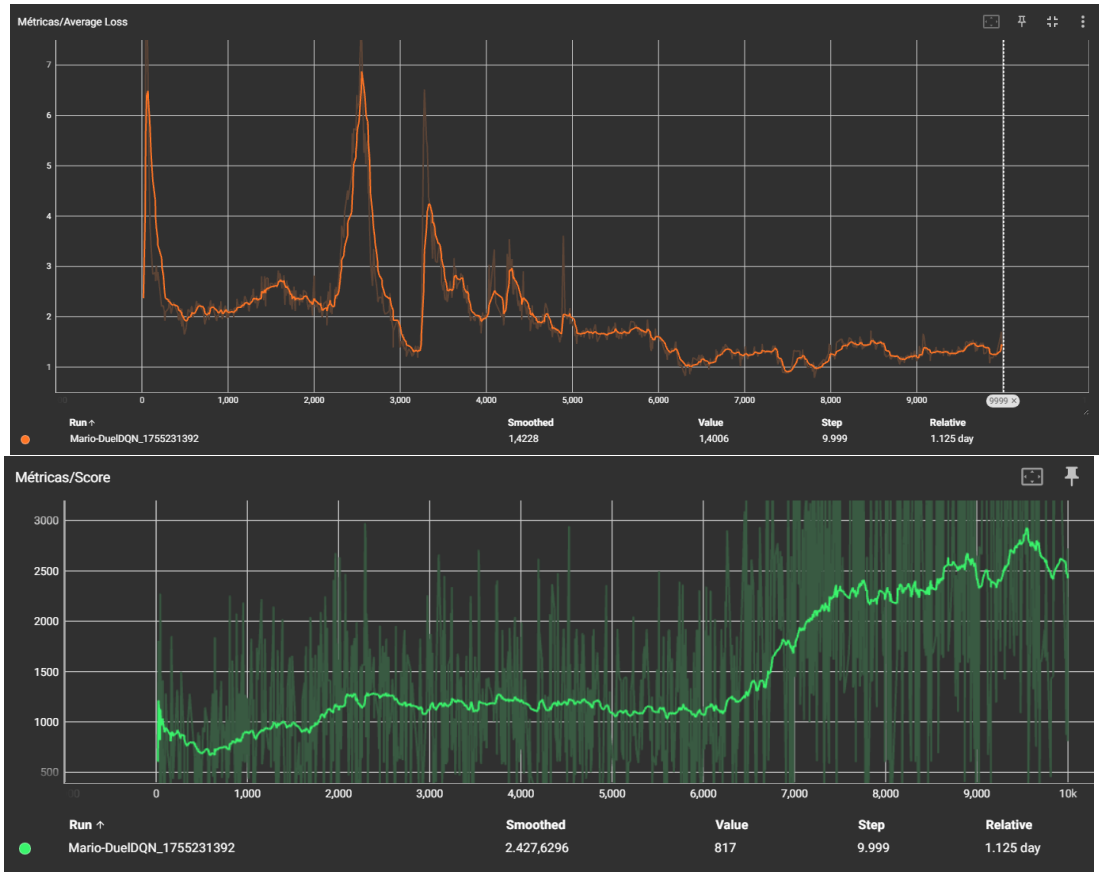

Figura 6 – Perda média (Average Loss) e Pontuação (Score) no ambiente Mario.

A partir dos gráficos da 6, é possível analisar alguns dados para justificar e comprovar que a rede neural está, de fato, aprendendo a melhorar no jogo ao longo das épocas. Os dois principais parâmetros a serem analisados são o Score, que deve aumentar, e o Average Loss, que tende a diminuir. Analisando apenas esses dois fatores, é possível notar que o comportamento dos gráficos é similar ao esperado em um bom treinamento. Outro ponto interessante é que o Score influencia o Loss: sempre que o Score aumenta abruptamente, o Loss também sobe, indicando que o agente encontrou um novo caminho ótimo. Assim, o Loss aumenta muito e vai se reduzindo conforme o agente aprende o valor da recompensa esperada ao seguir esse novo caminho.

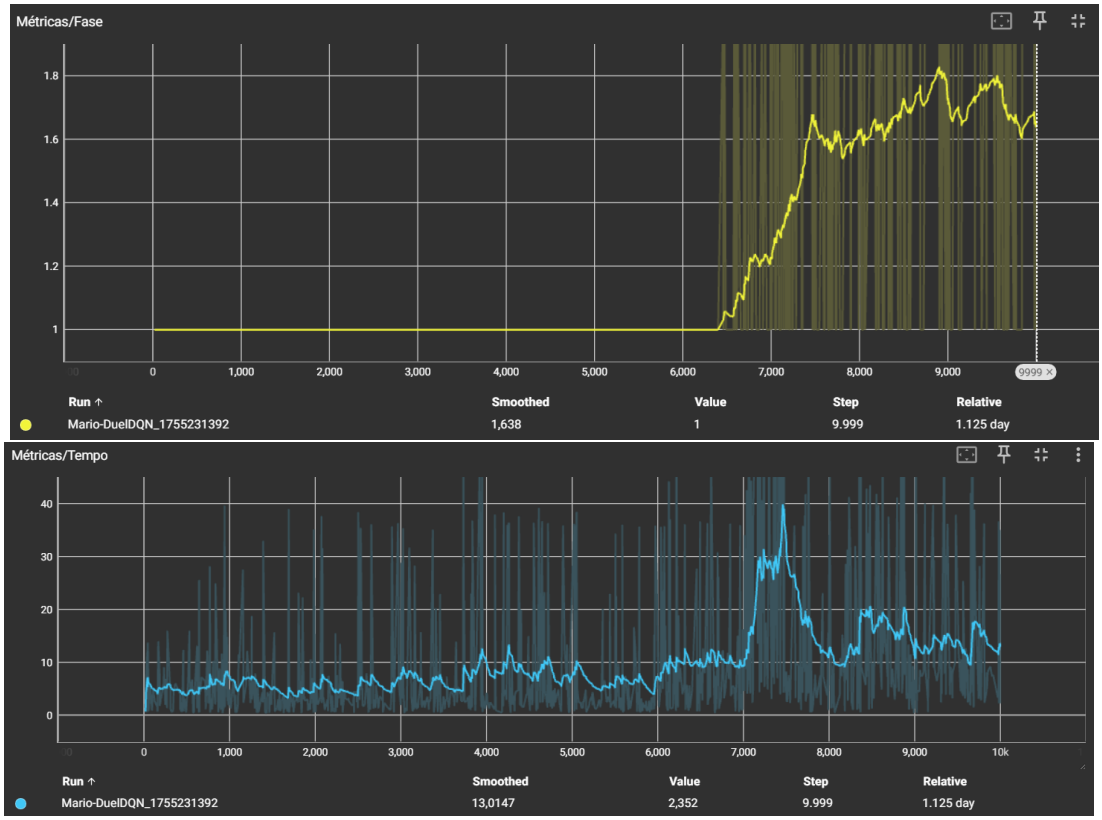

Figura 7 – Nível e Tempo de execução no ambiente Mario.

Os gráficos da 7 também servem para comprovar o aprendizado do agente. Analisando a fase alcançada por época, é possível notar que, a partir da época 7.500, o agente chegou ao segundo nível do jogo em 60% dos casos. Já o Tempo e o Total Steps podem indicar uma melhora, já que, com o passar das épocas, ambos aumentam, sugerindo que o agente está passando mais tempo vivo e tomando mais decisões. Porém, esses dois últimos parâmetros não devem ser os únicos para explicar a melhora, já que é possível que o agente fique preso em um pulo, o que também pode justificar os picos no gráfico de tempo. É importante notar que a métrica "Tempo" representa o tempo de execução médio de cada época; como o ambiente está acelerado em aproximadamente quatro vezes, um tempo de execução de 10 segundos significa que o Mario ficou vivo por 40 segundos dentro do jogo.

5.2 Sonic the Hedgehog - Sega Genesis

Para este segundo experimento, o ambiente escolhido foi o do jogo Sonic the Hedgehog, que também é um jogo de plataforma 2D, onde o principal objetivo é chegar ao final de cada fase no menor tempo possível, enquanto coleta a maior quantidade de itens. O jogo possui vários obstáculos, como inimigos, espinhos e buracos, com os quais o agente deve tomar cuidado. Além disso, o jogo Sonic possui alguns obstáculos adicionais que só podem ser ultrapassados se o personagem estiver acima de uma velocidade mínima, o que pode dificultar severamente o avanço. Existem também os "Anéis", que o agente pode coletar para conseguir uma pontuação maior, além de garantir que ele não morra na próxima vez que tomar dano. A tela do ambiente de treinamento pode ser observada na Figura 8.

Figura 8 – Tela do ambiente de treinamento do jogo Sonic

Foi necessária a biblioteca Gym Retro para implementar esse ambiente, pois ela permite carregar cenários de diversos jogos a partir da Memória Somente de Leitura (Read-Only Memory - ROM) instalada no computador. Essa biblioteca possui um mapeamento de botões para cada controle de jogo — neste caso, o do Sega Genesis. Porém, somente alguns botões e combinações são utilizados no jogo Sonic, por isso foi necessário definir o espaço de ações manualmente. A biblioteca também possui um parâmetro que armazena a pontuação, mas esse valor não deve ser utilizado diretamente como recompensa. Por isso,

foi necessário implementar uma função de recompensa que considerasse: pontuação, anéis coletados, progresso na fase, penalidade por morte e penalidade por ficar muito tempo parado. Os hiperparâmetros da recompensa estão presentes na Tabela 5

Tabela 5 – Hiperparâmetros utilizados na recompensa do jogo Sonic.

Hiperparâmetro	**Valor**
Multiplicador de avanço (forward_mult)	0.2
Multiplicador por anéis (ring_mult)	0.1
Multiplicador da pontuação (score_mult)	0.05
Penalidade por morte (death_penalty)	-5
Penalidade por inatividade (stuck_penalty)	-0.5 por frame

Diferente do Mario, o objetivo principal não é apenas terminar a fase, mas terminá-la no menor tempo possível, por isso o maior multiplicador por avançar e a penalidade por ficar preso. Outra funcionalidade diferente do experimento 1 é a contagem de cada época. No Mario, uma época era uma tentativa única. No ambiente do Sonic do Gym Retro, por padrão, uma época (done = true) é identificada como a perda de uma única vida. Ao morrer, Sonic perde uma vida e volta para o início da fase onde estava, retornando para a primeira fase apenas quando perde todas as suas vidas. Decidiu-se manter essa funcionalidade para aumentar o tempo de exploração do agente em fases novas. Os hiperparâmetros utilizados no treinamento do jogo Sonic estão na Tabela 6

Tabela 6 – Hiperparâmetros utilizados no treinamento do jogo Sonic.

Hiperparâmetro	**Valor**
Taxa de Aprendizagem (Learning Rate)	0.001
Tamanho do Lote (Batch Size)	64
Número de Épocas (Epochs)	2.000
Otimizador	AdamW
Capacidade da Memória	100.000
Epsilon Fixo (ϵ_{fixed})	0.01
Fator de Desconto (γ)	0.99
Frequência de Atualização da Rede Alvo (por época)	50

É importante notar que o número de épocas foi severamente reduzido, pois o ambiente do Sonic é mais pesado em comparação ao Mario, já que a tela possui uma quantidade maior de informações e o ambiente não é tão otimizado. Além disso, o Epsilon foi fixado, pois o agente apresentou um desempenho ligeiramente superior nas primeiras 100 épocas ao utilizar um epsilon fixo em vez de decaimento. Ao todo, este treinamento de 2.000 épocas durou 47 horas. A seguir, os principais resultados obtidos:

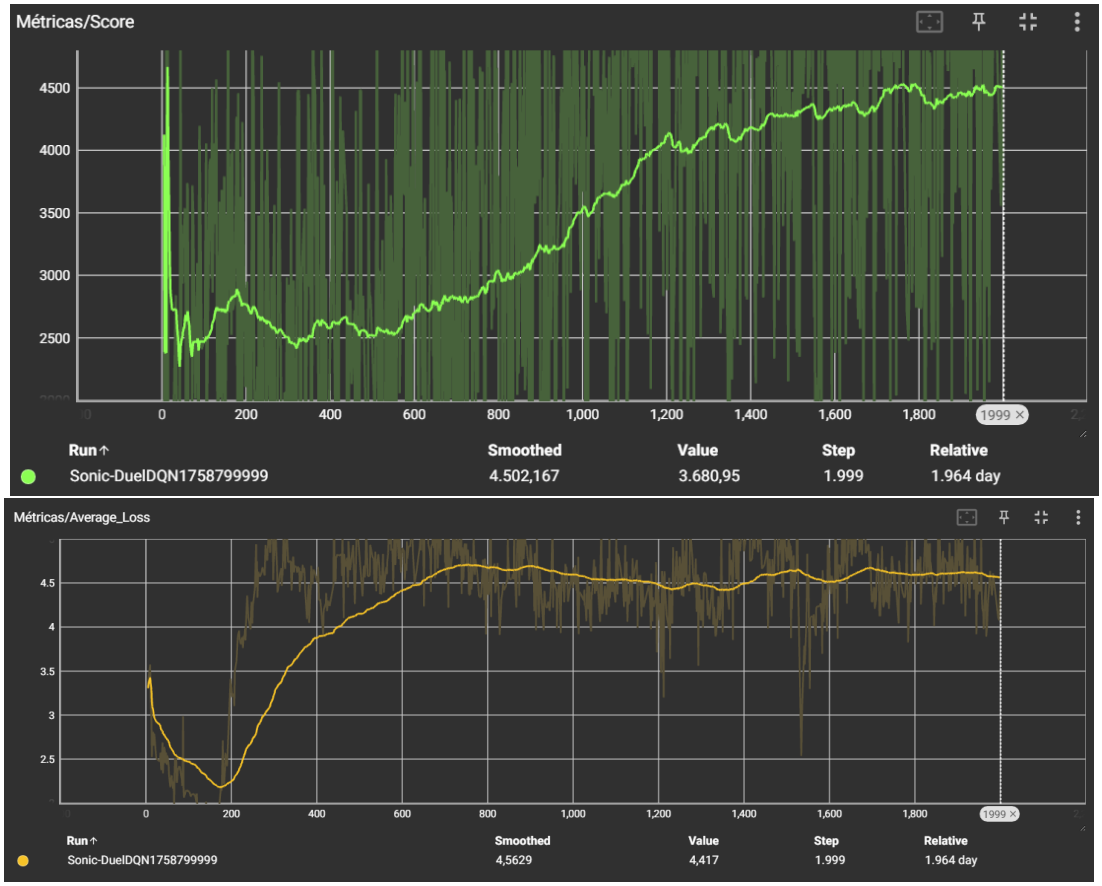

Figura 9 – Pontuação (Score) e Perda Média (Loss) no ambiente Sonic.

A partir dos gráficos da Figura 9, é possível identificar que a pontuação aumentou significativamente com o passar das épocas, enquanto o loss se manteve relativamente estável em torno de 4.5, indicando que o agente possivelmente encontrou uma política eficaz. No entanto, o fato de o loss não variar muito em um valor relativamente alto indica que, embora o agente tenha aprendido uma política robusta, ele tem dificuldade em prever quão boa ela é. Isso provavelmente se deve ao desafio da generalização, já que o agente se encontra frequentemente em fases novas, com novos elementos visuais e inimigos. Portanto, o loss alto não indica que o aprendizado falhou, mas sim que o agente frequentemente encontra surpresas. A seguir, as métricas de tempo médio de execução e fase máxima alcançada, lembrando que o jogo possui 3 fases, numeradas de 0 a 2:

Com a análise do tempo e da fase média alcançada observados na Figura 10, é possível notar que o tempo médio de execução não teve um grande aumento. Entretanto, ao final do treinamento, o valor médio da fase alcançada pelo agente é de aproximadamente 1.8, indicando que, em média, o agente sempre concluía a primeira fase e, na maioria das vezes, a segunda. Inclusive, nas últimas 150 épocas, o agente sempre terminou na terceira fase (valor 2).

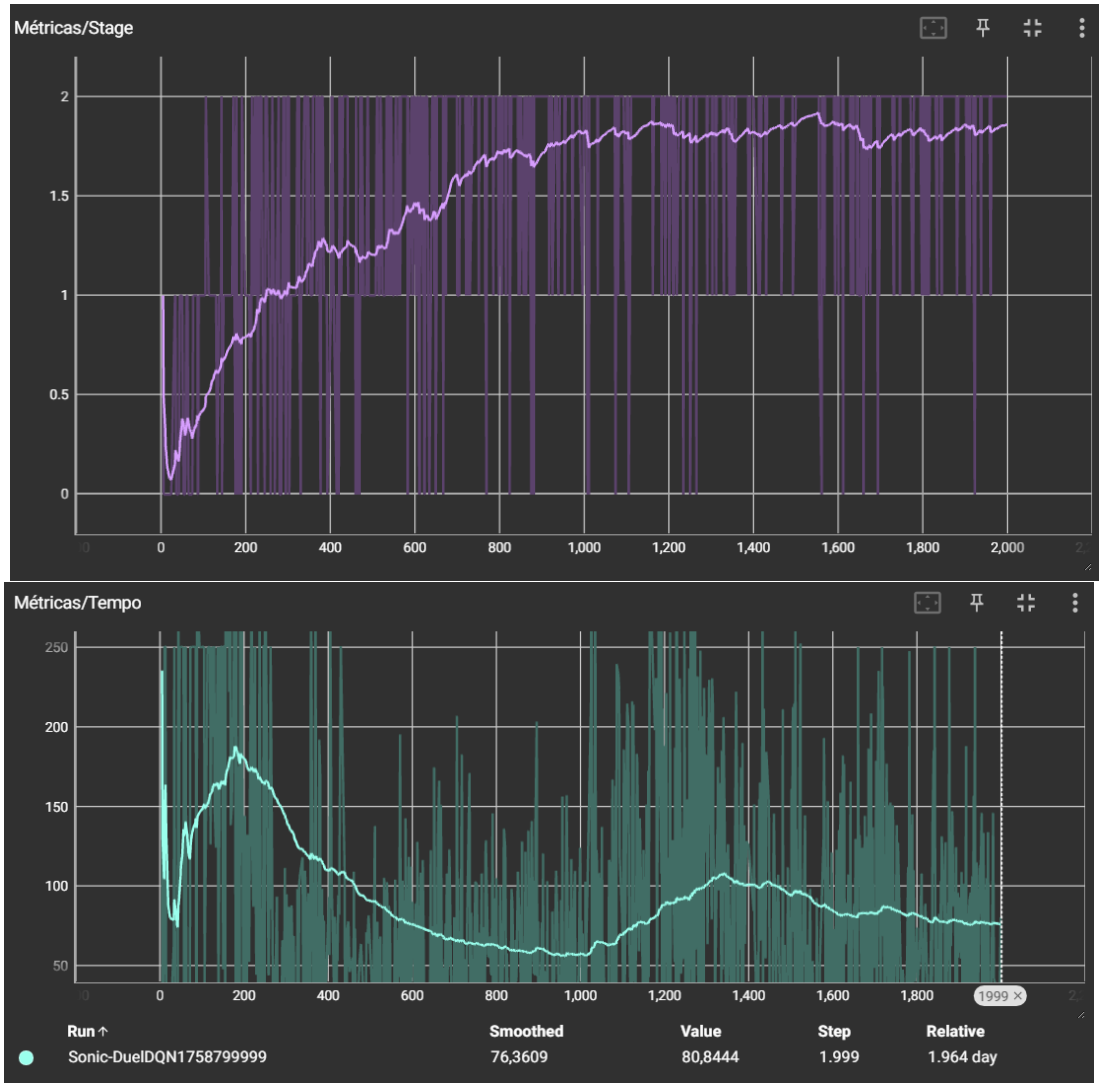

Figura 10 – Fase (Stage) e Tempo de execução no ambiente Sonic.

Além disso, foi possível notar que o ambiente do Sonic the Hedgehog é relativamente mais pesado que o dos outros experimentos, já que, mesmo com o 'frameSkip' de 4 frames, o tempo de execução é aproximadamente 1.1 vez mais rápido que o tempo padrão do ambiente.

5.3 R-Type - Arcade

Para o último experimento, foi utilizado o jogo R-Type, por ser relativamente diferente de Super Mario Bros. e Sonic. R-Type é um jogo de tiro (shooter) onde os obstáculos se movimentam em direção ao personagem. Nele, o principal objetivo é eliminar e sobreviver ao maior número de inimigos que aparecem na tela conforme o tempo passa. Para isso, o personagem deve atirar nos inimigos ao mesmo tempo que se preocupa em desviar de projéteis.

Este é um jogo relativamente difícil, já que, além de focar na própria sobrevivência, o agente também tenta eliminar os inimigos para obter uma pontuação maior. O ambiente em que o agente foi treinado pode ser observado na Figura 11

Figura 11 – Tela do ambiente de treinamento do jogo R-Type

Para este último experimento, também foi utilizado o ambiente da biblioteca Gym Retro, com uma ROM pessoal do jogo R-Type para arcade. Como essa biblioteca apenas mapeia todas as ações possíveis do controle de Arcade, também foi necessário filtrar essas ações para utilizar apenas aquelas que possuem alguma funcionalidade no jogo. A função de recompensa também teve de ser implementada manualmente, mas, diferente de Sonic, o agente não precisa avançar em direção aos obstáculos; ele apenas precisa sobreviver e desviar. Por isso, as seguintes métricas foram consideradas ao criar a função de recompensa: pontuação, bônus de sobrevivência e penalidade por morte. Na Tabela 7, seguem os hiperparâmetros da recompensa.

Tabela 7 – Hiperparâmetros utilizados na recompensa do jogo R-Type.

Hiperparâmetro	**Valor**
Bônus por frame sobrevivido (survival_bonus)	0.01
Multiplicador da pontuação (score_mult)	0.001
Penalidade por morte (death_penalty)	-5

Como o foco é garantir que o agente fique vivo pelo maior tempo possível e não necessariamente mate todos os inimigos, a recompensa pela pontuação do jogo foi reduzida, enquanto o bônus de sobrevivência e a penalidade por morte aumentaram. Alguns hiperparâmetros foram testados para o ambiente do R-Type; os que apresentaram a maior pontuação nas primeiras 1.000 épocas foram os utilizados na Tabela 8.

Tabela 8 – Hiperparâmetros utilizados no treinamento do jogo R-Type.

Hiperparâmetro	**Valor**
Taxa de Aprendizagem (Learning Rate)	0.0005
Tamanho do Lote (Batch Size)	128
Número de Épocas (Epochs)	8.000
Otimizador	AdamW
Capacidade da Memória	100.000
Epsilon Inicial (ϵ_{start})	1.0
Epsilon Final (ϵ_{end})	0.01
Decaimento do Epsilon	6.000
Fator de Desconto (γ)	0.99
Frequência de Atualização da Rede Alvo (por época)	100

Neste ambiente, o fim de uma época é identificado como a perda de 3 vidas. Assim, o agente avança para a próxima época somente após morrer 3 vezes; essa funcionalidade é padrão do ambiente e foi mantida. A execução do código ocorreu ao longo de 8.000 épocas, durando um total de 37 horas, com o ambiente acelerado em aproximadamente quatro vezes. Após o fim dessas épocas, os seguintes resultados foram obtidos:

Após a análise da Figura 12, é possível observar que, a partir da época 3.000, o tempo de execução se reduz consideravelmente. Entretanto, a pontuação média não decai tanto. Isso ocorre pois, a partir desse ponto, o agente aprendeu que eliminar vários inimigos, mesmo com uma alta chance de morte, é mais vantajoso do que apenas sobreviver por muito tempo. Por isso, acredita-se que mais testes seriam necessários com diferentes hiperparâmetros de treinamento e de recompensa para incentivar o agente a priorizar a sobrevivência. A seguir, algumas métricas adicionais:

Algo interessante a se notar na Figura 13 é que, mesmo que a pontuação não tenha aumentado muito, o loss médio se manteve relativamente baixo, o que pode indicar que o agente aprendeu a prever os valores de uma política subótima. Isso sugere a necessidade de maior exploração para que o agente encontre uma política melhor. Diante disso, é possível indicar que ocorreu aprendizado, mas de uma política subótima, e que mais estudos devem ser realizados neste ambiente para verificar se, com a mudança dos hiperparâmetros, o agente consegue encontrar uma política ótima.

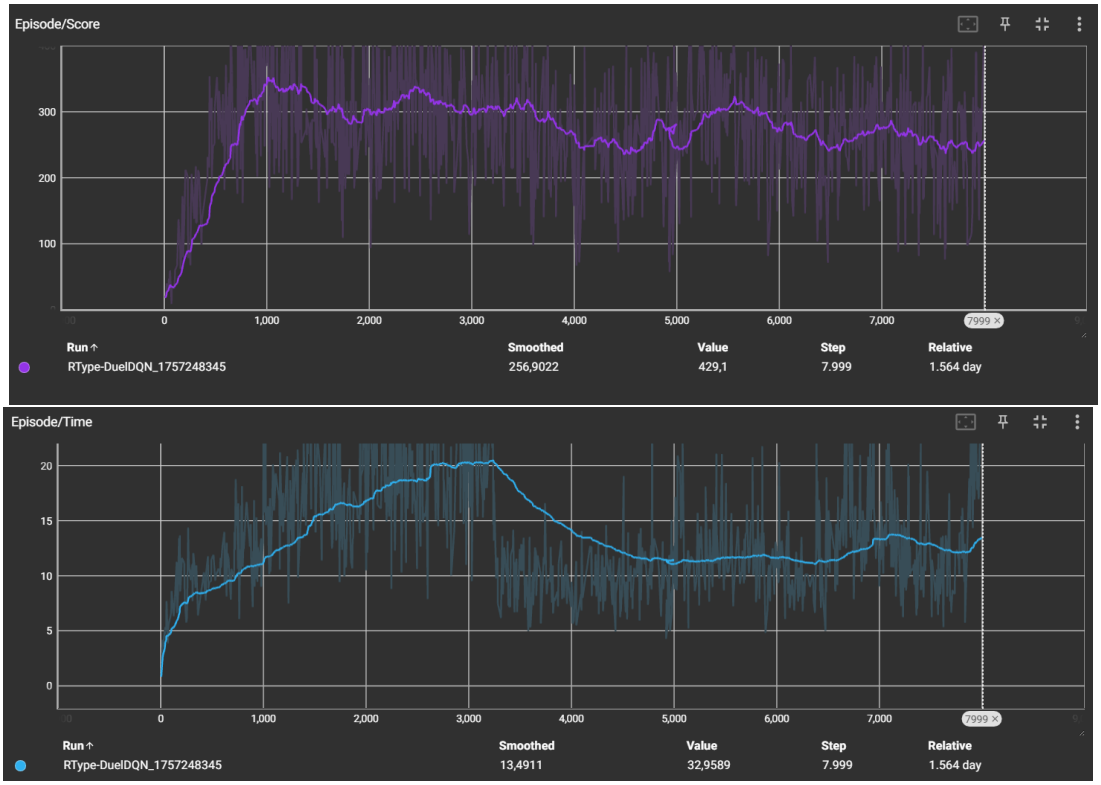

Figura 12 – Pontuação (Score) e Tempo de execução no ambiente R-Type.

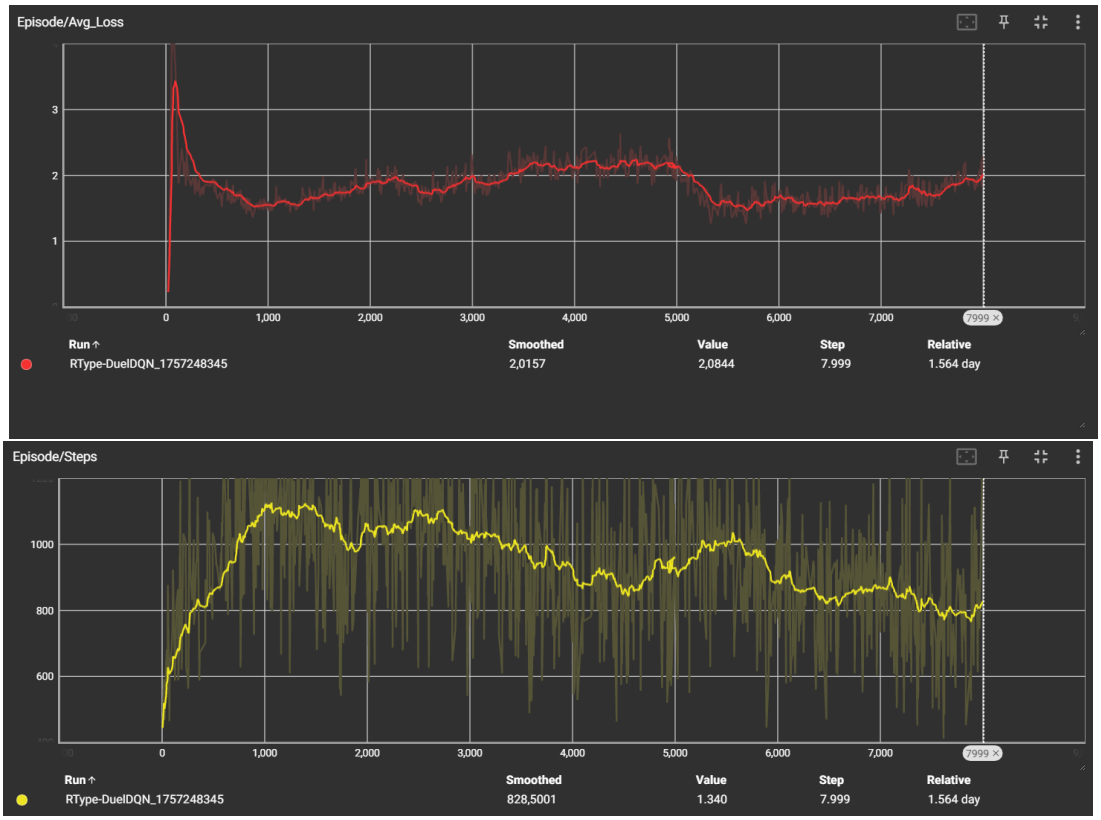

Figura 13 – Perda Média (Loss) e Passos (Steps) no ambiente R-Type.

6 Framework Geral de Treinamento

Após a implementação e o teste de três jogos diferentes, este capítulo busca detalhar, a partir do código criado, qual conjunto da infraestrutura de software pode ser considerado genérico para o treinamento de um agente em qualquer ambiente e quais componentes demandam um ajuste específico. Para isso, cada etapa do código será separada com base na quantidade de alterações necessárias. A figura 14 demonstra de forma visual quais partes são padrão e quais precisam de pequenos ou grandes ajustes.

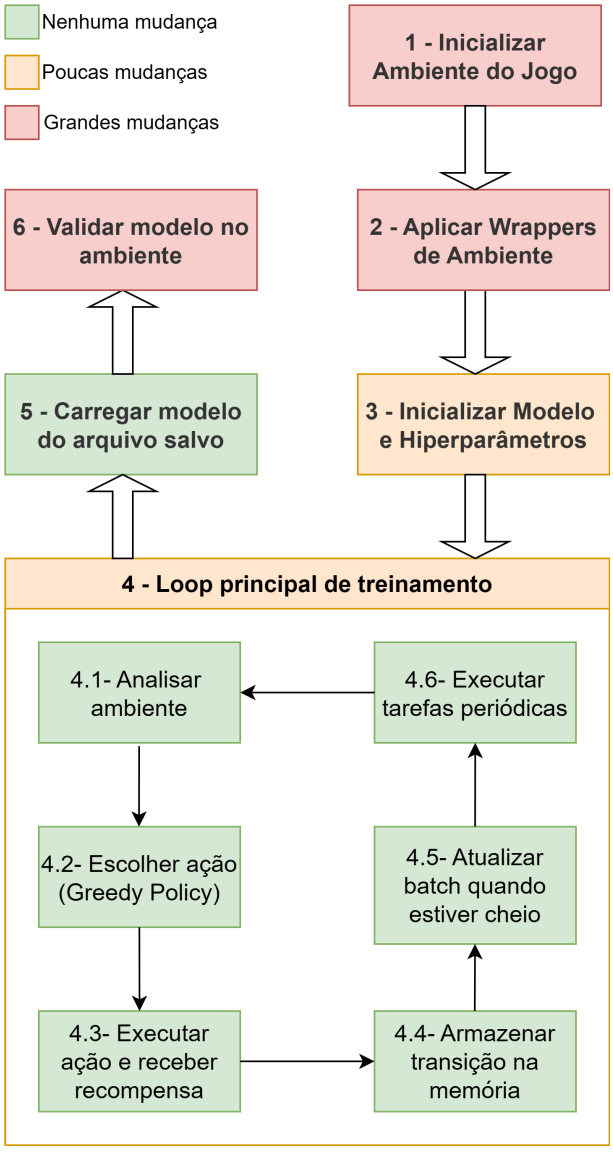

Figura 14 – Diagrama da Infraestrutura Genérica do Framework.

6.1 Ambiente do Jogo

Como o jogo será alterado, consequentemente o ambiente terá de ser mudado por completo. Para isso, é necessário alterar a inicialização do ambiente do Gym. Nessa etapa, apenas o nome do ambiente inicializado geralmente mudará, porém, muitas vezes, será necessário preparar a biblioteca para inicializar o ambiente. Por exemplo, o Gym Retro não possui as ROMs de todos os jogos por questões de direitos autorais, sendo necessário adquirir o jogo por meios oficiais e transformá-lo em uma ROM para uso pessoal. Além disso, o Gym permite inicializar o ambiente a partir de fases e estados salvos (save states) diferentes, por isso, muitas vezes, torna-se necessário definir não só o ambiente, mas também o ponto de partida do agente.

Outro ponto importante é que, frequentemente, o espaço das ações não estará bem definido, então pode ser necessário um pré-processamento antes de utilizar o wrapper `GameActions()`.

6.2 Wrappers de Ambiente

Alguns dos wrappers são utilizados apenas para o pré-processamento de dados e imagens para a rede DQN; por isso, alguns são genéricos, enquanto outros precisam de grandes mudanças. A seguir, a descrição de cada um dos Wrappers e, se necessárias, as mudanças que devem ser realizadas:

- **GameActions()**: O espaço de ações muitas vezes será o conjunto de todas as combinações possíveis de botões, mas só algumas são realmente necessárias para cada jogo. Essa função é totalmente específica e requer o conhecimento do conjunto de ações relevantes para o ambiente que se busca implementar.
- **NoopReset(noop_max)**: Como essa função apenas realiza ações 'noop' um número aleatório de vezes, ela é genérica para qualquer ambiente.
- **MaxAndSkip(n)**: Essa função apenas reduz a quantidade de decisões que a rede neural toma por segundo, sendo genérica na maioria dos casos. Entretanto, nem todos os jogos seguem o padrão de 60 FPS, por isso pode ser necessário ajustar o valor de n.
- **GameReward()**: Apenas alguns ambientes, como o Gym-Super-Mario-Bros, possuem uma função de recompensa bem definida. Por isso, em quase todos os cenários, será necessário implementar uma função própria, a depender do que for considerado importante para o agente aprender. Essa é uma das partes mais complexas, pois pode envolver o uso de variáveis internas e específicas de cada jogo, como no experimento do Sonic, em que o agente é recompensado por coletar anéis.

- **WarpFrame(size)**: Essa função reduz a quantidade de pixels da tela e converte a escala de cores para cinza, sendo, portanto, genérica para a maioria dos ambientes. Caso o jogo desejado já seja monocromático, a conversão não será necessária.
- **ScaledFloat()**: Normaliza os valores de cada pixel da tela, sendo uma função genérica para qualquer tipo de ambiente.
- **FrameStack(k)**: Empilha os últimos k frames antes de enviá-los para a rede neural. Essa função é genérica na maioria dos casos; apenas o valor de k poderia mudar a depender da taxa de FPS do ambiente.

6.3 Inicialização do Modelo DQN

Como a escala da imagem e o espaço de ações já foram tratados pelos Wrappers, a arquitetura e a inicialização do modelo DQN não seriam diferentes. Entretanto, raramente os mesmos hiperparâmetros funcionam para jogos distintos, então é necessário testar diferentes configurações para identificar quais são as mais adequadas para cada ambiente.

6.4 Loop Principal

A definição do que é uma 'época" nem sempre será a mesma para qualquer ambiente, então torna-se necessário alterar a condição de parada para identificar o fim de um episódio. Além disso, para ambientes em que o agente possui mais de uma vida, como em Sonic e R-Type, é necessário implementar algum sistema para gerenciar essas vidas, já que nem sempre a perda de uma significará uma nova época.

6.4.1 Analisar ambiente

Como o agente apenas analisa os frames do ambiente, o código em si não muda.

6.4.2 Escolher Ação

O espaço de ações já foi definido no wrapper `GameActions()`, então a escolha de uma ação é genérica, dado que o conjunto de ações possíveis já foi pré-definido.

6.4.3 Executar Ação e Receber Recompensa

A função que retorna a recompensa para uma dada ação a já foi definida no wrapper `GameReward()`, então o código não precisa ser alterado.

6.4.4 Armazenar transição

Como cada transição (state, action, reward, next-state, done) é genérica para cada ambiente, mudanças não são necessárias.

6.4.5 Atualização do batch

A memória temporária e o batch são estruturas de dados que independem do ambiente, por isso não precisam de mudanças (apenas seus tamanhos).

6.4.6 Tarefas Periódicas

Como as atividades periódicas de salvar o modelo em um arquivo e atualizar a rede alvo dependem unicamente da rede DQN, essa etapa é genérica.

6.5 Carregar Modelo do arquivo

Quando se carrega o modelo do arquivo, o que está sendo carregado é a rede neural em si, juntamente com todos os seus parâmetros. Por isso, o carregamento do modelo independe do ambiente.

6.6 Validar modelo

Uma vez que o modelo foi carregado e se deseja avaliá-lo, é necessário carregar o ambiente de teste, bem como os wrappers utilizados durante o treinamento. Como o ambiente e os wrappers não são genéricos, consequentemente a validação do modelo também requer essas mesmas mudanças.

7 Conclusão

Portanto, foi possível automatizar o processo de treinar um agente em jogos 2D, utilizando o algoritmo DQN e as bibliotecas Gymnasium e PyTorch. Os agentes em cada um dos experimentos foram capazes de aprender uma política que funciona especificamente para o ambiente em que estão. Embora não seja necessariamente a política ótima, é uma comprovação da capacidade de aprendizado do agente. Entretanto, a contribuição principal desta pesquisa reside na análise da infraestrutura de software utilizada.

A partir da implementação dos experimentos, identificou-se uma clara separação entre os componentes genéricos e os específicos. Verificou-se que, embora a definição de uma "época" no loop de treinamento não seja genérica, suas subetapas essenciais, bem como a inicialização do modelo, são genéricas para qualquer tipo de ambiente. A especificidade ocorre na inicialização do ambiente, na aplicação dos wrappers e na definição dos hiperparâmetros.

O estudo limitou-se a comprovar a capacidade do agente de aprender uma política em ambientes de jogos 2D com espaços de ações discretos. Para que o agente aprenda uma política robusta, seria necessária uma análise exaustiva dos hiperparâmetros e da função de recompensa de cada ambiente, visto que uma amostra limitada foi testada neste trabalho.

Para estudos futuros, recomenda-se a validação desta arquitetura em ambientes mais complexos, como ambientes tridimensionais ou com espaços de ações contínuos. Adicionalmente, propõe-se a criação de uma linguagem de domínio específico (DSL) para simplificar a configuração dos wrappers e hiperparâmetros, visando facilitar a adaptação do sistema para novos jogos.

Referências

BROCKMAN, G.; CHEUNG, V.; PETTERSSON, L.; SCHNEIDER, J.; SCHULMAN, J.; TANG, J.; ZAREMBA, W. **OpenAI Gym**. 2016. ArXiv:1606.01540v1 [cs.LG]. Disponível em: <https://arxiv.org/abs/1606.01540>. Acesso em: 26 jun. 2025. Citado 2 vezes nas páginas 28 e 29.

CHAPLOT, D. S.; LAMPLE, G.; SATHYENDRA, K. M.; SALAKHUTDINOV, R. Transfer learning for deep reinforcement learning in 3d environments. In: **NIPS 2016 Workshop on Deep Reinforcement Learning**. [s.n.], 2016. Disponível em: <https://devendrachaplot.github.io/papers/nips16_Transfer_Deep_RL.pdf>. Citado na página 31.

Christian Kauten. **Super Mario Bros. for OpenAI Gym**. 2018. Disponível em: <https://pypi.org/project/gym-super-mario-bros/>. Acesso em: 20 mar. 2025. Citado na página 28.

FU, X. A comparison of dqn and dueling dqn in a super mario environment. In: **Proceedings of the 2023 International Conference on Data Science, Advanced Algorithm and Intelligent Computing (DAI 2023)**. Atlantis Press, 2024. p. 221–231. Disponível em: <https://doi.org/10.2991/978-94-6463-370-2_25>. Citado 3 vezes nas páginas 10, 13 e 26.

GAOL, F. T. J. L.; MUNIR, R. A combinatorial analysis of tic-tac-toe and the theoretical advantage of playing first. **J. INFORMATIKA**, Program Studi Teknik Informatika STEI-ITB, v. 15, n. 1, p. 1–6, 2021. Citado na página 22.

HASSELT, H. V.; GUEZ, A.; SILVER, D. Deep reinforcement learning with double q-learning. In: **Proceedings of the AAAI conference on artificial intelligence**. [s.n.], 2016. v. 30, n. 1. Disponível em: <https://doi.org/10.1609/aaai.v30i1.10295>. Citado na página 24.

HU, C.; ZHAO, Y.; WANG, Z.; DU, H.; LIU, J. **Games for Artificial Intelligence Research: A Review and Perspectives**. 2024. Disponível em: <https://doi.org/10.1109/TAI.2024.3410935>. Citado na página 10.

HU, L.; PENG, R.; YANG, J. Application of reinforcement learning in games. In: **Proceedings of the 1st International Conference on Modern Logistics and Supply Chain Management - Volume 1: MLSCM**. [s.n.], 2024. p. 129–134. ISBN 978-989-758-738-2. Disponível em: <https://doi.org/10.5220/0013234900004558>. Citado na página 15.

MNIH, V.; KAVUKCUOGLU, K.; SILVER, D.; GRAVES, A.; ANTONOGLOU, I.; WIERSTRA, D.; RIEDMILLER, M. **Playing Atari with Deep Reinforcement Learning**. 2013. ArXiv:1312.5602v1 [cs.LG]. Disponível em: <https://arxiv.org/abs/1312.5602>. Acesso em: 26 jun. 2025. Citado 4 vezes nas páginas 10, 13, 23 e 24.

MNIH, V.; KAVUKCUOGLU, K.; SILVER, D.; RUSU, A. A.; VENESS, J.; BELLEMARE, M. G.; GRAVES, A.; RIEDMILLER, M.; FIDJELAND, A. K.;

OSTROVSKI, G.; PETERSEN, S.; BEATTIE, C.; SADIK, A.; ANTONOGLOU, I.; KING, H.; KUMARAN, D.; WIERSTRA, D.; LEGG, S.; HASSABIS, D. Human-level control through deep reinforcement learning. **Nature**, v. 518, n. 7540, p. 529–533, 2015. Disponível em: <https://doi.org/10.1038/nature14236>. Acesso em: 26 jun. 2025. Citado na página 24.

MORALES, M. **Grokking Deep Reinforcement Learning**. [S.l.]: Manning Publications, 2020. ISBN 9781617296756. Citado na página 20.

NOVAC, A. et al. **A Comparative Survey of PyTorch vs. TensorFlow for Deep Learning: Usability, Performance, and Deployment Trade-offs**. 2024. Citado na página 27.

PASZKE, A.; GROSS, S.; MASSA, F.; LERER, A.; BRADBURY, J.; CHANAN, G.; KILLEEN, T.; LIN, Z.; GIMELSHEIN, N.; ANTIGA, L.; DESMAISON, A.; KöPF, A.; YANG, E.; DEVITO, Z.; RAISON, M.; TEJANI, A.; CHILAMKURTHY, S.; STEINER, B.; FANG, L.; BAI, J.; CHINTALA, S. **PyTorch: An Imperative Style, High-Performance Deep Learning Library**. 2019. ArXiv:1912.01703v1 [cs.LG]. Disponível em: <https://arxiv.org/abs/1912.01703>. Acesso em: 26 jun. 2025. Citado na página 27.

PERI, J. Video gaming industry in the us. **International Journal of Science and Research Archive**, v. 11, p. 1257–1265, fev. 2024. Disponível em: <https://doi.org/10.30574/ijsra.2024.11.1.0194>. Acesso em: 26 jun. 2025. Citado na página 11.

PyTorch Foundation. **PyTorch: An Open Source Machine Learning Framework**. 2023. Disponível em: <https://pytorch.org/blog/>. Acesso em: 20 mar. 2025. Citado na página 27.

RAJAGEDE, R. A. Maintain agent consistency in surakarta chess using dueling deep network with increasing batch. **IIUM Engineering Journal**, International Islamic University Malaysia, v. 23, n. 1, p. 159–171, 2022. Disponível em: <https://doi.org/10.31436/iiumej.v23i1.1807>. Citado na página 16.

SAMUEL, A. L. Some studies in machine learning using the game of checkers. **IBM Journal of Research and Development**, v. 3, n. 3, p. 210–229, 1959. Disponível em: <https://doi.org/10.1147/rd.33.0210>. Citado na página 20.

SCHUMACHER, J.; PLEINES, M. **Improving Bidding and Playing Strategies in the Trick-Taking game Wizard using Deep Q-Networks**. 2022. Disponível em: <https://doi.org/10.1109/CoG51982.2022.9893614>. Citado na página 16.

SEWAK, M. Deep q network (dqn), double dqn, and dueling dqn. In: **Deep Reinforcement Learning**. Springer, 2019. p. 95–108. Disponível em: <https://doi.org/10.1007/978-981-13-8285-7_8>. Citado na página 26.

SHAO, K.; TANG, Z.; ZHU, Y.; LI, N.; ZHAO, D. **A Survey of Deep Reinforcement Learning in Video Games**. 2019. ArXiv:1912.10944v1 [cs.MA]. Disponível em: <https://doi.org/10.1109/CIG.2018.8490423>. Acesso em: 26 jun. 2025. Citado na página 14.

SHARIFI, A.; ZHAO, R.; SZAFRON, D. Learning companion behaviors using reinforcement learning in games. **Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment**, v. 6, n. 1, p. 69–75, Oct. 2010. Disponível em: <https://doi.org/10.1609/aiide.v6i1.12392>. Citado 2 vezes nas páginas 10 e 11.

SILVER, D.; HUANG, A.; MADDISON, C. J.; GUEZ, A.; SIFRE, L.; DRIESSCHE, G. van den; SCHRITTWIESER, J.; ANTONOGLOU, I.; PANNEERSHELVAM, V.; LANCTOT, M.; DIELEMAN, S.; GREWE, D.; NHAM, J.; KALCHBRENNER, N.; SUTSKEVER, I.; LILLICRAP, T.; LEACH, M.; KAVUKCUOGLU, K.; GRAEPEL, T.; HASSABIS, D. Mastering the game of go with deep neural networks and tree search. **Nature**, v. 529, n. 7587, p. 484–489, 2016. Disponível em: <https://doi.org/10.1038/nature16961>. Citado na página 21.

SUTTON, R. S.; BARTO, A. G. **Reinforcement Learning: An Introduction**. 2nd. ed. The MIT Press, 2018. Disponível em: <http://incompleteideas.net/book/the-book-2nd.html>. Citado 3 vezes nas páginas 18, 20 e 22.

TIAGO, L. M. P. **Implementação de um Agente DQN para Ambientes do Gym Retro**. GitHub, 2025. Disponível em: <https://github.com/Luc4smp/dqn-agent-retro>. Citado na página 38.

_____. **Implementação de um Agente DQN para Gym Super Mario Bros**. GitHub, 2025. Disponível em: <https://github.com/Luc4smp/dqn-agente-mario>. Citado na página 38.

WAFA, H.; SANTOSO, J. Multiagent simulation on hide and seek games using policy gradient trust region policy optimization. In: **2020 7th International Conference on Advance Informatics: Concepts, Theory and Applications (ICAICTA)**. [s.n.], 2020. p. 1–5. Disponível em: <https://doi.org/10.1109/ICAICTA49861.2020.9429052>. Citado na página 15.

WANG, Z.; SCHAUL, T.; HESSEL, M.; HASSELT, H.; LANCTOT, M.; FREITAS, N. Dueling network architectures for deep reinforcement learning. In: BALCAN, M.-F.; WEINBERGER, K. Q. (Ed.). **Proceedings of The 33rd International Conference on Machine Learning**. [S.l.]: PMLR, 2016. (Proceedings of Machine Learning Research, v. 48), p. 1995–2004. Citado na página 24.

WATKINS, C.; DAYAN, P. Q-learning. **Machine Learning**, v. 8, n. 3, p. 279–292, 1992. Disponível em: <https://doi.org/10.1007/BF00992698>. Acesso em: 26 jun. 2025. Citado na página 23.

WATKINS, C. J. **Learning from Delayed Rewards**. Tese (Doutorado) — King's College, Cambridge University, 1989. Citado 2 vezes nas páginas 21 e 22.