UNIVERSIDADE FEDERAL DE JUIZ DE FORA
INSTITUTO DE CIÊNCIAS EXATAS
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Celso Gabriel Dutra Almeida Malosto

Ferramenta de apoio automatizado à fase de balanceamento de regras em jogos de turnos

Juiz de Fora
2026

Celso Gabriel Dutra Almeida Malosto

Ferramenta de apoio automatizado à fase de balanceamento de regras em jogos de turnos

Trabalho de conclusão de curso apresentado ao Bacharelado em Sistemas de Informação da Universidade Federal de Juiz de Fora como requisito parcial à obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. Igor de Oliveira Knop
Coorientadora: Prof.ª Dr.ª Luciana Conceição Dias Campos

Juiz de Fora
2026

Malosto, Celso Gabriel Dutra Almeida.
Ferramenta de apoio automatizado à fase de balanceamento de regras em jogos de turnos / Celso Gabriel Dutra Almeida Malosto. – Juiz de Fora, 2026.
66 f. : il.

Orientador: Igor de Oliveira Knop
Coorientadora: Luciana Conceição Dias Campos
Trabalho de Conclusão de Curso (bacharelado) – Universidade Federal de Juiz de Fora, Instituto de Ciências Exatas. Bacharelado em Sistemas de Informação, 2026.

1. Game design. 2. Play-test automático. 3. AlphaZero. 4. Redes neurais artificiais. I. Knop, Igor de Oliveira, orient. II. Campos, Luciana Conceição Dias, coorient. III. Instituto de Ciências Exatas. IV. Título.

Celso Gabriel Dutra Almeida Malosto

Ferramenta de apoio automatizado à fase de balanceamento de regras em jogos de turnos

Trabalho de conclusão de curso apresentado ao Bacharelado em Sistemas de Informação da Universidade Federal de Juiz de Fora como requisito parcial à obtenção do título de Bacharel em Sistemas de Informação.

Aprovado em 23 de janeiro de 2026

Banca examinadora

Prof. Dr. Igor de Oliveira Knop – Orientador
Universidade Federal de Juiz de Fora

Prof.ª Dr.ª Luciana Conceição Dias Campos – Coorientadora
Universidade Federal de Juiz de Fora

Prof. Dr. Heder Soares Bernardino
Universidade Federal de Juiz de Fora

Prof. Dr. Marcelo Caniato Renhe
Universidade Federal de Juiz de Fora

AGRADECIMENTOS

Inescapavelmente, gostaria de fazer um aceno final às paredes e jardins com quem convivi noturnamente durante todos esses anos de formação na Universidade Federal de Juiz de Fora. Foi neles que contei com o apoio e o carinho de professores, técnicos e colegas.

Em especial, agradeço ao Prof Dr. Igor Knop e à Prof.ª Dr.ª Luciana Campos por carinhosamente terem atuado como orientadores desta pesquisa. Eles permitiram-me a flexibilidade da qual eu precisei para explorar, mas reforçando o ponderamento sobre o escopo.

Agradeço também ao querido coordenador do meu curso de Bacharelado em Sistemas de Informação, Prof. Dr. Luciano Jerez, que me ajudou a percorrer o caminho e compreender o meu objetivo nessa área de conhecimento.

Agradeço novamente ao Prof. Dr. Igor Knop, e aos Prof. Dr. Stênio de Sã e Prof. Dr. Marcelo Caniato por terem atuado brilhantemente como tutores do Grupo de Educação Tutorial em Sistemas de Informação (GetSi), do qual eu fiz parte e que forneceu recursos para o desenvolvimento desta pesquisa por meio de uma bolsa de graduação financiada pela UFJF.

Agradeço a todos os amigos cujas trajetórias se cruzaram com a minha nesse tempo. Em especial, desejo um abraço ao meu colega Lucas Paiva, com quem agradavelmente estudei diversas disciplinas e elaborei extensos trabalhos.

Agradeço com muito carinho ao meu companheiro, Samuel Nascimento, que esteve ao meu lado em todos os momentos de que precisei, apoiando-me emocionalmente e com sugestões práticas. Foi com a felicidade que me mostrou que pude me motivar a realizar esta pesquisa.

Com especialíssima gratidão, agradeço à minha família e aos meus pais, Leandra Dutra e Celso Malosto, por acreditarem em mim e por me apoiarem de todas as formas durante tantas mudanças necessárias para meu amadurecimento como pessoa e como pesquisador.

Perguntar-me-á o leitor porque não o construí mais cedo, ao mesmo tempo que os meus dirigíveis. É que o inventor, como a natureza de Linneu, não faz saltos; progride de manso, evolui. Comecei por fazer-me bom piloto de balão livre e só depois ataquei o problema de sua dirigibilidade. Fiz-me bom aeronauta no manejo dos meus dirigíveis; durante muitos anos, estudei a fundo o motor a petróleo e só quando verifiquei que o seu estado de perfeição era bastante para fazer voar, ataquei o problema do mais pesado que o ar (Dumont, 1918, p. 49).

RESUMO

Introdução: O mercado de jogos autorais apresenta um crescimento contínuo, com milhares de jogos publicados anualmente nas maiores feiras do mundo. Esse crescimento cria uma demanda por melhorias nas ferramentas de apoio à fase de criação. Nessa fase, um protótipo passa por *play-test* repetidamente a fim de identificar desbalanceamentos e estratégias dominantes, o que exige muito tempo e recursos humanos. **Objetivos:** Esta pesquisa busca desenvolver meios de aliviar a necessidade da equipe de *play-test*, ao explorar por exaustão os sistemas do jogo usando agentes inteligentes. Dessa forma, espera-se que os humanos se concentrem nos aspectos da experiência de jogo e não em testes de estresse. **Métodos:** Esta é uma pesquisa exploratória na qual é avaliado o uso de agentes inteligentes treinados por métodos de *self-play* inspirados pelo projeto *AlphaZero*, que é baseado nos métodos de busca em árvore de Monte Carlo (MCTS) e de redes neurais residuais (ResNets). Foi criado um sistema computacional de representação de jogos de turnos, de geração e treinamento de agentes inteligentes e de simulação e avaliação de partidas, que foi testado com o jogo Ligue-4. Dados colhidos durante e após o processo de treinamento são utilizados para levantar observações do comportamento emergente das regras do jogo. **Resultados:** Com as partidas sintéticas, a equipe de desenvolvimento passa a ter um conjunto de partidas para avaliar, coletadas com custos reduzidos. Essa abordagem permitiu construir um sistema que gera métricas acerca do jogo e visualizá-las, o que indicou a viabilidade de usar o método de *play-test* automatizado como apoio ao projetista, ainda que mais experimentos sejam requeridos utilizando diferentes parâmetros.

Palavras-chave: game design; play-test automático; AlphaZero; redes neurais artificiais.

ABSTRACT

Introduction: The market for designer's games shows continuous growth, with thousands of games published annually at the world's largest fairs. This growth creates a demand for improvements in tools supporting the creation phase. In this phase, a prototype undergoes *play-test* repeatedly to identify imbalances and dominant strategies, which requires significant time and human resources. **Objectives:** This research seeks to develop ways to alleviate the need for a *play-test* team by exhaustively exploring the game systems using intelligent agents. Thus, humans are expected to focus on aspects of the game experience rather than on stress testing. **Methods:** This is an exploratory research evaluating the use of intelligent agents trained by *self-play* methods inspired by the *AlphaZero* project, which is based on Monte Carlo tree search (MCTS) and residual neural networks (ResNets) methods. A computer system was created for representing turn-based games, generating and training intelligent agents, and simulating and evaluating matches, which was tested with the game *ConnectFour*. Data collected during and after the training process are used to raise observations about the emergent behavior of the game rules. **Results:** With synthetic matches, the development team now has a set of matches to evaluate, collected with reduced costs. This approach allowed creating a system to generate metrics about the game and visualize them, which has indicated the viability of using the automated *play-test* method as support for the designer, although further experiments using different parameters are required.

Keywords: game design; automated play-test; AlphaZero; artificial neural networks.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

APTS Sistema de Teste de Jogabilidade Automatizado.

COMPUTAÇÃO

Adam estimativa de momento adaptativo.
API interface de programação de aplicações.
CNN rede neural convolucional.
IA inteligência artificial.
JSON notação de objetos do JavaScript.
MCTS busca em árvore de Monte Carlo.
ReLU unidade linear retificada.
ResNet rede neural residual.
UCT limite superior de confiança aplicado a árvores.

SUMÁRIO

1 INTRODUÇÃO

Jogos são conceituados como atividades com propósito bem definido, o qual comumente é vencer um desafio. Um jogador apenas pode ser considerado vitorioso caso ele atinja o objetivo segundo condições pré-estabelecidas, definidas como as regras do jogo. Tais regras permitem diferentes estratégias, as quais podem ser consideradas melhores ou piores para obter a vitória, de acordo com o contexto da partida (Suits, 1967).

Dentre as categorias existentes, destacam-se os jogos de turnos (*turn-based games*), em que o tempo de partida evolui em unidades discretas. Essas são chamadas de turnos, nos quais os jogadores realizam um número finito de movimentos que resultam em mudanças no estado do jogo. Comumente, os turnos se alternam de forma pré-estabelecida, ao que se denomina rodada. Nessa classe de jogos, as rodadas se sucedem até que a partida chegue a um estado final o qual é avaliado com alguma métrica para decidir o sucesso ou fracasso dos jogadores dentro do desafio proposto. Uma característica marcante deles é a possibilidade representar a tomada de decisão dos jogadores durante uma partida por meio de árvores de decisão. Essas estruturas permitem formalizar em um grafo os movimentos possíveis, definidos pelas regras, e os estados resultantes delas (Salen; Zimmerman, 2003, p. 410).

O mercado dos jogos de tabuleiro (*board game*) modernos teve um marco com o lançamento de *Colonizadores de Catan* (Teuber, 1995), quando jogos contemporâneos se tornaram populares mundialmente a partir da Alemanha e criaram um novo movimento cultural. Atualmente existem sites focados em catalogar esses jogos, sendo o maior o BoardGameGeek[1], que registra mais de 140 mil itens entre jogos, suas reimplementações e suas subsequentes expansões.

Uma grande parcela desses jogos se destaca pelo seu perfil tático ou estratégico durante as partidas, com uma série de reações em cadeia oriundas dos movimentos escolhidos pelas decisões dos jogadores, ocasionando diversas dinâmicas sociais e complexidade emergente. Estes jogos são também conhecidos como *designer's games*, ou jogos autorais, por trazerem o nome do autor na capa. Eles são fruto de uma organização de criadores que proporciona uma série de benefícios para um mercado baseado em novidades (Woods, 2012). Anualmente, acima de 1000 novos jogos são apresentados nas maiores convenções do meio, além de reimpressões, reedições expansões de conteúdo e jogabilidade (BoardGameGeek, LLC, 2025).

O processo de criação de um jogo é um processo exaustivamente iterativo. O criador implementa a sua ideia em um protótipo para facilitar as contínuas modificações necessárias. Assim que o autor julga que esse protótipo está pronto dentro da experiência de jogo desejada, ele deve ser testado na que se denomina a fase de *play-test* (teste de jogabilidade). Esta é a etapa na qual se realizam partidas para explorar o comportamento dos sistemas e encontrar possíveis desequilíbrios (Fullerton, 2019; Marcelo; Pescuite, 2009).

Deve-se ressaltar que desenvolvimento de um jogo autoral é um processo complexo e custoso, sobretudo durante a fase de *play-test*. Não é incomum o autor realizar os testes sozinho, simulando vários jogadores. Contudo, ao considerar dinâmicas e mecanismos mais complexos, é necessário convidar outras pessoas para auxiliá-lo. Adicionalmente, são feitos testes de estresse

[1] Acesso em: https://boardgamegeek.com/wiki/page/Welcome_to_BoardGameGeek.

para diversos sistemas do jogo. Entre eles, podemos citar a realização da mesma ação durante quase toda a partida, caso aparente ser muito vantajosa, o que ajuda a verificar se ela consegue sobrepujar todas as demais (Marcelo; Pescuite, 2009). Esta é a etapa do *play-test* que é conhecida como balanceamento.

A busca pelo balanceamento em jogos apresenta um desafio grande para a indústria, pois o próprio termo não é consenso (Becker; Görlich, 2020). Tal processo é altamente dependente de contexto, com desdobramentos para equilíbrio matemático, progressão de dificuldade, progressão de conteúdo, variedade de estratégias e imparcialidade entre jogadores. Cada um desses grupos apresenta suas próprias características, constituindo subsistemas altamente inter-relacionados de um sistema complexo maior, que é o jogo (Romero; Schreiber, 2021).

Essa etapa, na qual partidas do jogo são performadas repetidamente, tem alto custo de recursos humanos e tempo. É difícil manter um grupo de teste ativo e focado, dado que se trata de um processo cansativo quando o número de partidas começa a ficar alto. Além disso, o objetivo do teste repetitivo nem sempre é claro para os jogadores, de forma que o projetista tenta não contaminar a partida divulgando quais mecanismos estão sob teste (Trzewiczek, 2017).

Ademais, efeitos sobre os próprios testadores podem influenciar os resultados dos testes com suas expectativas, humores, excessos ou falta de concentração. Esses são pontos importantes a se observar em um teste de experiência de jogo (Marcelo; Pescuite, 2009), mas não são relevantes quando os objetivos são equilíbrios durante testes de estresse, nos quais os movimentos executados devem ser puramente efetivos e alheios ao divertimento e emoções dos jogadores ou dinâmicas do grupo.

O estudo de jogos de mesa por meios computacionais segue a própria história da Computação, em que pioneiros buscaram construir máquinas, modelos e algoritmos para jogar xadrez em um nível avançado (Silver et al., 2018). Tradicionalmente, jogos de tabuleiro são descritos por estados discretos e tidos como jogos combinatoriais. A área foi conduzida pelo estudo da busca eficiente em árvores de decisão via variações do algoritmo minimax e poda em árvore alfa-beta nas últimas duas décadas (Plaat et al., 1995).

Os estudos continuaram com as heurísticas especializadas até que os resultados da busca em árvore de Monte Carlo (MCTS) na implementação de algoritmos de decisão se mostraram positivos (Holmgård et al., 2019; Kocsis; Szepesvári, 2006). Seu uso não requeria qualquer outro conhecimento prévio além das regras do jogo e apresentava um bom desempenho sem necessitar que se implementasse uma heurística especializada.

Com base nela, o projeto *AlphaZero*, do laboratório de pesquisa Google DeepMind, se destacou por substituir a necessidade de adaptar conhecimento de domínio de um jogo específico pelo uso de uma rede neural residual (ResNet), atuando como um algoritmo de aprendizado profundo independente de heurísticas especializadas (Silver et al., 2016). Essa estratégia permitiu realizar buscas eficientes na árvore de decisão através de um modelo treinado por aprendizado profundo. Esse método teve várias aplicações em jogos diferentes, como o Shogi e Go, que apresentavam complexidade superior ao xadrez. Assim, ao considerar o aprendizado não informado resultante das repetidas partidas simuladas, percebe-se que essas tecnologias são promissoras para aprimorar jogos de mesa em desenvolvimento, observando desde a avaliação do estado do jogo, bem como a massa de dados gerada ao final do treinamento.

Dado este contexto, o presente trabalho continua uma pesquisa exploratória para investigar relações de balanceamento em jogos durante sua criação (Araki; Knop, 2020; Malosto; Campos; Knop, 2025; Malosto; Knop; Campos, 2023). Seu objetivo geral é oferecer perspectivas e ferramentas inovadoras ao cenário de criação de jogos de turnos, estabelecendo como foco a fase de *play-test*.

Os autores estabelecem como hipótese que é viável construir sistemas computacionais para a execução de partidas sintéticas que ofereçam dados relevantes aos projetistas de jogos, de forma a reduzir o emprego de recursos humanos nessa fase. Assim, espera-se que a participação de pessoas seja empregada para investigar aspectos lúdicos, sociais e a experiência do jogador, ao passo em que os testes repetitivos sejam realizados majoritariamente por agentes inteligentes.

A nível dos objetivos específicos, é proposto construir um ambiente de *play-test* simulado para auxiliar as pessoas autoras de jogos a realizar as primeiras iterações do processo de teste. Esse sistema deve permitir representar um jogo de turnos arbitrário e descrever suas regras na forma de classes específicas elaboradas em código-fonte da linguagem JavaScript. Ele deve então oferecer ao usuário um ambiente de simulação de partidas sintéticas que exporte históricos das jogadas. Nessa perspectiva, é necessário estudar a modelagem de estruturas de dados capazes de organizar informações sobre diferentes conceitos, como: jogo, partida, rodada, turno, jogador, movimento e estado.

Outro requisito do sistema é oferecer formas de avaliação dos movimentos viáveis a partir de um estado fornecido pelo usuário. Isso deve ser implementado tanto pelo método clássico do algoritmo de MCTS, como também pelo uso de agentes inteligentes guiados por ResNets. O treinamento desses é feito no processo de aprendizado por reforço, o que requer que o sistema gerencie a criação de massas de dados por meio do processo de *self-play* ao simular partidas sintéticas pelo método *AlphaZero* e, em seguida, utilize-os no alinhamento de pesos e vieses.

O presente trabalho está organizado em seis capítulos. Este Capítulo 1, de Introdução, apresenta o tema geral e a situação de mercado, delimita o problema de pesquisa e descreve a contribuição esperada. O Capítulo 2, de Fundamentação teórica, aborda conceitos fundamentais para a pesquisa segundo a literatura, apresentando estudos que abordam o tema proposto ou correlatos, a fim de situar o presente trabalho no contexto da pesquisa. Por sua vez, o Capítulo 3, de Material e métodos, descreve a metodologia de pesquisa e desenvolvimento da solução proposta. Segue o Capítulo 4, de Desenvolvimento, que apresenta o processo de construção do sistema de representação de jogos e de simulação de partidas sintéticas. Então, o Capítulo 5 descreve a execução de um experimento realizado com o sistema para gerar agentes inteligentes e testar seu uso no processo de coleta de dados de partidas, além de discutir resultados obtidos e os artefatos gerados no processo. Por fim, o Capítulo 6, de Considerações finais, apresenta comentários acerca da pesquisa, suas limitações e as perspectivas para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

A fim de atingir os objetivos propostos, o presente trabalho investiga duas técnicas para a construção de jogadores digitais autônomos para jogos de mesa, sendo elas a MCTS e as ResNets, de acordo com os usos que o *AlphaZero* faz delas. Este capítulo faz a revisão desses métodos, bem como elenca os trabalhos relacionados a uso de agentes inteligentes como ferramentas de *play-test*.

2.1 COMPONENTES FUNDAMENTAIS DE UM JOGO

A descrição de um jogo num ambiente de simulação exige identificar seus componentes fundamentais. Com esse objetivo, a comunidade de projetistas e desenvolvedores de *software* criou o projeto colaborativo BoardGame.io (Boardgame.io developers, 2022), que disponibiliza um ambiente de representação e simulação de jogos de turnos. Ele define uma partida como uma sequência de fases, que estão associadas às regras que definem as ações que os jogadores podem efetuar. Uma fase pode ser constituída por rodadas, em que os jogadores se alternam segundo uma ordem definida pelas regras. A permissão dada a um jogador de realizar uma ou mais ações é chamada de turno, o qual pode ser divido em estágios, similarmente às fases.

O projeto mantém dados mutáveis acerca de um momento da partida por meio de estados e contextos. A manipulação dos estados deve ser descrita pelo projetista do jogo, ao passo em que o contexto de cada turno é gerenciado pela plataforma e salva dados como a quantidade de jogadores e o marcador do jogador atual. Essa atualização dos dados de um estado ao efetuar uma ação é formalmente definida como um movimento, que é implementado como uma função imutável. Isso significa que todas as informações manipuladas por um movimento devem estar no estado que ele recebe como argumento.

2.2 JOGOS DE TURNOS DE DESTAQUE

Alguns jogos de turnos clássicos são utilizados como exemplo ou como método nesta pesquisa e nos trabalhos a ela relacionados. Esta seção contextualiza os seguintes jogos: Jogo da Velha, Ligue-4 e Go, que são jogos de tabuleiro entre dois jogadores com informação completa.

2.2.1 Jogo da Velha

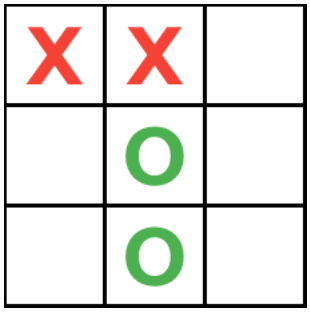
O clássico Jogo da Velha (em inglês, *Tic-tac-toe*)[2] é jogado em um tabuleiro de 3 linhas e 3 colunas, em que os jogadores se alternam marcando peças nas casas disponíveis. Um jogador é considerado vitorioso quando três peças adjacentes por ele colocadas formam uma linha na horizontal ou na vertical ou ainda uma diagonal principal ou secundária no tabuleiro. Um tabuleiro parcialmente preenchido desse jogo está demonstrado na Figura 1a, ao passo em que a Figura 1b mostra o mesmo tabuleiro com uma representação de cada jogador na forma de números inteiros. Nesse caso, o primeiro jogador é salvo como o valor 0 enquanto o segundo é registrado como o número 1.

A fim de adaptar o Jogo da Velha para marcar pontuação dos jogadores, os autores deste trabalho elaboraram um jogo variante chamado *Snowball*. Ele é jogado em um tabuleiro de 9

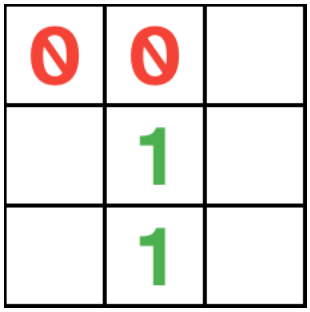
[2] Acesso em https://boardgamegeek.com/boardgame/11901/tic-tac-toe.

Figura 1 — Tabuleiro do Jogo da Velha e sua representação numérica.

(a) Tabuleiro do Jogo da Velha.



(b) Tabuleiro do Jogo da Velha representado numericamente.



Fonte: elaborado pelo autor (2026).
Nota: Neste estado, o jogador atual é o de símbolo “X”.

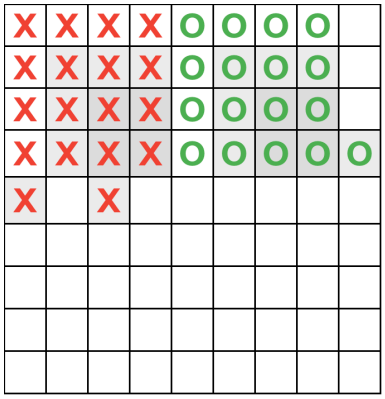
linhas e 9 colunas, de forma que um jogador marca 1 ponto quando a casa em que ele posiciona sua peça forma um formato especificado com as suas casas previamente marcadas.

Os formatos que conferem pontos são uma linha ou diagonal de 5 peças adjacentes, ou ainda um quadrado de dimensão 2 ou 3. Por isso, beneficia-se o jogador que focar seu controle sobre uma região do tabuleiro, porque cada nova casa marcada tende a levar a mais de um formato ao mesmo tempo, o que aumenta a pontuação como uma bola de neve — o que motivou na escolha do nome da variante.

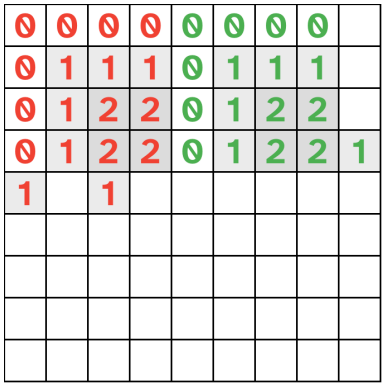
Essa mecânica é demonstrada na Figura 2, em que o jogador “X” fez 15 pontos e o jogador “O” fez 14. A partida chega ao fim quando um dos jogadores marca 15 pontos ou quando 45 dentre as 81 peças do tabuleiro são preenchidas. Então, o jogador com mais pontos é reconhecido como vencedor.

Figura 2 — Tabuleiro do *Snowball* e os pontos atribuídos a cada jogador após efetuar cada jogada.

(a) Tabuleiro do *Snowball*.



(b) Pontuação dos jogadores num estado do *Snowball*.



Fonte: elaborado pelo autor (2026).
Nota: Os jogadores escolheram sequencialmente suas casas da esquerda para a direita e de cima para baixo.

2.2.2 Ligue-4

O Ligue-4 (em inglês, *ConnectFour*)[3] é jogado em um tabuleiro vertical de 6 linhas e 7 colunas, o que resulta em 42 casas disponíveis para marcação. Suas peças são discos de mesmo tamanho divididas igualmente entre cada um dos jogadores, que recebe todas as peças de uma das duas cores disponíveis. Demonstrando as casas marcadas, a Figura 3a representa um tabuleiro parcialmente preenchido, cujo turno atual é do jogador "O". Sua representação numérica considerando a ordem de turnos dos jogadores é exibida na Figura 3b.

Figura 3 — Tabuleiro do Ligue-4 e sua representação numérica.

(a) Tabuleiro do Ligue-4.

		O				
		X	O	O	O	
		O	O	O	X	X
		X	O	X	X	X
	O	X	X	X	O	X

(b) Tabuleiro do Ligue-4 representado numericamente.

		1				
		0	1	1	1	
		1	1	1	0	0
		0	1	0	0	0
	1	0	0	0	1	0

Fonte: elaborado pelo autor (2026).
Nota: Neste estado, o jogador atual é o de símbolo "O".

Dentro de um turno, o jogador atual deve escolher uma coluna que já não tenha sido completamente preenchida para colocar sua peça. Sendo o tabuleiro vertical, ela cairá até a linha mais baixa ainda não preenchida naquela coluna. Após colocada, uma peça não pode mais ser removida naquela partida.

Então, a rodada passa a vez para o segundo jogador, que deve escolher seu movimento da mesma forma que o primeiro. Um jogador vence caso ele posicione 4 de suas peças de forma adjacente na mesma linha, coluna ou diagonal. Configura um empate o caso em que todas as casas tenham sido preenchidas e nenhum jogador tenha marcado um dos formatos especificados. Essas regras fazem com que haja mais de 4.5 trilhões de combinações possíveis de peças no tabuleiro, mesmo que o jogo permita no máximo 7 movimentos em qualquer turno (Cahn, 2024).

2.2.3 Go

O Go[4] é um jogo de estratégia baseados em turnos originado na China. Ele é jogado por duas pessoas, sendo composto por um tabuleiro de 19 linhas verticais e horizontais. Assim, consideram-se casas as intersecções das linhas horizontais e verticais, que totalizam 361. O jogo dispõe de 180 peças brancas e 180 peças pretas, sendo cada cor associada a um dos jogadores. Uma partida se inicia com o tabuleiro vazio e, em cada rodada, os jogadores se alternam colocando uma de suas peças em qualquer intersecção não ocupada. Então, elas não podem mais ser movidas até o fim da partida.

[3] Acesso em https://boardgamegeek.com/boardgame/2719/connect-four.
[4] Acesso em https://boardgamegeek.com/boardgame/188/go.

O objetivo do jogo é cercar totalmente as peças adversárias pois, quando um grupo dessas é totalmente cercado, elas são removidas do tabuleiro. O outro jogador tenta evitar a captura ao posicionar peças em interseções não dominadas pelo oponente. Vence o jogador que, ao se esgotarem todos os movimentos, ainda tiver a maior quantidade de peças dispostas no tabuleiro (Britannica, 2023).

2.3 BUSCA EM ÁRVORE DE MONTE CARLO

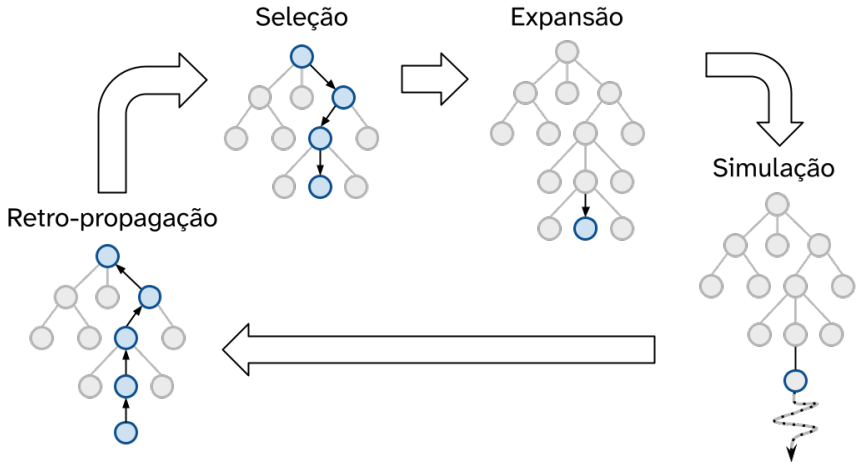
O método de busca em árvore de Monte Carlo (MCTS) é um algoritmo de decisão em que cada nó de uma árvore representa dado estado de um jogo (Coulom, 2006; Kocsis; Szepesvári, 2006). Além disso, cada nó guarda um contador de visitas e um marcador referente à qualidade daquele nó para a partida. Os nós se relacionam por arestas entre nó pai e nó filho. Uma dada aresta representa um movimento tomado por um jogador, que conduz uma transição entre os estados representados.

O nó raiz da árvore de busca é considerado o seu primeiro nível. Esse nó representa o primeiro turno, em que está disposto o estado inicial do jogo. O agente inteligente que opera como o jogador inicial escolhe aleatoriamente um dentre todos os movimentos disponíveis, segundo as regras do jogo. Essa jogada leva à criação de um novo estado, que é colocado no segundo nível da árvore. Para o caso de um jogo entre dois jogadores, o segundo jogador escolherá um dentre os movimentos possíveis. Isso criará um novo estado, que passa o turno novamente para o primeiro jogador. Esse estado é posicionado no terceiro nível da árvore.

Os níveis irão alternadamente representar as jogadas de cada um dos jogadores. Essa estrutura possibilita ao algoritmo jogar como cada um dos jogadores, de forma a explorar o próximo movimento realizado pelo oponente. Dessa forma, o método busca prever a melhor ação futura segundo o histórico disponível a cada iteração (Świechowski et al., 2022).

O processo de busca em árvore de Monte Carlo tem o objetivo de encontrar as melhores sequências de jogadas, que conduzam a uma vitória do jogador. Ele é formado por quatro etapas: seleção, expansão, simulação, e retro-propagação, as quais são representas na Figura 4 (Świechowski et al., 2022, p. 2504).

Figura 4 — Ciclo da busca em árvore de Monte Carlo: suas quatro etapas são a seleção, a expansão, a simulação e a retro-propagação.



Fonte: Adaptado de Świechowski et al. (2022, p. 2504).

A etapa de seleção procura, a partir do nó raiz, o ramo com o melhor nó folha a explorar, orientada por uma diretriz de busca. A mais frequentemente utilizada nas implementações de referência é chamada de limite superior de confiança aplicado a árvores (UCT) — ou *Upper Confidence bounds applied to Trees*, em inglês — (Kocsis; Szepesvári, 2006).

Essa política atribui a cada nó da árvore um contador de visitas e um marcador da qualidade parcial da partida, incrementado conforme o ramo da árvore do qual ele faz parte leva a mais vitórias, ou decrementado caso contrário. Com base nesses dados, a Equação 1 apresenta como o valor de *fitness* (avaliação) de um movimento é calculado. Seu objetivo é alinhar a exploração (*exploration*) e o aproveitamento (exploitation) do espaço de busca.

Equação 1 — Cálculo de *fitness* da diretriz de limite superior de confiança aplicado a árvores (UCT) usada pela busca em árvore de Monte Carlo (MCTS) clássica.

$$m^* = \max_{m \in M(s)} \left(Q(s,m) + C * \sqrt{\frac{\ln(V(s))}{V(s,m)}} \right) \tag{1}$$

Na qual:

- m^* é o nó que representa o movimento ótimo selecionado pela diretriz;
- $M(s)$ é o conjunto de nós que representam os movimentos válidos a partir do estado s, segundo as regras do jogo;
- $Q(s,m)$ é a qualidade da partida calculada por meio de simulações ao jogar o movimento m no estado s;
- $V(s)$ é quantidade de vezes em que o nó que guarda o estado s foi visitado nas iterações anteriores;
- $V(s,m)$ é a quantidade de vezes em que o nó que representa o movimento m foi visitado nas interações anteriores;
- C é o coeficiente que regula a relação entre exploração e aproveitamento.

Fonte: Adaptado de Świechowski et al. (2022, p. 2505).

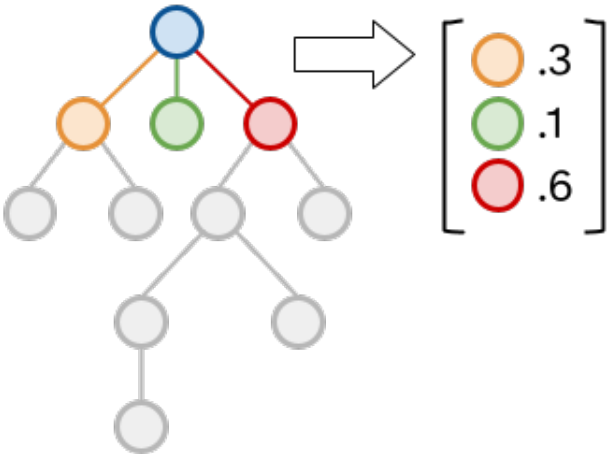
Havendo sido selecionado um nó folha e não sendo este um nó que represente o fim do jogo, então se executa a fase de expansão. Nela, escolhe-se aleatoriamente um movimento dentre aqueles disponíveis para o estado atual segundo as regras do jogo. Então o estado resultante é criado, o qual é armazenado em um novo nó, definido como filho daquele que fora selecionado.

A partir do nó criado, realiza-se a fase de simulação. Nela, sucedem-se turnos entre os jogadores, em que os movimentos são aleatoriamente selecionados. A simulação se encerra quando é atingido um estado que represente o fim da partida. Uma função de *fitness* (avaliação) quantifica a qualidade da partida com o objetivo de aferir a influência do movimento escolhido na pontuação dos jogadores.

Por fim, na fase de retro-propagação, os nós do ramo selecionado são atualizados com os dados gerados. O contador de visitas é aumentado em 1, ao passo em que o marcador de qualidade é incrementado pelo valor de *fitness* calculado.

Para executar o ciclo de busca, deve-se definir o número de iterações desejado. Cada iteração levará à expansão de um único novo nó. Ao final de todos os ciclos, os filhos diretos do nó raiz terão os marcadores de visitas e de qualidade atualizados segundo o andamento das partidas. A partir desses dados, uma função deve calcular a probabilidade de jogar cada um dos

Figura 5 — Uso da busca em árvore de Monte Carlo (MCTS) para calcular as probabilidades de jogar cada um dos movimentos válidos a partir de um estado inicial.



Fonte: Adaptado de Świechowski et al. (2022, p. 2505).
Nota: Neste exemplo, o cálculo das probabilidades dos três movimentos válidos a partir do estado inicial utilizou apenas a quantidade de visitas a cada um dos ramos iniciados pelo respectivo movimento.

movimentos. Um exemplo de função que utiliza somente o contador de visitas a cada ramo para calcular as probabilidades é demonstrado na Figura 5. Dispondo do vetor de probabilidades, o método da seleção aleatória por roleta escolhe um dos movimentos.

A descrição do método de MCTS permite concluir que ele apresenta boas soluções para problemas nos quais o espaço de busca não pode ser percorrido completamente em tempo hábil. Isso se dá porque a política de seleção (UCT) descrita na Equação 1 privilegia os ramos com maior relevância e deixa de gastar recursos explorando aqueles que não tendem a gerar bons resultados. O método também diminui a necessidade de uma heurística prévia sobre o domínio para operar, embora existam trabalhos que buscam defini-la para melhorar o desempenho.

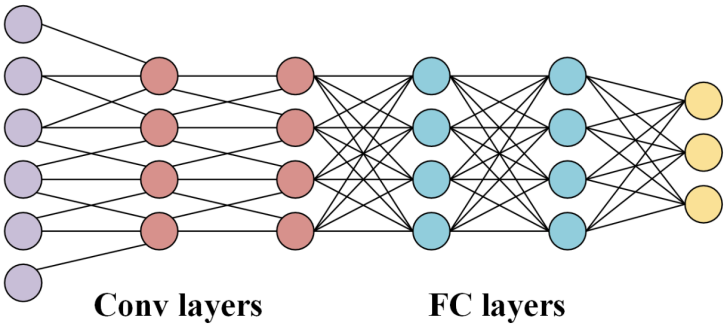
2.4 REDES NEURAIS RESIDUAIS

As redes neurais convolucionais (CNNs) são uma classe de redes neurais profundas especialmente projetadas para processar dados estruturados em grade. Seus usos se destacam na áreas de visão computacional, sobretudo para o reconhecimento de imagens. Aprimorando as redes neurais tradicionais totalmente conectadas, as CNNs utilizam operações de convolução que permitem capturar padrões espaciais e hierárquicos nos dados de entrada sem definição prévia dos elementos de interesse (Li et al., 2022).

A arquitetura típica de uma CNN consiste em camadas convolucionais, camadas de *pooling* (agrupamento) e camadas totalmente conectadas, conforme demonstrado na Figura 6. As camadas convolucionais aplicam filtros que detectam características locais, como bordas e texturas, enquanto as camadas de *pooling* reduzem a dimensionalidade espacial (*downsampling*), preservando as informações mais relevantes (Li et al., 2022), como representado na

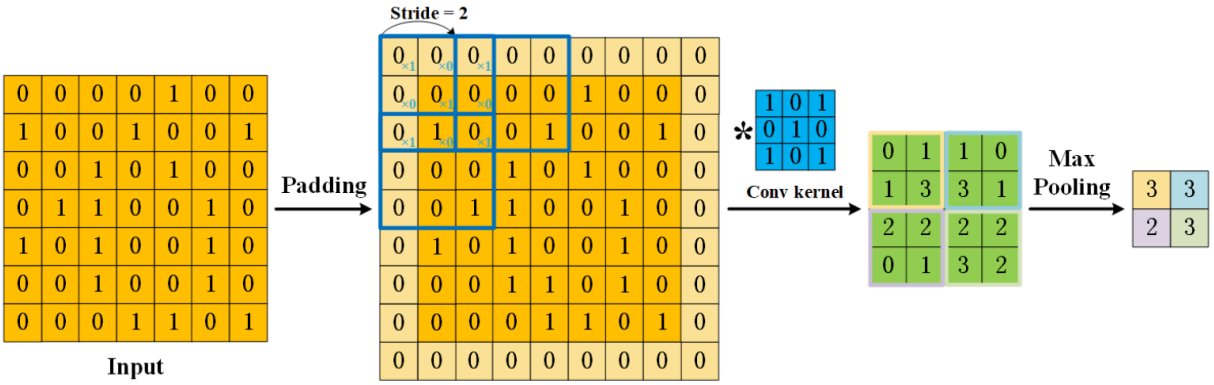
Figura 7. Dessa forma, essa classe de redes neurais balanceia a precisão dos detalhes com a rapidez de convergência pelo processo de *downsampling*.

Figura 6 — Arquitetura dos métodos uma rede neural convolucional (CNN).



Fonte: Li et al. (2022, p. 7000).

Figura 7 — Métodos de processamento de entrada em uma rede neural convolucional (CNN).



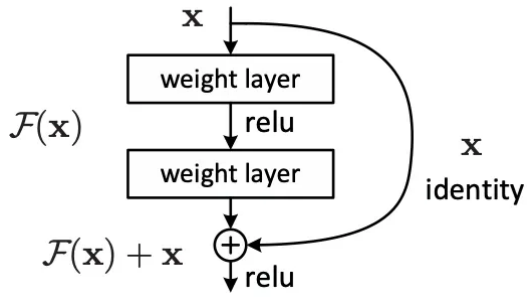
Fonte: Li et al. (2022, p. 7000).

Seguindo os trabalhos na área, He et al. (2015) introduziram as redes neurais residuais (ResNets) como uma evolução importante das CNNs. Seu principal objetivo era resolver o problema de degradação em redes muito profundas. Quando redes neurais convencionais se tornam excessivamente profundas, sua acurácia tende a saturar e depois degradar, não devido ao *overfitting* (sobre-ajuste), mas à dificuldade de otimização (He et al., 2015).

A inovação fundamental das ResNets é a introdução de conexões residuais (*shortcut connections*), que permitem que o gradiente flua diretamente através da rede durante o treinamento (He et al., 2015; Liang, 2020).

Tais conexões são incorporadas em uma estrutura padrão chamada bloco residual, como se pode observar na Figura 8. Em vez de aprender uma transformação direta $H(x)$, cada bloco aprende uma função residual $F(x) = H(x) - x$, onde x é a entrada do bloco. A saída final do bloco é então $F(x) + x$, combinando a transformação aprendida com a entrada original (He et al., 2015). Essa estrutura permite que a rede aprenda transformações incrementais enquanto preserva informações da entrada (Liang, 2020).

Figura 8 — Estrutura de um bloco residual usado em uma rede neural residual (ResNet).



Fonte: He et al. (2015).

O formato de uma ResNet consiste de sucessivos blocos residuais, cada um composto por camadas convolucionais e normalizações, nas quais a função de ativação utilizada é a unidade linear retificada (ReLU). Essa função é não-linear, de forma que retorna exatamente o valor de entrada caso seja positivo, ou retorna 0, caso seja negativo, como detalhado nos trabalhos de Nair; Hinton (2010). Essa arquitetura possibilita a construção de redes extremamente profundas mantendo alta precisão e facilitando o treinamento (He et al., 2015).

2.5 PROJETO *ALPHAZERO*

O laboratório DeepMind, que é um braço de pesquisa em inteligência artificial (IA) da Google, visava a criar um jogador autônomo para o Go a nível competitivo. Para atingir esse objetivo, seus pesquisadores desenvolveram o método de construção de modelos de redes neurais chamado de *AlphaGo*. Essa versão gerava dados para treinar o modelo ao colocá-lo para jogar contra jogadores humanos.

Foi então desenvolvida sua evolução, chamada de *AlphaGo Zero*, que acumula dados de treinamento jogando contra si mesma, no que se define como *self-play* (autoaprendizado por simulação de partidas). Um novo modelo construído é iniciado com pesos (*weight*) e vieses (*bias*) aleatórios, o que leva a movimentos arbitrários. Ainda assim, a massa de dados gerada permite identificar quais estados levaram a melhores avaliações pela função de *fitness* (Silver et al., 2016).

Dessa forma, por meio de treinamentos e geração de dados sucessivos, o modelo tende a alcançar desempenho excepcional. Esse processo de lapidação dos pesos e vieses por meio de *self-play* é compreendido como um método de aprendizado por reforço (Silver et al., 2017).

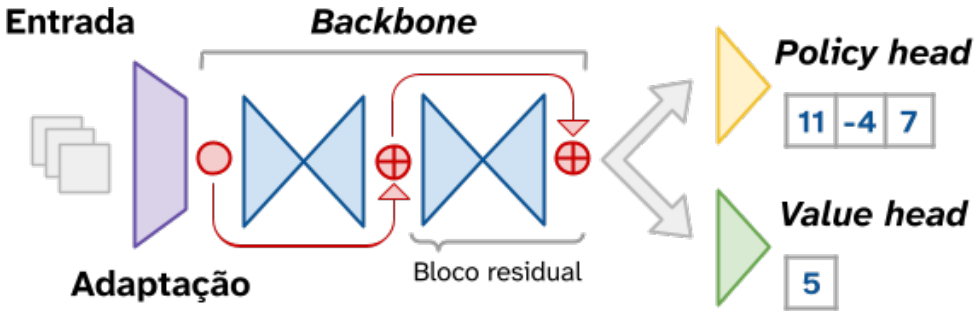
O método foi então generalizado para permitir a criação de modelos capazes de aprender qualquer jogo de tabuleiro dadas apenas as suas regras, ao que se denominou *AlphaZero*. Os principais destaques foram os jogos Go, Shogi e Xadrez (Silver et al., 2018).

Um dos objetivos do método *AlphaZero* é reduzir o custo computacional de agentes inteligentes que atuam como jogadores. Essa preocupação se torna mais evidente ao considerar a complexidade das árvores de busca para jogos que apresentam muitos movimentos. Com esse foco, os pesquisadores propuseram substituir as buscas por modelos de inteligência artificial baseados em redes neurais. Em vez de simular uma partida para calcular a qualidade de cada

movimento, o agente inteligente pode solicitar uma predição a um modelo de ResNet previamente treinado para aquele jogo.

A arquitetura da ResNet aplicada no *AlphaZero* é representada na Figura 9. Ela se inicia pela recepção do estado do jogo cujos movimentos viáveis se deseja analisar. Esse estado passa por uma camada de adaptação, que transforma a entrada em um formato adequado para realizar as sucessivas convoluções. Em seguida, inicia-se a construção da cadeia profunda de blocos residuais, ao que se denomina *backbone*. Por fim, a rede neural duplica o tensor em processamento para gerar duas saídas.

Figura 9 — Arquitetura de uma rede neural residual (ResNet) composta por uma camada de adaptação da entrada, uma *backbone* e camadas de saída *policy head* e *value head*.



Fonte: elaborado pelo autor (2026).

A primeira saída é construída pela camada de *policy head*, que retorna um vetor de números reais. Esses valores representam a qualidade atribuída a cada um dos movimentos válidos a partir do estado fornecido. Na verdade, devido à restrição de formato da saída da rede, o modelo atribuirá uma classificação para todos os movimentos possíveis de acordo com as regras do jogo, sendo estes válidos ou não a partir do estado atual. Dessa forma, é necessário que o *designer* do jogo simulado descreva previamente a lista de todos os movimentos e os guarde em um vetor. O algoritmo do agente inteligente indexará as posições deste àquelas do vetor retornado pela rede.

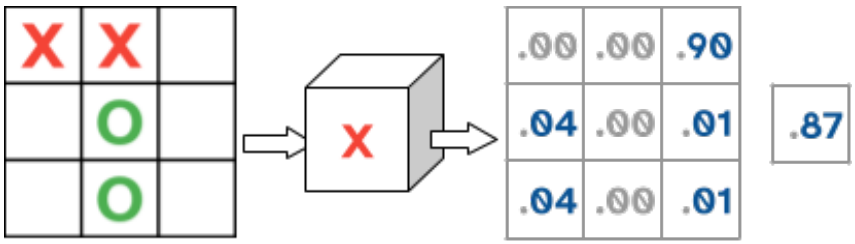
A segunda saída da ResNet é construída pela camada de *value head*. Seu retorno é um valor escalar que representa a estimativa da qualidade do resultado da partida a partir do estado fornecido. Esse valor será maior para quando houver uma expectativa de vitória e menor para quando a expectativa for de derrota.

Esses retornos são exemplificados pela Figura 10, que utiliza valores fictícios. O exemplo considera um estado vantajoso no Jogo da Velha para o jogador “X” que será o próximo a jogar. O primeiro retorno se refere às qualidades atribuídas pela *policy head*[5]. As casas já preenchidas por peças têm qualidade 0 atribuída, uma vez que nelas não são permitidos mais movimentos. A casa no canto superior direito, que pode ser marcada pelo terceiro movimento, apresenta uma qualidade de 0.9, uma vez que sua marcação levaria à vitória imediata do jogador “X”. As demais casas apresentam qualidades pouco significativas. Além disso, a figura também

[5]Para fins de melhor visualização consideramos que os valores de qualidade foram transformados em probabilidades. O retorno da rede neural na verdade é composto por valores reais não normalizados. No algoritmo, eles devem passar por uma função de *softmax* para poderem ser sorteados pelo método da roleta.

mostra a forma de retorno da estimativa de qualidade da partida, dada pela *value head*. Uma vez que o estado analisado está a um movimento de levar à vitória, a probabilidade de vitória se mostra alta.

Figura 10 — Predição de um modelo de rede neural residual (ResNet) para as qualidades estimadas de cada movimento do jogo e para a expectativa de qualidade da partida a partir de um estado do tabuleiro no turno do jogador “X”.



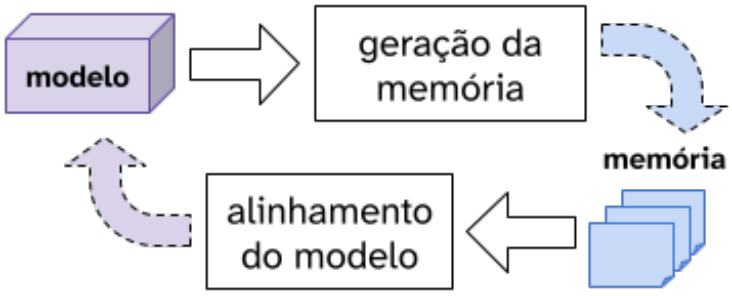
Fonte: elaborado pelo autor (2026).
Nota: As predições de qualidade são representadas como probabilidades para facilitar a visualização, mas seus valores são números reais sem normalização.

O processo de treinamento de um modelo é feito em duas fases. A primeira se denomina fase de geração de memória de treinamento, que utiliza a técnica de *self-play*. Ela constrói um histórico de partidas que guarda, para cada partida, a pontuação final dos jogadores e a sequência de turnos e seus estados. No caso de jogos sem cálculo de pontuação, como o Jogo da Velha ou Xadrez, o resultado final será de 1 ponto para o vencedor e 0 pontos para o perdedor (Świechowski et al., 2022, p. 2533).

Segue-se então a fase de alinhamento do modelo, que utiliza aprendizado de máquina (*machine learning*) para ajustar os pesos e vieses. Para isso, o conjunto de dados gerado é convertido em conjuntos de entradas e de saídas esperadas, que são fornecidos para um algoritmo de treinamento. Espera-se que o modelo resultante possa gerar uma memória de partidas mais significativa que o anterior. Assim, entende-se o treinamento como um ciclo, conforme demonstrado na Figura 11.

É interessante que, durante a fase de geração de memória de treinamento, o agente inteligente tenha alguma orientação sobre quais movimentos levam a melhores jogadas. Para esse objetivo, o método de MCTS se mostrou útil. Para otimizar sua aplicação, o método *AlphaZero*

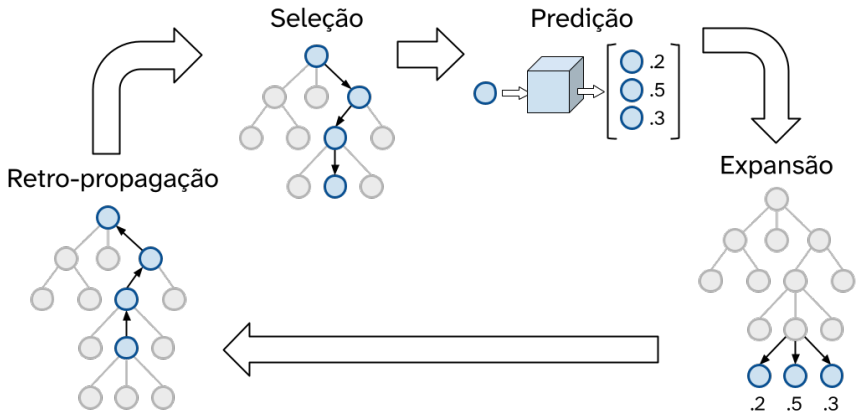
Figura 11 — Ciclo de treinamento de um modelo do *AlphaZero*, constituído das fases de geração da memória de partidas e de alinhamento do modelo de rede neural residual (ResNet).



Fonte: elaborado pelo autor (2026).

removeu a etapa de simulação do ciclo de busca. Em vez dela, a etapa de predição solicita à ResNet uma estimativa da qualidade dos movimentos e da qualidade da partida, como mostrado na Figura 12.

Figura 12 — Ciclo da busca em árvore de Monte Carlo guiada por agentes inteligentes, conforme adaptação do *AlphaZero*: suas quatro etapas são a seleção, a predição, a expansão e a retro-propagação.



Fonte: elaborado pelo autor (2026).

Outra alteração se dá na fase de expansão. No método adaptado, em vez de expandir um único movimento por iteração e avaliar seu resultado, a MCTS guiada por agente inteligente expande todos os movimentos viáveis a partir do estado atual. Para cada nó gerado, ela incrementa o contador de visitas e define um novo marcador de qualidade do movimento, o qual é preenchido com a estimativa de qualidade dada pela rede para o movimento que gera aquele nó.

Sem que haja uma simulação da partida, não seria possível realizar a retro-propagação, uma vez que ela depende da análise da pontuação final dos jogadores. Para adaptar essa questão, a retro-propagação é realizada a partir do nó selecionado e não mais a partir do filho expandido. O valor de qualidade da partida utilizado como referência é aquele fornecido pela rede.

Uma exceção a esse ciclo se dá quando o estado selecionado pela iteração atual representa o fim do jogo. Nesse caso, não se realiza predição nem expansão. Em vez disso, a pontuação dos jogadores é utilizada para calcular a qualidade da partida segundo a perspectiva do jogador do turno atual. Então, a retro-propagação é feita a partir desse nó terminal com base na qualidade calculada.

A definição do novo marcador de qualidade em cada nó é relevante para realizar o cálculo de uma diretriz de *fitness* adaptada, como demonstrada na Equação 2. A UCT passa a considerar como componente de aproveitamento apenas a qualidade da partida simulada pelas iterações. Já como componente de exploração, a política alinha dois fatores: como numerador, a predição do modelo para o sucesso do movimento representado; e como denominador, a quantidade de visitas realizadas ao nó resultante da aplicação do movimento, que é somada ao número 1 para garantir que o resultado não seja indefinido.

Equação 2 — Cálculo de *fitness* da diretriz de limite superior de confiança aplicado a árvores (UCT) usada pela busca em árvore de Monte Carlo (MCTS) adaptada pelo *AlphaZero*.

$$m^* = \max(m \in M(s)) = Q(s,m) + X(s,m) \tag{2.1}$$

$$X(s,m) = C \times \frac{P(s,m)}{V(s,m)+1} \tag{2.2}$$

Na qual:

- m^* é o nó que representa o movimento ótimo selecionado pela diretriz;
- $M(s)$ é o conjunto de nós que representam os movimentos válidos a partir do estado s, segundo as regras do jogo;
- $Q(s,m)$ é a qualidade da partida calculada por meio de simulações ao jogar o movimento m no estado s;
- $X(s,m)$ é o componente de exploração (*exploration*) calculado ao jogar o movimento m no estado s;
- $V(s)$ é quantidade de vezes em que o nó que guarda o estado s foi visitado nas iterações anteriores;
- $V(s,m)$ é a quantidade de vezes em que o nó que representa o movimento m foi visitado nas interações anteriores;
- $P(s,m)$ é a qualidade previamente atribuída pelo modelo de ResNet para jogar o movimento m no estado s;
- C é o coeficiente que regula a relação entre exploração e aproveitamento.

Fonte: Adaptado de Silver et al. (2016, p. 486); Świechowski et al. (2022, p. 2505).

É relevante considerar como a MCTS utilizada pelo *AlphaZero* representa um estado do jogo. Cada casa do tabuleiro guarda a informação sobre a peça marcada em si e o jogador que a posicionou. O tabuleiro é salvo atribuindo um número a cada um dos jogadores, que pode ser indexado pela lista de jogadores definida previamente pelo *designer* do jogo. Essa representação foi brevemente discutida na Subseção 2.2.1, em que a Figura 1 mostra como o tabuleiro do Jogo da Velha na Figura 1a é codificado em um estado na Figura 1b. Nessa forma, o primeiro jogador, de símbolo "X", é representado pelo número 0, ao passo que o segundo jogador, de símbolo "O", é representado pelo número 1. As posições sem peças são definidas com o valor `null`. Outra informação armazenada no estado é um marcador de qual jogador deve jogar no turno atual, o que é feito pelo uso dos mesmos índices da ordem dos jogadores.

A entrada da ResNet utilizada pelo agente inteligente requer que o estado seja codificado como uma pilha de canais que contêm apenas valores binários (0 ou 1). Essa técnica busca aproximar a representação do tabuleiro daquela usada por imagens RGB, comumente fornecidas como entrada a ResNets de reconhecimento de imagens.

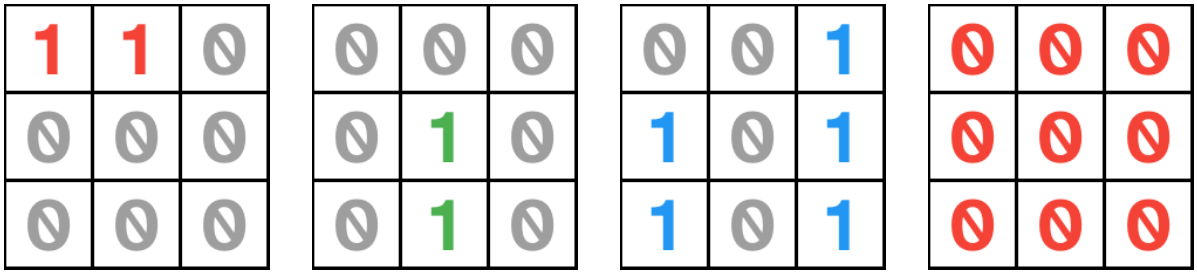
No exemplo do Jogo da Velha, o tabuleiro representado na Figura 1b se torna um conjunto de três canais, como disposto na Figura 13. O primeiro, associado à cor vermelha, tem uma posição ativada quando o primeiro jogador (representado pelo símbolo "X") posiciona nela uma peça, como mostrado na Figura 13a. Similarmente, o segundo canal, associado à cor verde, representa as casas marcadas pelo segundo jogador (representado pelo símbolo "O"),

como mostrado na Figura 13b. Por fim, as casas vazias são representadas no terceiro canal, associado à cor azul, como mostrado na Figura 13c.

Caso necessário, outras informações podem ser representadas por meio da adição de novos canais à pilha. Os jogos de tabuleiro para dois jogadores citados requerem a representação de qual jogador deve executar um movimento no turno atual. Isso é definido em um quarto canal, cujas posições são marcadas com o número atribuído ao jogador, como mostrado na Figura 13d. Assim, um estado do Jogo da Velha define todo esse canal como 0 para o jogador de símbolo “X”, e como 1 para o jogador de símbolo “O”.

Figura 13 — Estado do Jogo da Velha representado como canais binários.

(a) Canal do jogador “X”. (b) Canal do jogador “O”. (c) Canal de casas vazias. (d) Jogador atual.

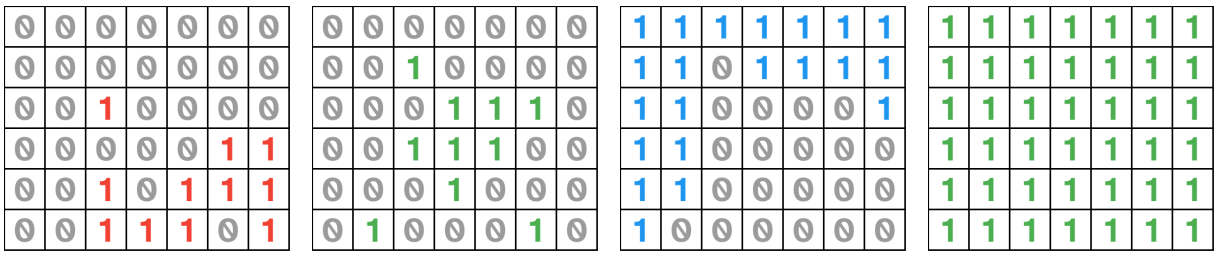


Fonte: elaborado pelo autor (2026).

Similarmente, a codificação do jogo Ligue-4 pode ser realizada em quatro canais. Essa forma é exemplificada na Figura 14, em que os canais mostram o resultado da codificação do estado mostrado na Figura 3, discutido na Subseção 2.2.2. os dois primeiros codificam as casas marcadas por cada um dos jogadores, o terceiro representa as casas vazias e o quarto indica que o jogador do turno atual é o de símbolo “O”.

Figura 14 — Estado do Ligue-4 representado como canais binários.

(a) Canal do jogador “X”. (b) Canal do jogador “O”. (c) Canal de casas vazias. (d) Jogador atual.



Fonte: elaborado pelo autor (2026).

2.6 TRABALHOS RELACIONADOS

Os autores deste trabalho buscaram pesquisas relacionadas à área de estudo em dois campos de interesse. No primeiro foco, elencam-se estudos acerca do uso de agentes inteligentes na criação e avaliação de jogos. Essa perspectiva visa a verificar o andamento da proposta de realizar a fase de *play-test* de forma automatizada, considerando seus métodos e parâmetros de interesse. Em seguida, os autores buscaram elencar estudos acerca da visualização de dados analíticos sobre partidas de uma forma efetiva para o usuário dos sistemas.

Zook; Fruchter; Riedl (2019) reforçam as vantagens da substituição de jogadores humanos em partes bem específicas do processo de *play-test*. O principal destaque é no ajuste de parâmetros e de dificuldade quando os sistemas do jogo já estão definidos mas se busca uma melhor experiência para o público alvo do jogo.

Ademais, os autores desenvolvem um estudo combinando técnicas de regressão e classificação para realizar uma aprendizagem ativa (Cohn; Atlas; Ladner, 1994) de um jogo *shoot'em up*. A mecânica desse jogo é bem definida, mas os parâmetros — como velocidades de jogador, inimigos e tiros — são ajustados através de testes exaustivos. Nesse trabalho, eles foram substituídos pelo *play-test* automatizado.

Nos trabalhos de Gudmundsson et al. (2018); Zook; Fruchter; Riedl (2019), a MCTS é utilizada junto a redes neurais convolucionais (CNNs). Elas são treinadas através de um massivo conjunto de dados de jogadores reais para prever a dificuldade de missões em jogos digitais *match-3* — respectivamente *Candy Crush* e *Jewels Star Story*. Neste tipo de jogo, o jogador deve mover figuras em uma grade, buscando colocar três ou mais figuras iguais adjacentes, que são retiradas do tabuleiro e podem gerar outras remoções em cadeia. Os trabalhos conseguem reproduzir comportamentos de jogadores humanos e avaliar a dificuldade do nível proposto pelo *game designer* para uma melhor experiência de jogo.

Sob a ótica de comunicação dos dados gerados ao *designer*, Wallner; Halabi; Mirza-Babaei (2019) desenvolveram um sistema para traçar, em jogos digitais de plataforma, a trajetória de dados de partidas colhidas diretamente sobre os mapas do jogo. Ele integra dados de fontes diferentes em uma única visualização capaz de representar o *feedback* dado pelos jogadores, suas medidas fisiológicas colhidas e a rastreabilidade dos movimentos em jogo.

Esses dados fisiológicos relacionados ao estímulo do jogador são visualizados de forma intuitiva pela sua representação em mapas de cor. O espaço do jogo é dividido em regiões, de forma que a movimentação por linhas que conectam essas regiões têm sua opacidade e espessura relacionadas à frequência. Ademais, os eventos discretos são agrupados em ícones cujo tamanho é relacionado à sua frequência, relatando observações de comportamentos durante a partida. A abordagem diminui a poluição visual, compila um grande conjunto de informações e provê um grande valor para avaliar um cenário em desenvolvimento.

Similarmente, Stahlke; Nova; Mirza-Babaei (2020) investigam técnicas de representação de dados em jogos em três dimensões, apresentando os caminhos sobre superfícies para auxiliar no processo de projeto dos níveis. Registra-se também o uso de agentes para o projeto ou validação da economia interna dos jogos, mostrado nos resultados iniciais de Ranandeh; Mirza-Babaei (2023).

Apesar de os trabalhos de testes serem em sua maioria referentes a jogos digitais, que são normalmente modelados sistemas em tempo contínuo, acreditamos que as mesmas técnicas podem ser aplicadas a jogos físicos e modelados por sistemas discretos.

3 MATERIAL E MÉTODOS

O presente trabalho se classifica como uma pesquisa de natureza aplicada acerca do uso de agentes inteligentes para realizar a fase de *play-test* em jogos de tabuleiro. Ela busca, através de uma abordagem qualitativa, aplicar os métodos utilizados no projeto *AlphaZero* para criar um sistema de representação de jogos e simulação de partidas sintéticas para gerar dados de apoio ao balanceamento. A pesquisa também é exploratória, pois permitirá aumentar familiaridade acerca da modelagem de jogos e seus mecanismos com os métodos de aprendizagem profunda para uso como ferramentas de projeto.

3.1 MATERIAL

Dando continuidade aos trabalho desenvolvidos em Araki; Knop (2020); Malosto; Campos; Knop (2025); Malosto; Knop; Campos (2023), foi desenvolvida neste trabalho a aplicação de linha de comando chamada **APTS**, capaz de representar jogos discretos e gerar agentes inteligentes que simulem partidas conforme o método de *self-play*. As simulações coletam dados sobre as partidas para prover ao projetista do jogo informações estatísticas usadas para orientar testes de estresse e de balanceamento, que focam em aspectos técnicos em vez de tratar da experiência do jogador.

3.1.1 Ambiente de execução

Os autores têm a expectativa de que o APTS possa ser acessado por meio de programas navegadores da internet, dispondo de uma interface de usuário satisfatória para usuários não familiarizados com programação. Entretanto, concluiu-se que seria vantajoso desenvolver *scripts* de teste de *software* para verificar sua qualidade durante as versões iniciais de desenvolvimento. Por isso, estabeleceu-se como requisito que o sistema funcionasse como uma biblioteca, de forma que possa ser utilizado tanto por um programa de linha de comando, como também por uma página da *web*.

Com essa perspectiva, escolhemos escrever o código-fonte do sistema na linguagem de programação JavaScript. Essa é utilizada comumente para o desenvolvimento de páginas da *web*, tendo suporte oferecido pelos principais navegadores. Essa linguagem também pode ser utilizada em um ambiente de execução de linha de comando, sendo o mais comum o Node.js. Ele utiliza o motor de JavaScript V8, o que aprimora o desempenho dos programas ao compilar o código-fonte na forma de *Just-In-Time* (JIT). Apesar de rodar em apenas uma *thread*, o ciclo de processamento trata eventos assíncronas por meio de operações primitivas (Node.js, 2025).

3.1.2 Ambiente de desenvolvimento

O ambiente de desenvolvimento do projeto foi configurado utilizando o gerenciador de pacotes PNPM[6]. Ele instala e mantém atualizadas as ferramentas citadas e suas dependências por meio do registro de pacotes NPM[7].

[6] Acesso em: https://pnpm.io/motivation.
[7] Acesso em: https://www.npmjs.com/.

A fim de evitar enganos de programação, utilizamos um superset do JavaScript chamado TypeScript, que permite atribuir tipos estáticos e mais complexos a variáveis e funções. Isso assegura a compatibilidade entre elas ainda em tempo de compilação (TypeScript Team, 2026).

Outra ferramenta de inspeção de código-fonte utilizada é o ESLint (ESLint contributors, 2025) e sua extensão typescript-eslint[8]. Esse programa é um *linter*, que encontra e corrige problemas no código-fonte segundo os padrões e regras configurados. Associamos essa ferramenta ao formatador automático de código-fonte Prettier[9] com o fim de padronizar a disposição de importações e de atributos de classes, funções, objetos, e demais estruturas.

A fim de arquitetar o APTS como uma biblioteca modular, utilizamos o sistema de construção Turborepo[10]. Ele divide um repositório em pacotes, cada um com suas dependências. Um pacote pode ter dependência em outro dentro do mesmo repositório, o que permite construir um sistema complexo, mas composto por partes simples. De acordo com as relações inter-módulos, o Turborepo gerencia a compilação e a execução do *linter* de forma independente e faz cache dos resultados quando possível.

Finalmente, utilizamos a biblioteca de testes de unidade Vitest[11]. Ela permite definir casos de teste e executá-los para entradas variadas, o que se provou útil sobretudo para garantir que as regras dos jogos modelados de fato levem às alterações esperadas nos estados.

3.1.3 Dependências externas

A construção do sistema requereu o uso de bibliotecas e demais pacotes externos instalados por meio do registro NPM. A biblioteca de maior destaque é a implementação em JavaScript[12] do projeto TensorFlow (Abadi et al., 2016). Ele foi desenvolvido pelo time de pesquisa da empresa Google e se propõe a facilitar a construção e o treinamento de modelos de aprendizado de máquina. Os autores deste trabalho selecionaram-no para construir dinamicamente ResNets em JavaScript, ao passo em que o processamento efetivo do treinamento é descrito internamento pela linguagem C++.

Com o objetivo de tornar a execução do programa construído o mais determinística possível, os autores utilizaram a biblioteca `seedrandom`. Isso foi necessário porque a função disponibilizada pela linguagem JavaScript para gerar números pseudo-aleatórios não permite ao desenvolvedor definir uma *seed*.

Outro pacote utilizado foi o ts-graphviz[13], que disponibiliza uma interface de programação de aplicações (API) para o uso do programa Graphviz[14], em conjunto com uma aplicação em JavaScript. Esse projeto descreve uma linguagem de representação de grafos e redes e oferece algoritmos que geram imagens a partir das descrições. Os autores o utilizaram para exibir ao usuário as árvores de busca construídas ao executar o método de MCTS.

[8] Acesso em: https://typescript-eslint.io/.
[9] Acesso em: https://prettier.io/.
[10] Acesso em: https://turborepo.com/docs.
[11] Acesso em: https://vitest.dev/guide/.
[12] Acesso em: https://www.tensorflow.org/js.
[13] Acesso em: https://ts-graphviz.github.io/.
[14] Acesso em: https://graphviz.org/.

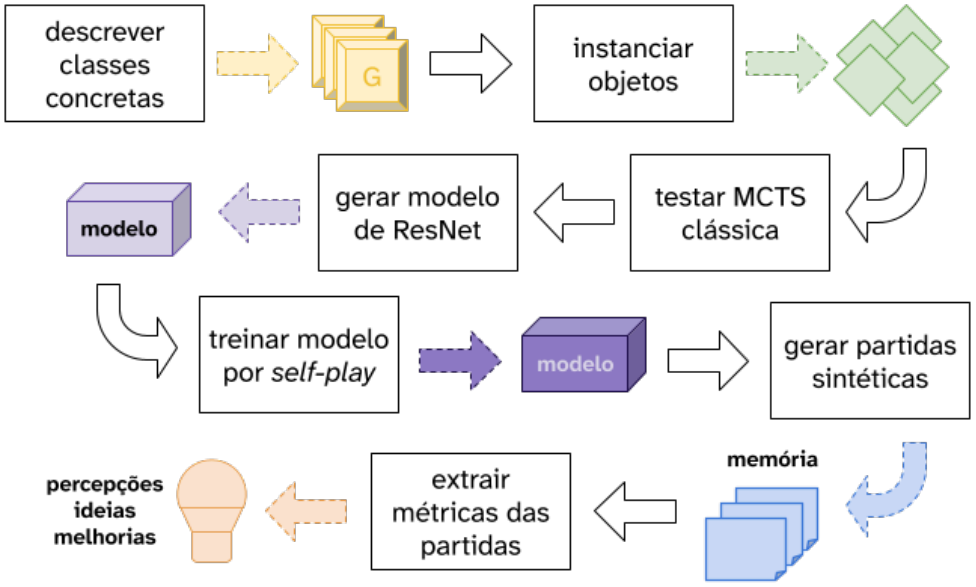
Finalmente, para elaborar a aplicação de linha de comando, os autores dispuseram da biblioteca Commander.js[15], que facilita a definição de comandos e argumentos. Ela gerencia o tratamento de dados recebidos do terminal e exibe mensagens de auxílio ao usuário sobre como preenchê-los. Já para permitir ao usuário selecionar dentre opções de interface já dentro da execução de um comando, os autores escolheram a biblioteca Inquirer.js[16].

3.2 MÉTODOS

Os métodos dessa pesquisa descrevem os passos que o pesquisador deve efetuar para executar o experimento. As atividades desempenhadas e os artefatos por elas gerados gerados são representados na Figura 15.

O primeiro requisito para executar a plataforma APTS é descrever por meio de classes concretas e suas consequentes instâncias todos os componentes fundamentais de um jogo. Então, o pesquisador poderá simular partidas por meio do algoritmo de MCTS clássico.

Figura 15 — Fluxo de trabalho dos métodos necessários e seus artefatos.



Fonte: elaborado pelo autor (2026).

A fim de construir os agentes inteligentes conforme o método do projeto *AlphaZero*, o usuário deve gerar, para aquele jogo, um modelo de ResNet que tenha pesos e vieses aleatórios. Esse processo exporta a rede em arquivos que definem sua estrutura e seus pesos e vieses. Então, esse modelo precisa passar por um processo de treinamento em ciclos, contando com coleta de partidas sintéticas e alinhamento das conexões da rede neural.

A primeira etapa do ciclo de treinamento é executar um algoritmo de *self-play* que usa a técnica de MCTS adaptada pelo *AlphaZero* para direcionar a simulação de várias partidas. Nesse processo, é gerado um artefato que guarda dados relevantes da atuação dos jogadores durante as partidas. Entre eles estão a sequências de turnos, em que cada um guarda o estado do jogo, a expectativa dada pela ResNet da qualidade de cada movimento possível e o movimento que o agente inteligente de fato tomou. Além disso, para cada partida, é salva pontuação final

[15] Acesso em: https://github.com/tj/commander.js.
[16] Acesso em: https://github.com/SBoudrias/Inquirer.js/.

dos jogadores, o que permite avaliar se a tomada de uma decisão em certo estado levou a uma vitória ou não.

O passo seguinte do ciclo de treinamento é fornecer o conjunto de dados sintéticos construído para um algoritmo que utiliza a técnica de aprendizado de máquina para reforçar as conexões da rede neural. Esse processo tem o objetivo de capacitar a rede neural a prever movimentos mais adequados para um estado fornecido. Ao final do alinhamento de pesos e vieses, o algoritmo exporta como artefato o novo modelo de ResNet. Então, essa rede treinada pode voltar ao primeiro passo do ciclo para gerar mais um conjunto de memórias, agora mais especializadas.

Após dispor de modelos de ResNet suficientemente treinados, o APTS deve permitir que seu usuário os utilize para orientar agentes inteligentes na simulação de partidas. Elas devem salvar os mesmos artefatos de registro de histórico, que podem ser usados para extrair dados relevantes sobre a atuação de cada jogador.

Espera-se que esse processo seja capaz de levantar informações comuns à fase de *play-test*, mas reduzindo a necessidade de testadores humanos. Dessa forma, foi determinado como foco do experimento realizado nesta pesquisa: verificar se o processo de treinamento de modelos de inteligência artificial é capaz de gerar agentes inteligentes viáveis para realizar a etapa de *play-test* na prototipagem de jogos.

Para executar o experimento, os autores deste trabalho representaram no sistema o jogo Ligue-4, que é organizado em turnos e apresenta informação completa. Em seguida, geraram uma ResNet compatível com o jogo, e a sujeitaram a 21 ciclos de treinamento. Dentre os modelos criados, os autores selecionaram o que apresentava as melhores métricas de acurácia segundo determinado pelo algoritmo de alinhamento de pesos e o utilizaram para orientar ambos os jogadores.

Então, esses agentes inteligentes foram usados na simulação de 100 partidas, cujo histórico foi salvo da mesma forma como os artefatos utilizados no ciclo de treinamento. Por meio de um algoritmo, os autores extraíram informações de interesse dos históricos e as compilaram em um artefato final. Esse descreve métricas acerca: (1) da duração das partidas, medida em quantidade de turnos; (2) da distribuição de movimentos mais escolhidos por cada jogador; e (3) da contagem de vitórias e derrotas de cada jogador relacionada à duração da partida.

A avaliação da solução proposta foi realizada de forma qualitativa por meio da análise e discussão sobre a capacidade de os artefatos gerados expressarem conclusões relevantes acerca do jogo testado. Além disso, também foi avaliada a capacidade do sistema de representar o Ligue-4 e de gerar agentes inteligentes que o simulem.

4 DESENVOLVIMENTO

Este capítulo descreve o desenvolvimento do sistema APTS, realizado como um projeto de código-livre em um repositório público[17] (Malosto; Knop, 2026). Essa aplicação permite a uma pessoa projetista de um jogo de tabuleiro descrever as regras de um protótipo de jogo. Então, o programa oferece métodos para gerar e treinar modelos de inteligência artificial que atuam como agentes inteligentes para simular partidas.

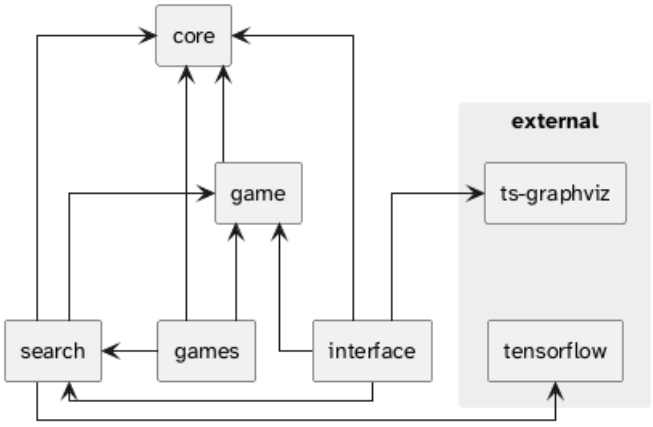
As simulações geram conjuntos de dados acerca de quais movimentos tomados levam a melhores resultados. Espera-se que, por meio deles, o projetista possa gerar informações estatísticas acerca das regras implementadas. Isso tem o objetivo de diminuir o esforço humano nas etapas de *play-test*, sobretudo aquelas que envolvem testes de estresse e balanceamento, em que a experiência do jogador não é a variável principal.

O projeto da aplicação desenvolvida a divide em cinco módulos, quais sejam: `core`, `game`, `search`, `games` e `interface`. A Figura 16 representa as relações de dependência entre tais módulos e com os pacotes externos `ts-graphviz` e `tensorflow`. Esta seção discute a responsabilidade e a implementação de cada um dos módulos internos.

4.1 UTILITÁRIOS

O módulo `core` tem a responsabilidade de definir constantes, tipos e funções utilitárias para todos os demais módulos. Destacam-se algumas funções de conversão de tipos de dados, sobretudo para tratar argumentos fornecidos pela linha de comando em suas representações numéricas. Também estão disponíveis utilitários para a formatação de dados de tipos compostos e de descritores dos testes de unidade. Além disso, o módulo gerencia a codificação de dados para o formato de notação de objetos do JavaScript (JSON)[18] e a equivalente conversão para objetos em memória, o que é necessário para salvar e interpretar o histórico de partidas para o treinamento de modelos.

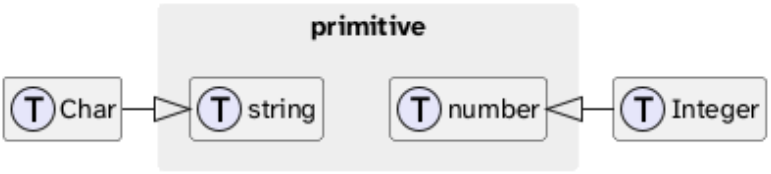
Figura 16 — Dependências entre os módulos do sistema e com pacotes externos.



Fonte: elaborado pelo autor (2026).
Nota: Um módulo aponta para o pacote do qual ele depende.

[17] Acesso em: https://github.com/ufjf-gamelab/apts.
[18] Acesso em: https://www.json.org/json-en.html.

Figura 17 — Tipos de dados comuns definidos pelo pacote `core`.



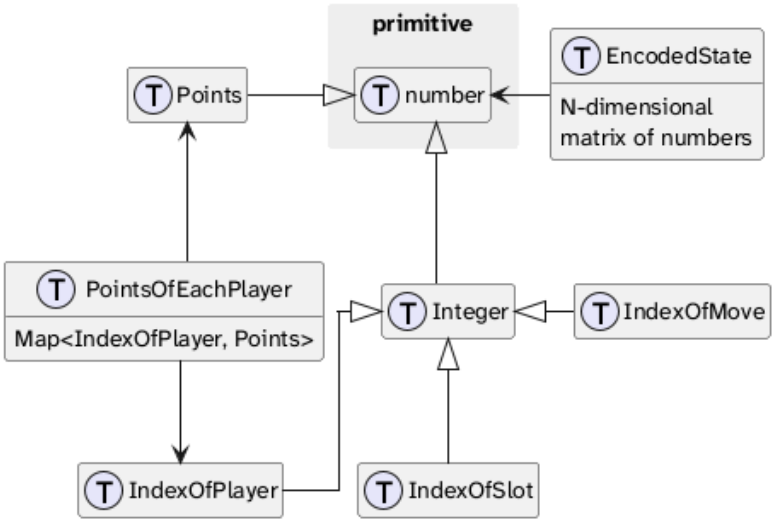
Fonte: elaborado pelo autor (2026).
Nota: O pacote `primitive` se refere aos tipos de dados concretos disponibilizados pela linguagem JavaScript.

A fim de facilitar a compreensão de conceitos comuns ao domínio da aplicação, definimos por meio do TypeScript alguns tipos derivados, utilizados por todo o projeto. Os principais estão dispostos na Figura 17. Ela explicita os tipos concretos `string` e `number` da linguagem JavaScript, que guardam, respectivamente, texto e números reais. O tipo `Char` foi um apelido (em inglês, *alias*) dado para campos de texto de apenas um caractere, como a marcação de uma peça em uma casa do tabuleiro. Já o tipo `Integer` é um apelido para um valor numérico que deve ser preenchido apenas por um número inteiro, como por exemplo na indexação de dados em vetores.

4.2 DESCRIÇÃO DE JOGOS

Seguindo a descrição modular do sistema, o módulo `game` é responsável por estabelecer os componentes necessários para descrever um jogo de turnos digitalmente. Primeiramente, definimos tipos úteis para esse pacote e para seus dependentes, como apresentado na Figura 18. Uma vez que utilizamos vetores extensamente pelo projeto, decidimos criar apelidos para nomear os índices de movimentos (moves), de casas (slots) e de jogadores (players).

Figura 18 — Tipos de dados comuns definidos pelo pacote `game`.



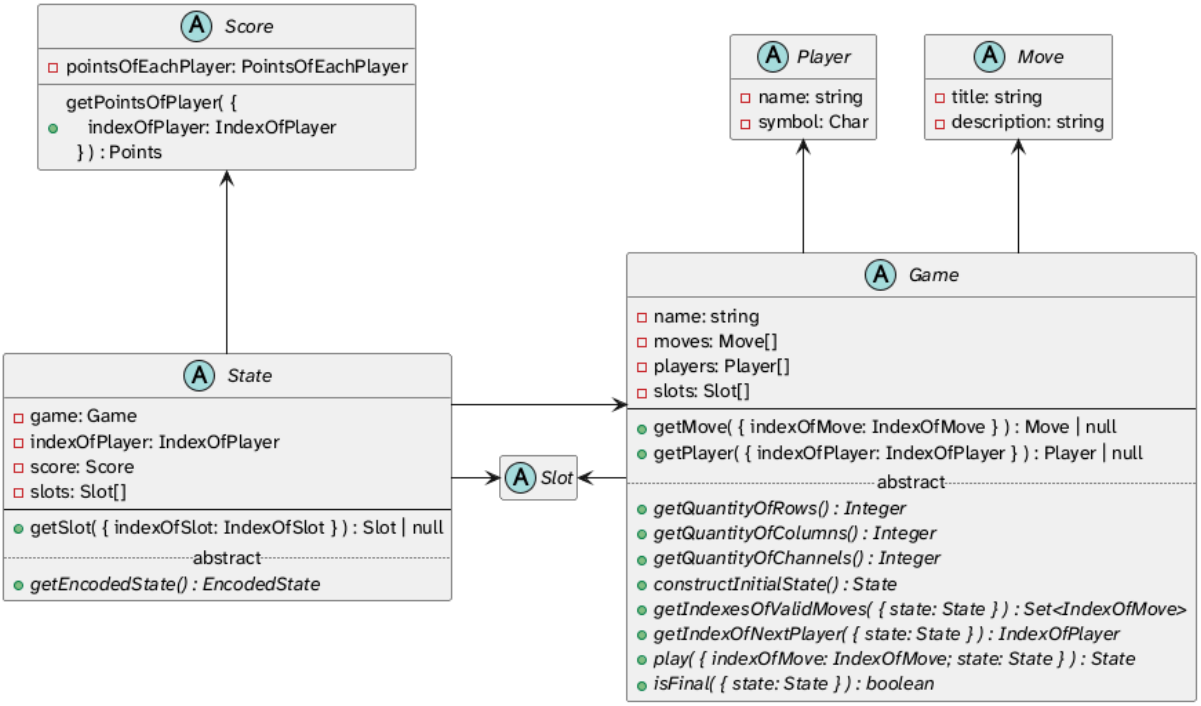
Fonte: elaborado pelo autor (2026).
Nota: O pacote `primitive` se refere aos tipos de dados concretos disponibilizados pela linguagem JavaScript.

Outro dado comumente referenciado é a marcação de pontos dos jogadores, que é feita com números inteiros pelo apelido `Points`. Guardamos a pontuação completa de todos os jogadores por meio da estrutura de indexação por chave-valor `Map`, do JavaScript. No tipo abstrato `PointsOfEachPlayer`, as chaves são definidas pelo índice de cada jogador, conforme registrado pelo projetista do jogo, ao passo em que os pontos são salvos no campo de valor. Finalmente, o tipo `EncodedState` representa o formato de codificação de um estado em canais, como descrito na Seção 2.5. Ele aceita qualquer matriz multidimensional de valores reais, embora tenhamos respeitado a convenção de utilizar apenas os valores 0 e 1 para definirmos tais codificações.

Após definir os tipos, passa-se à construção dos componentes fundamentais para descrever um jogo. Eles foram implementados por meio de classes abstratas, uma vez que a linguagem JavaScript não dispõe de estruturas como interfaces ou protocolos. Os principais atributos e métodos de cada classe, além das relações entre elas, podem ser vistos na Figura 19.

A classe mais simples é a que representa uma casa do tabuleiro, chamada de `Slot`. Esse conceito é um dos mais variáveis em jogos de turnos. Por exemplo, no Jogo da Velha, cada casa pode apenas ser marcada com o símbolo de um jogador. Já no Xadrez, há vários conjuntos de peças, que apresentam comportamentos diferentes. Ainda há jogos, como o Gobblet Gobblers[19], em que cada casa pode receber camadas empilhadas de peças. Ou ainda, essa classe poderia representar uma carta específica dentro de uma mão[20]. Essa variabilidade não nos permite

Figura 19 — Classes definidas pelo pacote `game`.



Fonte: elaborado pelo autor (2026).
Nota: As propriedades com visibilidade privada têm métodos públicos de encapsulamento para a obtenção de seus valores que não foram representados.

[19] Descrição disponível em: https://boardgamegeek.com/boardgame/13230/gobblet-gobblers.

[20] Apesar de termos determinado como limite do escopo desta pesquisa a investigação de jogos de tabuleiro, tentamos manter a implementação genérica para representar jogos de cartas futuramente.

atribuir nenhum dado comum. Dessa forma, cabe inteiramente ao projetista definir o conteúdo possível por meio de uma classe concreta que a implemente.

Em seguida, implementamos a classe abstrata `Player`, que representa os dados fixos de um jogador durante todo o período de duração da partida. Os dados comuns identificados foram acerca da distinção entre os jogadores na interface de execução por linha de comando. Nesse sentido, quando o projetista for criar um objeto da classe `Player`, ele deve atribuir um nome por meio do atributo `name` e um símbolo por meio do atributo `symbol` — como "primeiro" (1) e "segundo" (2), peças "brancas" (B) e "pretas" (P), ou (X) e (O), por exemplo.

Para registrar as possibilidades de transição entre estados, criamos a classe abstrata `Move` que representa um movimento. Por padrão, ela apenas guarda dados de identificação para a interface com o usuário, quais sejam o título com atributo `title` e sua descrição com atributo `description`. Para todas as classes abstratas, o projetista pode definir novos atributos caso sejam necessários para efetuar as regras do jogo.

A fim de permitir que os agentes inteligentes gerados possam avaliar as qualidade dos movimentos, é necessário que o projetista descreva previamente ao início da partida todos aqueles que são possíveis em qualquer momento. Por exemplo, Silver et al. (2017) representam o Xadrez por meio de 4672 movimentos, por meio de uma matriz de 8 casas na horizontal, 8 casas na vertical e 73 mudanças de estado que uma peça pode efetuar. Apesar de essa lista de opções ser extensa, ela é necessária porque a estrutura da rede neural usada pelo agente atribui um valor de qualidade para todos os movimentos do jogo, mesmo aqueles que não são válidos em um estado específico.

As classes descritas previamente têm a função de armazenar dados imutáveis no contexto de uma partida. Para representar um estado — o qual sintetiza a disposição variável dos elementos em um turno —, desenvolvemos a classe abstrata `State`. Ela deve manter, por meio do atributo `game`, uma referência para a classe que representa um jogo a fim de ter acesso às suas regras e a outros dados invariáveis.

Outra característica de um estado é manter a disposição de peças nas casas do tabuleiro, o que é feito por meio do vetor `slots`. Ele guarda objetos da classe `Slot` e deve ser indexado da mesma forma em todos os estados para que o programa consiga acessar os componentes de forma direta. O método concreto `getSlot` oferece uma facilidade ao desenvolvedor por implementar uma busca de uma casa naquele vetor dado o seu índice. Por isso, a decisão de como organizar os objetos naquele vetor deve ser pensada no mesmo momento em que o projetista implementa o método abstrato `getEncodedState`, o qual sintetiza todas as informações relevantes num conjunto de canais a ser fornecido para a rede neural. Outro atributo armazenado em cada objeto da classe `State` é o `indexOfPlayer`, que guarda a informação sobre qual dos jogadores pode realizar um movimento no turno atual, usualmente chamada de "vez do jogador".

A pontuação dos jogadores também depende de como os turnos decorreram durante a partida, o que é salvo no atributo `score`. Para fins de organização do código-fonte e de abertura para expansão, criamos uma classe abstrata chamada `Score` para representar a pontuação de todos os jogadores em um determinado estado. O único atributo dessa classe é o mapa `pointsOfEachPlayer`, que atribui um valor em pontos para cada jogador de acordo com o índice a esse atribuído pelo projetista. É relevante ressaltar que alguns jogos de tabuleiro, como o

Xadrez, não utilizam sistema de pontuação, atribuindo apenas o resultado de vitória para um dos jogadores. Nesses casos, recomendamos a implementação de forma que a quantidade de pontos permaneça como 0 durante toda a partida e que, no estado que representa fim de jogo, esse marcador seja alterado para 1 na entrada referente ao vencedor.

Finalmente, a classe abstrata `Game` representa as regras do jogo e guarda os conjuntos de dados imutáveis durante uma partida. Para representá-lo em interfaces com o usuário, o atributo `name` requer que o projetista o nomeie. Então, no atributo `slots`, o projetista deve fornecer a lista de casas organizada previamente. O mesmo deve ser feito em relação ao argumento `moves` para a lista de movimentos e em relação ao atributo `players` para a lista de jogadores. A classe oferece métodos auxiliares que buscam por um movimento ou por um jogador em seu respectivo vetor dado o seu índice.

Em relação aos métodos abstratos da classe `Game`, destacamos os `getQuantityOfRows`, `getQuantityOfColumns` e `getQuantityOfChannels` que respectivamente definem a a quantidade de linhas, de colunas e de canais da matriz que representa um estado codificado. Esses dados devem ser definidos previamente e ser imutáveis para um jogo, porque eles são usados na construção da arquitetura da ResNet que orienta o agente inteligente.

Outro método que deve ser determinístico é o `constructInitialState`, em que o projetista descreve a forma como o estado inicial da partida é construído. Por exemplo, no Jogo da Velha, ele se iniciaria com um tabuleiro vazio. Já no Xadrez, as casas de um lado do tabuleiro e do outro devem estar preenchidas pelas devidas peças de cada um dos jogadores.

O comportamento dos quatro últimos métodos citados seria melhor representado por métodos abstratos estáticos, uma vez que seus resultados não dependem de nenhum dos atributos guardados da classe. Entretanto, a linguagem JavaScript não permite a definição desse tipo de método, motivo pelo qual foram implementados como métodos abstratos e dinâmicos.

Agora tratando dos métodos da classe `State` que utilizam dados de seus objetos, destacamos a responsabilidade do método `getIndexesOfValidMoves`. Sua função é determinar, a partir de um certo estado fornecido, quais são os movimentos que o jogador daquele turno poderá executar. Para fins de otimização de memória, seu retorno deve ser um conjunto sem repetição de índices referentes aos movimentos válidos de acordo com a ordem dada pelo vetor salvo na classe `Game`. Esse comportamento é obtido pela estrutura de dados `Set`, implementada na linguagem JavaScript. Esse conjunto de jogadas válidas é utilizado, entre outros, para filtrar o vetor de qualidades atribuídas pelo modelo de ResNet e apresentar apenas os adequados ao agente inteligente.

Com uma lógica de implementação similar, o método `getIndexOfNextPlayer` deve determinar de qual jogador será a vez no próximo turno. É comum que os jogadores se alternem sequencialmente a cada turno durante uma rodada, mas é possível para o projetista definir as regras do jogo de forma que um jogador deixe de jogar por um turno ou que tenha nele mais de um movimento. O retorno desse método deve ser o índice do jogador escolhido conforme o vetor salvo na classe `Game`.

Com o auxílio do último método, o projetista pode descrever as regras para atualizar um dado estado. Uma vez que seguimos a convenção de que os componentes de descrição do jogo devem ser imutáveis, o método `play`, responsável por essa atualização, retorna um novo

objeto da classe `State`. Seus argumentos são o estado do turno atual e o índice do movimento a ser realizado. O projetista deve codificar a lógica para descrever a lista de casas atualizada, incrementar ou decrementar as pontuações e definir próximo jogador.

Após cada turno, é necessário determinar se o estado gerado leva ao fim da partida. O projetista deve descrever essa consulta por meio do método `isFinal`, que recebe o estado referenciado e retorna um valor do tipo `boolean`, definido como `true` para quando a partida deve se encerrar ou como `false` para quando ela deve continuar. Para isso, ele dispõe de todos os dados discutidos, como a disposição das peças, a pontuação dos jogadores e quaisquer outros atributos que ele tenha acrescentado às classes concretas criadas por ele.

4.3 IMPLEMENTAÇÃO DOS JOGOS

A fim de executar o experimento desta pesquisa, descrevemos e implementamos os componentes necessários de três jogos no módulo `games`, quais sejam: o Jogo da Velha uma variante dele nomeada de *Snowball* e o Ligue-4. Para cada um, definimos objetos concretos de forma a permitir ao usuário do sistema jogá-los. Uma parte representativa dos objetos foi selecionada para realizar testes de unidade, a fim de garantir que as regras dos jogos estavam bem definidas antes de prosseguir com a execução dos métodos de busca.

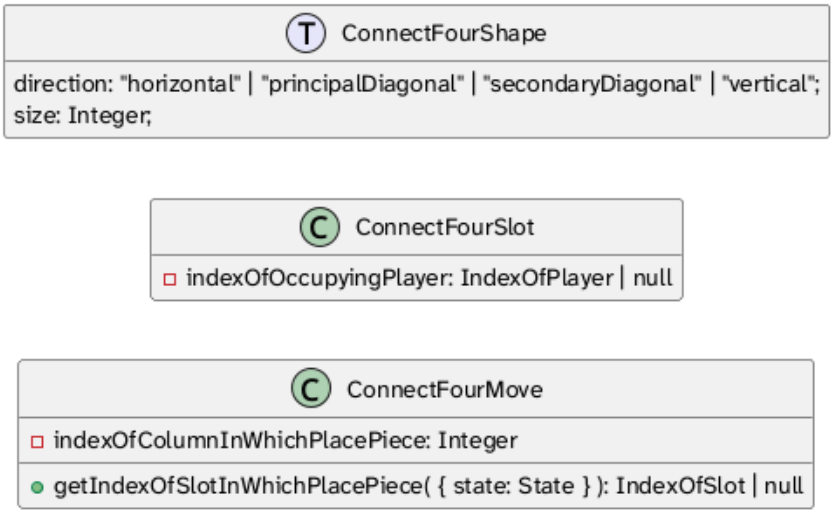
Conforme descrito na Subseção 2.2.1, percebe-se que não há marcação de pontuação durante a partida do Jogo da Velha, nem há em seu espaço de busca complexidade suficiente para avaliar o uso de agentes inteligentes. Por isso, elaboramos e implementamos sua variante *Snowball*, que permitiu comprovar a viabilidade do ambiente de representação de jogos para aqueles dependentes de manutenção do dado de pontuação em cada estado.

Nesta seção, descrevemos o processo de implementação do jogo Ligue-4, discutido na Subseção 2.2.2, destacando seus componentes de descrição. Ele foi escolhido para realizar o experimento porque é um jogo de informação completa entre dois jogadores que apresenta um tamanho de tabuleiro razoável e uma quantidade pequena de movimentos possíveis. Em relação à implementação das classes abstratas, poucas adaptações foram necessárias. Todas as classes concretas seguiram a convenção de iniciar seus nomes com o termo `ConnectFour` seguido do nome da classe que ela implementa.

Conforme visto na Figura 20, as classes `Slot` e `Move` foram acrescidas de novos atributos. Além disso, observamos a necessidade de criar uma nova estrutura de dados abstrata para representar os formatos considerados para levar à vitória, o que foi feito pelo tipo `ConnectFourShape`. Ele permite definir linhas de um tamanho arbitrário — embora tenhamos escolhido 4 peças conforme a descrição padrão do jogo — e a direção de marcação — se horizontal, vertical ou em uma diagonal principal ou secundária.

A primeira classe concreta implementada foi a `ConnectFourPlayer`, referente aos dados imutáveis de cada jogador. O Ligue-4 não guarda nenhuma informação relevante sobre um jogador exceto aquelas necessárias para a sua distinção na interface com o usuário. Assim, não foi necessária nenhuma alteração na classe. Ao criar seus objetos, escolhemos arbitrariamente o nome “Alice” e o símbolo “X” para o jogador de índice 0 e o nome “Bruno” e símbolo “O” para

Figura 20 — Classes concretas alteradas na implementação do Ligue-4 e tipo utilitário nela definido.



Fonte: elaborado pelo autor (2026).
Nota: As propriedades com visibilidade privada têm métodos públicos de encapsulamento para a obtenção de seus valores que não foram representados.

o jogador de índice 1. Tais valores não representam nomes reais de pessoas, mas servem apenas como facilitadores de distinção entre esses objetos para os desenvolvedores do protótipo.

Em seguida, implementamos a classe concreta `ConnectFourSlot`, que representa o conteúdo guardado em uma casa do tabuleiro. O Ligue-4 utiliza peças simples, cuja única diferença é a cor, que é associada a cada um dos jogadores. Por isso, a única informação relevante para cada casa é se ela está vazia ou, caso não esteja, qual jogador a preencheu. Então, acrescentamos o atributo `indexOfOccupyingPlayer`, que pode ser assinalado com o índice 0 caso o jogador "X" tenha marcado uma peça, com o índice 1 caso o jogador "O" o tenha feito, ou com o valor `null` se a casa estiver vazia. Quanto aos objetos utilizados pelo experimento, criamos todas as 49 casas, definindo o atributo de jogador ocupante como `null` e nomeando-as com a convenção "rXcY", em que os termos "X" representam o índice da linha que ela ocupa e o termo "Y" representa o da coluna. Para os testes de unidade, também criamos novos objetos preenchidos em diferentes combinações.

Diferentemente do Jogo da Velha, em que cada movimento tem relação direta com uma única casa do tabuleiro, o Ligue-4 precisa calcular a posição onde marcar uma peça a depender de dois fatores: o índice da coluna escolhida pelo jogador e a disposição de peças já marcadas nela. Percebe-se então que esse índice deve ser armazenado no atributo `indexOfColumnInWhichPlacePiece` da classe concreta `ConnectFourMove`. Implementamos também o método auxiliar `getIndexOfSlotInWhichPlacePiece`, responsável por acessar, de baixo para cima, cada casa da coluna para encontrar a primeira que esteja vazia no estado fornecido. Depois, criamos um objeto para cada uma das colunas, cujo índice guardamos no atributo discutido e cujos títulos e descrições foram dados em relação ao seu número ordinal.

A verificação acerca da marcação dos formatos de linha foi implementada por funções no arquivo nomeado `ConnectFourShape`. A lógica desses utilitários é sintetizada no Algoritmo 1,

que determina se um formato iniciado em dada casa está sendo ocupado por algum jogador e, caso esteja, qual é o seu índice.

Algoritmo 1 — Código-fonte simplificado da função `getIndexOfPlayerWhoIsOccupyingShape`.

```
function getIndexOfPlayerWhoIsOccupyingShape(
    indexOfFirstSlot: IndexOfSlot, shape: Shape
): IndexOfPlayer | null {
    const slots = getSlotsThatFormShape(indexOfFirstSlot, shape);
    let indexOfPlayerOccupyingPreviousSlot = null

    for (const slot of slots) {
        const indexOfPlayer = slot.getIndexOfOccupyingPlayer();
        if (indexOfPlayerOccupyingPreviousSlot == null) {
            indexOfPlayerOccupyingPreviousSlot = indexOfPlayer;
        } else if (indexOfPlayer !== indexOfPlayerOccupyingPreviousSlot) {
            return null;
        }
        indexOfPlayerOccupyingPreviousSlot = indexOfPlayer;
    }

    return indexOfPlayerOccupyingPreviousSlot;
}
```

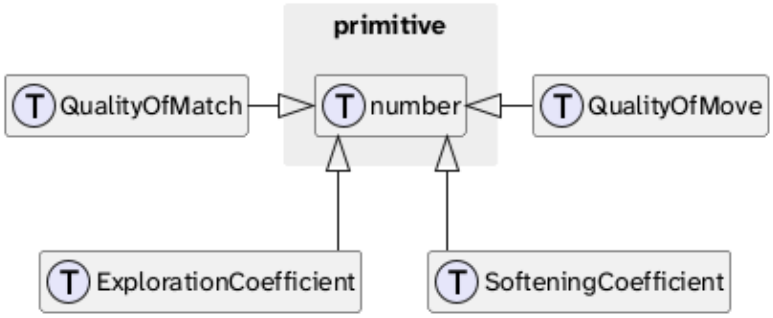
Fonte: elaborado pelo autor (2026).

Essa função é utilizada na classe concreta `ConnectFourScore`, que representa e oferece métodos para calcular a pontuação dos jogadores. Quando seus objetos são inicializados, todos os jogadores têm atribuído o valor de 0 pontos. Por meio de seu método auxiliar `getUpdatedScore`, o programa verifica, para cada uma das casas, se houve marcação de qualquer uma das linhas de 4 peças adjacentes. Caso positivo, a função retorna um objeto `Score` em que o jogador vencedor é marcado com 1 ponto.

Então, a classe concreta `ConnectFourGame` utiliza todos os dados discutidos para representar as regras do jogo. Ao criar seu objeto, o projetista deve fornecer as listas de jogadores, movimentos e casas previamente instanciadas. Um primeiro método de destaque dessa classe é o `getIndexesOfValidMoves`, cuja implementação recebe um estado e retorna os índices das colunas do tabuleiro em que alguma de suas casas ainda esteja vazia. Após selecionar um movimento, o jogador deve executar o método `play`, que retorna o estado atualizado com a marcação da peça na posição escolhida, além da eventual pontuação nova caso tenha sido uma jogada vitoriosa. Em seguida, o algoritmo utiliza o método `isFinal` para determinar se a partida chegou ao fim com o novo estado, o que ocorre quando todas as casas estão preenchidas ou quando um dos jogadores marcou um ponto. Caso a partida continue, o método `getIndexOfNextPlayer` é responsável por passar a vez para o oponente.

Outra responsabilidade da implementação da classe `Game` é estabelecer a quantidade de linhas, de colunas e de canais do estado codificado. Decidimos utilizar a mesma dimensão do tabuleiro (6 linhas e 7 colunas) para a codificação e empilhar sobre ela 4 canais de dados, inicializados com o valor 0. Como descrito na Seção 2.5, o canal de índice 0 terá cada um de seus valores definido como 1 se a casa correspondente por estiver marcada pelo jogador “X”. Já as casas do canal de índice 1 serão ativadas pelas peças do jogador “O”, ao passo em que as casas vazias ativam o canal de índice 2. Finalmente, o canal de índice 3 tem a responsabilidade de

Figura 21 — Tipos de dados comuns definidos pelo pacote `search`.



Fonte: elaborado pelo autor (2026).
Nota: O pacote `primitive` se refere aos tipos de dados concretos disponibilizados pela linguagem JavaScript.

informar à rede neural de qual jogador é a vez no turno atual, sendo completamente preenchido com 0 caso seja do jogador “X” ou com 1 caso seja do jogador “O”.

4.4 ELABORAÇÃO DOS ALGORITMOS DE BUSCA

Havendo devidamente representado o jogo Ligue-4, passamos à implementação do módulo `search`, responsável pelos algoritmos de busca em árvore de Monte Carlo e de predição por meio de ResNets. A lógica de construção de suas principais classes foi inspirada pela implementação de referência de Förster (2023).

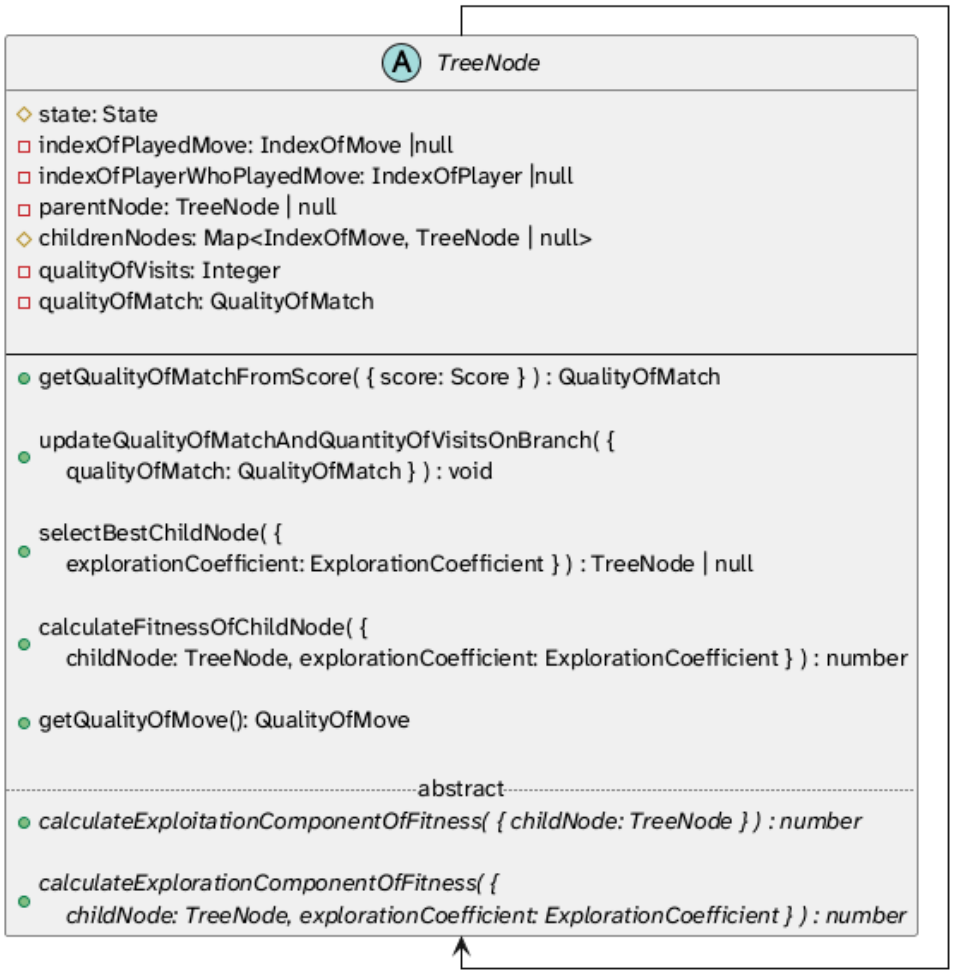
Primeiramente define-se tipos úteis para a melhor descrição de conceitos comuns, como mostrado na Figura 21. Todos eles são apelidos do tipo primitivo `number`, que representa números reais. Seus significados são descritos nesta seção, conforme a discussão acerca de seus usos.

O primeiro elemento necessário para implementar a MCTS é a classe abstrata `TreeNode`, cujo diagrama é apresentado na Figura 22. Ela tem a função de implementar um nó da árvore de busca, o qual representa um estado da partida simulada e que é guardado em seu atributo `state`. Também são importantes os dados sobre o histórico que levou até esse estado. Por isso, armazenamos no atributo `indexOfPlayedMove` o índice do movimento jogado no turno anterior e no atributo `indexOfPlayerWhoPlayedMove` o índice do jogador que o efetuou. O caso em que esses dois marcadores estarão vazios é no início da partida, que corresponde ao nó raiz da árvore.

Para representar a transição entre os nós e permitir realizar a fase de retro-propagação da busca, salvamos em cada nó a referência para seu nó pai por meio do atributo `parentNode`, que estará vazio apenas para a raiz da árvore. Uma vez que um estado pode levar a múltiplas configurações da partida por meio de cada um de seus movimentos válidos, decidimos representar, no atributo `childrenNodes`, as transições do nó para seus filhos por meio de um mapa indexado, em que cada entrada marca o movimento escolhido e o nó que ele gerou.

Como discutido na Seção 2.3 sobre a diretriz de *fitness* da MCTS, a cada ciclo de busca, a etapa de retro-propagação incrementa o contador de visitas e atualiza a expectativa de qualidade da partida para todos os nós do ramo selecionado. Esses dois marcadores são armazenados nos atributos `quantityOfVisits` e `qualityOfMatch`, respectivamente.

Figura 22 — Classe `TreeNode` definida no pacote `search`.



Fonte: elaborado pelo autor (2026).
Nota: As propriedades com visibilidade privada e protegida têm métodos públicos de encapsulamento para a obtenção de seus valores que não foram representados.

Quanto aos métodos da classe `TreeNode`, destacamos o `getQualityOfMatchFromScore`, que converte a pontuação final dos jogadores em um número do tipo `QualityOfMatch`, representante da qualidade da partida para o jogador atual. Uma vez que esse comportamento é necessário em outras partes do projeto, a maior parte do seu processamento é, na verdade, realizado por um método auxiliar chamado `calculateQualityOfMatch`, que recebe as pontuações e o índice do jogadores atual. Esse dado de qualidade é retro-propagado recursivamente até o nó raiz por meio do método `updateQualityOfMatchAndQuantityOfVisitsOnBranch`, incrementando-o nos turnos do jogador vencedor e decrementando-o para os demais.

Já a etapa de seleção é gerenciada pelo método `selectBestChildNode`, que calcula o valor de *fitness* para cada nó já expandido e escolhe o melhor. Para isso, é chamado o método `calculateFitnessOfChild`, que soma os componentes de aproveitamento e de exploração da equação de UCT, equilibrando-os por meio do coeficiente de exploração fornecida. Uma vez que a MCTS clássica e a adaptada pelo *AlphaZero* calculam o valor de *fitness* de forma diferente, utilizamos os métodos abstratos para defini-los.

Finalmente, o método `qualityOfMove` é responsável por classificar os movimentos válidos a partir do estado inicial da árvore. A forma de avaliação utilizada pela implementação

de referência (Förster, 2023) prioriza os movimentos que levaram a ramos com o maior número de visitas. Essa lógica se justifica porque se entende que um estado muito visitado foi aquele mais selecionado pela diretriz de busca. Entretanto, percebemos que, quando realizamos a busca a partir de um estado próximo de levar a uma vitória, essa heurística se prova falha. Isso ocorre porque o estado vitorioso não gera mais filhos e, dessa forma, não pode mais ser visitado pela busca. Assim, o algoritmo é obrigado a visitar seus vizinhos, o que os torna melhor classificados. Para resolver esse problema, decidimos alterar o cálculo da qualidade de um movimento para a Equação 3, que alinha a qualidade estimada da partida e a quantidade de visitas ao dado ramo.

Equação 3 — Cálculo da qualidade de um movimento a partir da árvore de busca construída pelo método de busca em árvore de Monte Carlo (MCTS).

$$A(n) = Q(n) + \sqrt[4]{V(n)} \tag{3}$$

Na qual:

- $A(n)$ é a qualidade do movimento representado pelo nó n;
- $Q(n)$ é a qualidade da partida calculada por meio de simulações a partir do nó n;
- $V(n)$ é quantidade de vezes em que o nó n foi visitado nas iterações anteriores.

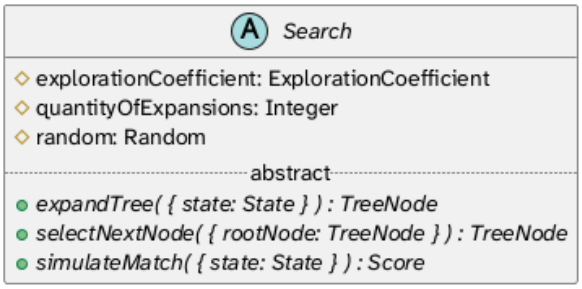
Fonte: elaborado pelo autor (2026).

A busca em árvore de Monte Carlo é gerenciada pela classe abstrata `Search`, cujo diagrama é mostrado na Figura 23. Ela armazena dados relevantes para executar o algoritmo, como o coeficiente de exploração e a quantidade de ciclos a serem realizados, o que é guardado no atributo `quantityOfExpansions`, além de um objeto da classe auxiliar `Random` que realiza operações pseudo-aleatórias a partir da mesma *seed* informada ao programa.

O método abstrato `expandTree` da classe `Search` executa o ciclo de busca, utilizando o método `selectNextNode` para realizar a etapa de seleção e o método `simulateMatch` para implementar a etapa de simulação da MCTS clássica ou de predição da MCTS adaptada pelo *AlphaZero*. Esse primeiro algoritmo foi implementado nas classes concretas `CommonSearch` e `CommonTreeNode`. Essa define a etapa de expansão por um método chamado `expand`, que recebe o movimento a expandir e gera um único novo nó.

Já em relação à MCTS adaptada, a classe concreta `AgentGuidedSearch` implementa a busca e define um novo atributo chamado `predictionModel`, que guarda o modelo de ResNet responsável por orientar a etapa de predição. Em seguida, durante a etapa de expansão, os valores estimados por sua *policy head* geram todos os movimentos válidos

Figura 23 — Classe `Search` definida no pacote `search`.



Fonte: elaborado pelo autor (2026).

para o estado atual. Essa fase é implementada pelo método `expand` da classe concreta `AgentGuidedTreeNode`, que recebe aquele vetor e guarda a qualidade estimada no novo atributo `qualityOfMoveAttributedByModel` de cada nó filho. Por fim, a predição da qualidade da partida é utilizada para orientar a fase de retro-propagação.

4.5 CONSTRUÇÃO DA REDE NEURAL RESIDUAL

Considerando a variação de complexidade entre diferentes jogos e seguindo a recomendação da implementação de referência (Förster, 2023), possibilitamos ao projetista de um protótipo definir parâmetros da arquitetura da rede neural residual utilizada pelos agentes inteligentes. Para isso, criamos a classe `ResidualNeuralNetwork`, que recebe os seguintes dados: (1) `seed`, usado para inicializar os pesos e vieses da rede neural; (2) `quantityOfResidualBlocks`, para definir a quantidade de blocos residuais a serem criados na *backbone* da rede; e (3) `quantityOfHiddenChannels`, referente à quantidade de canais usada nas camadas internas de processamento da rede.

A classe construtora de modelos de rede neural e as operações sobre tensores foram disponibilizadas pelo pacote do projeto TensorFlow.js. Ele disponibiliza algumas formas de construir a arquitetura da rede, dentre as quais selecionamos a de *LayersModel*. Tomamos o cuidado de encapsular o uso do TensorFlow dentro dessa classe, a fim de permitir sua substituição se necessário sem requerer a refatoração de outros componentes do projeto. Então, definimos funções auxiliares para a construção das camadas de adaptação da entrada, de blocos residuais e de saída para a *policy head* e para a *value head*.

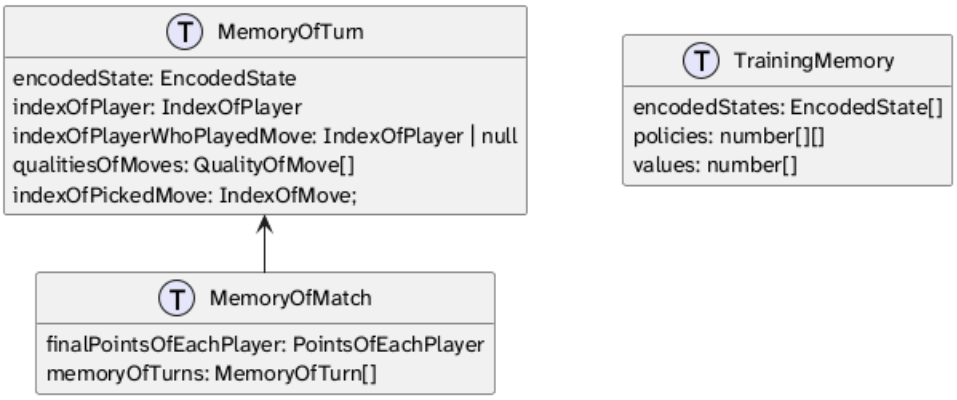
Acerca do treinamento, o método `train` dessa mesma classe recebe os conjuntos de estados codificados e de saídas esperadas para a *policy head* e para a *value head*. O alinhamento dos pesos e vieses é realizado pelo método `fit` do objeto de `LayersModel`, utilizando o otimizador estimativa de momento adaptativo (*Adam*). Para a *policy head*, selecionamos a função de perda de entropia cruzada categórica (em inglês, *categorical cross-entropy*), ao passo em que escolhemos a função de erro quadrático médio (em inglês, *mean squared error*) para calcular a perda da *value head*. Quanto à execução do programa, permitimos que o usuário escolha os seguintes parâmetros: (1) `quantityOfEpochs`, para definir a quantidade de épocas de treinamento; e (2) `sizeOfBatch`, para ajustar o tamanho do conjunto de entradas e saídas usado a cada passo de alinhamento.

4.6 GERAÇÃO DE MEMÓRIAS DE TREINAMENTO

Com o fim de encapsular o uso da ResNet e de relacioná-la com um jogo implementado, criamos uma nova classe no módulo `search` chamada `PredictionModel`. Seu método mais relevante é denominado `predict`, que recebe um estado e retorna dois elementos: (1) o vetor das qualidades atribuídas a cada movimento listado para aquele jogo; e (2) a qualidade estimada para a partida a partir do turno atual.

Definimos também a função auxiliar `calculateProbabilityOfPlayingEachMove`, que recebe o vetor de qualidades mencionado, o conjunto dos índices dos movimentos válidos naquele estado e um valor do tipo `SofteningCoefficient`, o qual é definido pelo usuário do

Figura 24 — Tipos de dados relacionados à criação de uma memória de partidas definidos pelo pacote `search`.



Fonte: elaborado pelo autor (2026).

programa. Essa função aplica uma transformação de *softmax*, utilizando o coeficiente citado para ajustar a proporção em que os movimentos mais bem avaliados devem se destacar entre as probabilidades calculadas. Essas são retornadas na estrutura de um mapa que contém apenas entradas para os movimentos válidos.

Criamos então, na classe auxiliar `Random`, o método `pickMoveConsideringItsQuality`, que usa essas probabilidades para ordenar a lista de movimentos válidos e sorteia um número aleatório para selecionar um deles. Dessa forma, aqueles com maiores probabilidades associadas têm mais chance de serem selecionados na roleta.

Para implementar o ciclo de treinamento do modelo, que envolve gerar uma memória de partidas simuladas e alinhar os pesos e vieses da rede neural aos resultados dos turnos, descrevemos os tipos de dados mostrados na Figura 24.

O uso dos tipos `MemoryOfTurn` e `MemoryOfMatch` estão associados ao algoritmo de *self-play*, implementado pela função `buildMemoryOfMatch`, cujo código-fonte simplificado é mostrado no Algoritmo 2. Ele recebe um objeto do tipo `AgentGuidedSearch`, que realiza a busca em árvore de Monte Carlo adaptada pelo projeto *AlphaZero*.

A inicialização do processo de geração de memória define a variável que armazenará o histórico de turnos, implementada como um vetor de objetos do tipo `MemoryOfMatch`. Além disso, são criados os marcadores auxiliares do estado atual e do jogador que realizou o último movimento na partida.

Então, inicia-se um laço de repetição, em que o algoritmo utiliza a MCTS para obter as qualidades atribuídas a cada um dos movimentos. Uma vez que a ResNet precisa receber o vetor completo de todos os movimentos possíveis no jogo, as posições referentes aos movimentos inválidos são preenchidas com o número especial que representa infinito negativo no JavaScript.

O algoritmo dá prosseguimento ao turno, ao utilizar o método pseudo-aleatório da roleta para selecionar um movimento. Em seguida, os dois marcadores, o vetor de qualidades, o estado codificado e o índice do movimento escolhido são armazenados no histórico.

Esse movimento selecionado é executado sobre o estado atual, gerando um novo estado, o qual é aferido para determinar se ele levou ao fim da partida. Caso positivo, a função `buildMemoryOfMatch` retorna um objeto do tipo `QualityOfMatch`, que é composto pelo histórico

Algoritmo 2 — Código-fonte simplificado da função `buildMemoryOfMatch`.

```
function buildMemoryOfMatch(
    search: AgentGuidedSearch
): MemoryOfTurn[] {
    const game = search.getGame();
    const memoryOfTurns: MemoryOfTurn[] = [];

    let currentState = game.constructInitialState();
    let indexOfPlayerWhoPlayedMove: IndexOfPlayer | null = null;

    while (true) {
        const qualitiesOfMoves = searchQualityOfMoves(search, currentState);

        const indexesOfValidMoves = game.getIndexesOfValidMoves(currentState);
        const indexOfPickedMove = random.pickMoveConsideringItsQuality( ↵
            indexesOfValidMoves, qualitiesOfMoves);

        memoryOfTurns.push({
            encodedState: currentState.getEncodedState(),
            indexOfPlayer: currentState.getIndexOfPlayer(),
            indexOfPlayerWhoPlayedMove,
            qualitiesOfMoves,
            indexOfPickedMove
        });

        const nextState = game.play(indexOfPickedMove, currentState);
        if (nextState.isFinal()) {
            const finalPointsOfEachPlayer = nextState.getScore() ↵
              .getPointsOfEachPlayer();
            return {
                finalPointsOfEachPlayer,
                memoryOfTurns,
            };
        }

        indexOfPlayerWhoPlayedMove = currentState.getIndexOfPlayer();
        currentState = nextState;
    }
}
```

Fonte: elaborado pelo autor (2026).

de turnos e pela pontuação de todos os jogadores no fim da partida. Caso contrário, os marcadores auxiliares são atualizados e mais um passo de simulação é realizado.

Considerando que o treinamento de um agente inteligente requer um histórico grande de partidas, criamos uma nova função chamada `buildMemoryOfMatches`. Ela recebe do usuário o parâmetro `quantityOfIterations`, acerca da quantidade de partidas a serem simuladas. Então, começa um laço de repetição que salva num vetor do tipo `MemoryOfMatch` todos os resultados das execuções do método `buildMemoryOfMatch` já discutido.

Por fim, o método `convertMemoryOfMatchesToTrainingMemory` transforma o resultado da fase de geração de memórias em três vetores de tipo único. O primeiro deles, `encodedStates`, guarda os estados codificados salvos em cada turno simulado. Por sua vez, o segundo, `policies`, armazena os vetores de qualidade de movimentos também salvos durante a simulação. Finalmente, o terceiro, `values`, é obtido pelo uso do método auxiliar `calculateQualityOfMatch`,

Figura 25 — Interface do programa Sistema de Teste de Jogabilidade Automatizado (APTS).

```
Options:
  -V, --version                        output the version number
  -h, --help                           display help for command

Commands:
  search-quality [options]             Search quality of a Monte-Carlo Tree node.
  play-match-using-search [options]    Play match using Monte-Carlo Tree Search.
  play-match-using-agent [options]     Play match using agent trained using ResNet.
  predict-quality [options]            Predict quality of a Monte-Carlo Tree node using a prediction model.
  play-match-pvp [options]             Play match against another player.
  construct-model [options]            Construct a Residual Neural Network model.
  build-training-memory [options]      Build memory of inputs and outputs for training model via self-play.
  train [options]                      Train model using already exported training memory.
  help [command]                       display help for command
```

Fonte: elaborado pelo autor (2026).

que usa a pontuação e o marcador de jogador atual em cada turno para calcular a qualidade da partida. Esses três vetores são retornados num objeto do tipo `TrainingMemory`.

4.7 INTERFACE COM O USUÁRIO

As funcionalidades criadas e discutidas requeriam uma interface padronizada para que aplicações as acessassem sem interagir com os detalhes de implementação. Para isso, organizamos no pacote `interface` um conjunto de ações disponíveis ao usuário. Elas foram implementadas como comandos de terminal em um pacote do projeto chamado `node`, que utilizou para isso a biblioteca Commander.js. A Figura 25 exibe a interface da tela de ajuda do programa APTS, mostrando os comandos disponíveis, que são discutidos nesta seção.

Inicialmente, oferecemos no comando `search-quality` uma forma de visualização da árvore de busca gerada pelo método de MCTS. Para isso, o usuário fornece os seguintes dados: (1) a estratégia de busca — se a clássica ou a adaptada pelo *AlphaZero* —; (1a) o modelo de predição, caso o usuário escolha a versão adaptada; (2) o coeficiente de exploração para cálculo da diretriz UCT; (3) a quantidade de ciclos iterados pela MCTS; (4) o coeficiente de suavização para calcular as probabilidades atribuídas a cada movimento; (5) uma *seed* para calcular os valores pseudo-aleatórios; e (6) o estado sobre o qual se quer descobrir os melhores movimentos viáveis. O programa executará a busca, calculará as qualidades e probabilidades dos movimentos e os imprimirá, conforme exemplo dado da Figura 26. Além disso, será gerado um arquivo do formato SVG que exibe árvore de busca montada, o qual é gerado pelo programa Graphviz e cujos recortes são mostrados na Figura 27.

Figura 26 — Qualidades de movimentos e probabilidades de vitória a efetuá-los estimadas pela MCTS clássica.

(a) Qualidades dos movimentos.

0	-7.50
1	5.61
2	-11.60
3	136.40
4	2.75
5	5.57
6	41.01

(b) Probabilidades de vitória.

0	1.07e-250
1	6.31e-228
2	7.84e-258
3	1
4	6.67e-233
5	5.26e-228
6	1.93e-166

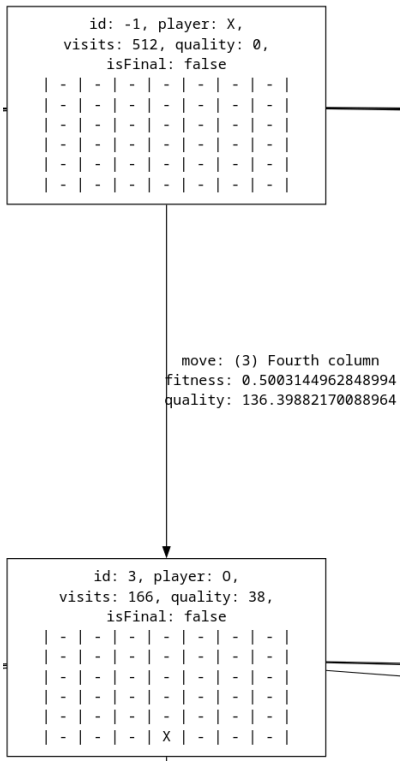
Fonte: elaborado pelo autor (2026).

Caso o usuário queira obter apenas a avaliação de um modelo de predição sobre um determinado estado, ele pode informá-los ao comando `predict-quality`, que também requer o coeficiente de suavização. Ela solicitará a predição ao modelo e imprimirá as qualidades dos movimentos retornadas e probabilidades calculadas.

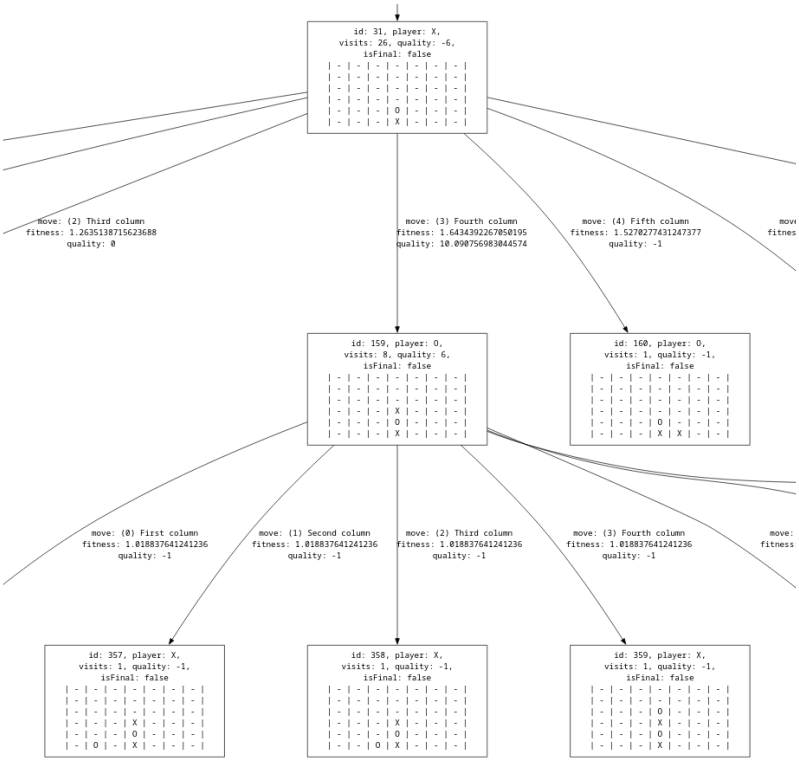
O programa também oferece ambientes de execução de partidas entre dois jogadores humanos, que interajam pelo mesmo terminal por meio do comando `play-match-pvp`, como mostrado na Figura 28a. Ele requer que se informe o estado do jogo sobre o qual se deseja iniciar a partida. Então, inicia um laço de repetição até que a partida chegue a um estado de fim de

Figura 27 — Árvore de busca montada ao avaliar a qualidade de um estado por meio da MCTS clássica.

(a) Recorte a partir da raiz.



(b) Recorte a partir de um estado avançado.



Fonte: elaborado pelo autor (2026).

jogo. A cada iteração, o algoritmo obtém por meio das regras quais são os movimentos válidos a partir do estado atual. Em seguida, mostra essa lista ao usuário por meio da biblioteca Inquirer.js e requer que ele escolha um movimento. O programa o efetua, marca o estado gerado como o atual e verifica se ele representa o fim da partida.

Laços similares são implementados para as ações em que o usuário decide jogar contra o computador ou quando ele inicia um jogo entre dois agentes inteligentes, como exibido na Figura 28b. Nesses casos, em vez de solicitar a seleção de movimentos para o jogador, o algoritmo utiliza a MCTS para obter as probabilidades dos movimentos por meio do comando `play-match-using-search`, ou apenas solicita essas predições para modelos de ResNet fornecidos, por meio do comando `play-match-using-agent`. Então, o movimento efetuado é escolhido pseudo-aleatoriamente pelo método da roleta.

Acerca da geração de agentes inteligentes, o programa oferece três comandos relevantes. O primeiro é o `constuct-model`, que gera um modelo de ResNet segundo os parâmetros informados e o exporta em dois arquivos de descrição do TensorFlow. O primeiro é um arquivo de formato JSON que descreve toda a estrutura da rede neural — a qual pode ser observada na Figura 29 —, e o segundo é um arquivo binário que salva os pesos e vieses aleatoriamente gerados. Ao usar esse comando, o usuário deve fornecer os dados acerca: (1) do jogo a ser simulado; (2) da quantidade de blocos residuais; (3) da largura em canais da *backbone* da rede; e (4) da *seed* usada para inicializar as conexões.

Esse primeiro modelo gerado não estará apto a orientar um agente inteligente. Antes disso, é necessário sujeitá-lo ao processo de treinamento. O primeiro passo para isso é gerar a

Figura 28 — Ambiente de jogatina entre jogadores e entre agentes inteligentes.

(a) Modo jogador vs. jogador.

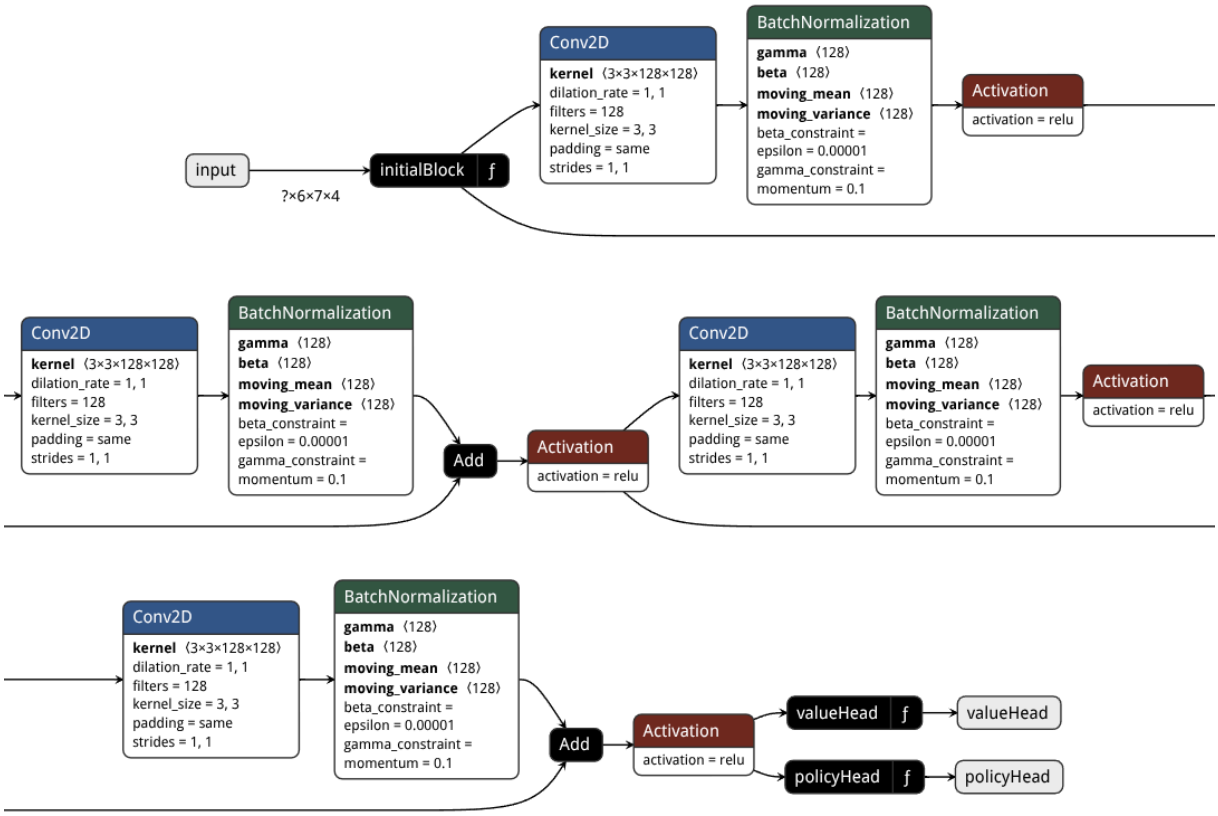
```
Turn of: (O) Bruno
| - | - | - | - | - | - | - |
| - | - | O | - | - | - | - |
| - | - | X | O | O | O | - |
| - | - | O | O | O | X | X |
| - | - | X | O | X | X | X |
| - | O | X | X | X | O | X |

✔ Select a move (2) Second column

Turn of: (X) Alice
| - | - | - | - | - | - | - |
| - | - | O | - | - | - | - |
| - | - | X | O | O | O | - |
| - | - | O | O | O | X | X |
| - | O | X | O | X | X | X |
| - | O | X | X | X | O | X |

? Select a move
❯ (1) First column
  (2) Second column
  (3) Third column
  (4) Fourth column
  (5) Fifth column
  (6) Sixth column
  (7) Seventh column

Place piece on the First column.
↑↓ navigate • ⏎ select
```

(b) Modo agente inteligente vs. agente inteligente.

```
Turn of: (O) Bruno
| - | - | - | - | - | - | - |
| - | - | O | - | - | - | - |
| - | - | X | O | O | O | - |
| - | - | O | O | O | X | X |
| - | - | X | O | X | X | X |
| - | O | X | X | X | O | X |

Played move: Fifth column. Place piece on the Fifth column.

Turn of: (X) Alice
| - | - | - | - | - | - | - |
| - | - | O | - | O | - | - |
| - | - | X | O | O | O | - |
| - | - | O | O | O | X | X |
| - | - | X | O | X | X | X |
| - | O | X | X | X | O | X |

Played move: Fifth column. Place piece on the Fifth column.

Turn of: (O) Bruno
| - | - | - | - | X | - | - |
| - | - | O | - | O | - | - |
| - | - | X | O | O | O | - |
| - | - | O | O | O | X | X |
| - | - | X | O | X | X | X |
| - | O | X | X | X | O | X |

Played move: Second column. Place piece on the Second column.

| - | - | - | - | X | - | - |
```

Fonte: elaborado pelo autor (2026).

Figura 29 — Estrutura de uma ResNet criada para o jogo Ligue-4 com dois blocos residuais.

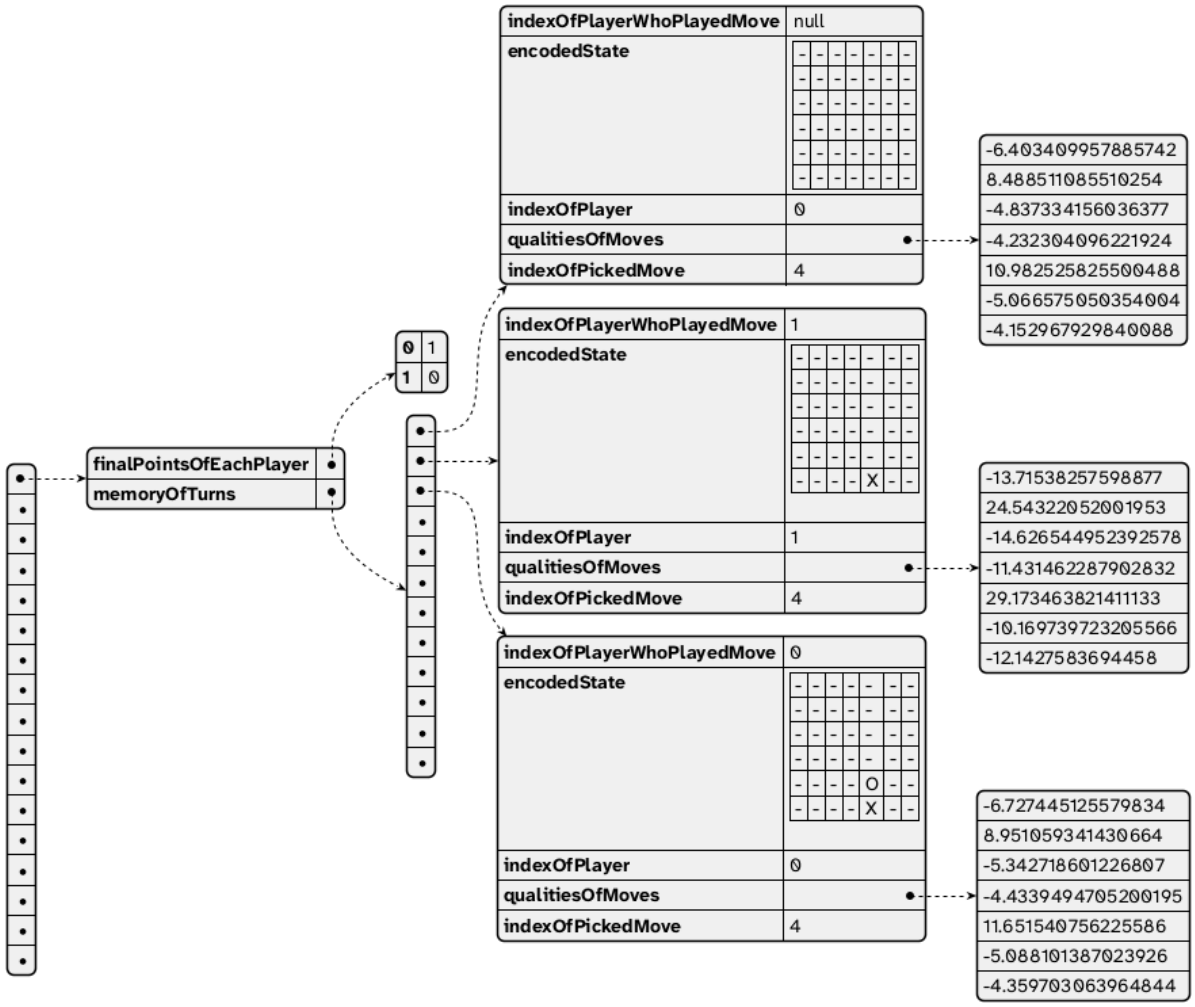


Fonte: elaborado pelo autor (2026).

memória de partidas sintéticas. Com esse objetivo, o comando `build-training-memory` gera um vetor do tipo `MemoryOfMatch` por meio da função `buildMemoryOfMatches` discutida na Seção 4.6 e o salva em um arquivo de tipo JSON, como exibido na Figura 30. Em seguida, o programa converte a memória num objeto do tipo `TrainingMemory` e também o salva em outro arquivo de tipo JSON.

Finalmente, o comando `train` pode ser chamado para alinhar um modelo ao histórico gerado. Para isso, o algoritmo utiliza o método `train` discutido na Seção 4.5. Um parâmetro novo que esse comando requer é chamado `valueToReplaceInfinity`, que tem o objetivo de substituir o marcador de movimento impossível nos vetores de qualidade salvos na memória de partidas. Isso é necessário para que o TensorFlow consiga realizar operações sobre os valores de entrada dentro de seu limite de representação de bits. Dessa forma, o valor fornecido para o comando de treinamento atua como uma penalidade para os movimentos inválidos.

Figura 30 — Dados representativos de memórias de partidas sintéticas geradas pelo método de *self-play*.



Fonte: elaborado pelo autor (2026).
Nota: Os estados codificados foram representados como tabuleiros para facilitar a visualização. Nos arquivos, eles são salvos no formato de canais binários.

5 RESULTADOS

A execução do experimento de geração de agentes inteligentes requer que se realizem as atividades descritas na Seção 3.2. Seu primeiro passo é codificar classes de representação do jogo simulado, qual seja o Ligue-4, e instanciar os objetos relativos aos seus componentes fundamentais, como descrito na Seção 4.3. Então, deve-se criar um modelo de rede neural específico para essa implementação, realizar seu treinamento e executar a coleta de dados de partidas sintéticas. Este capítulo descreve como tais atividades foram efetuadas e discute os resultados delas obtidos.

5.1 GERAÇÃO DE AGENTES INTELIGENTES

A fim de gerar os agentes inteligentes usados no experimento, deve-se criar uma estrutura de ResNet por meio da execução do comando `constuct-model`, discutido na Seção 4.7. Inspirados pela sugestão dada pela implementação de referência (Förster, 2023), decidimos construir um modelo para o jogo Ligue-4 de 8 blocos residuais e com largura de 128 canais internos. O algoritmo desse comando constrói a rede neural e a exporta como uma pasta que guarda dois arquivos: o de estrutura das camadas; e o de definição dos pesos e vieses.

Em seguida, elaboramos, com o apoio da ferramenta de IA para geração de texto Claude Sonnet 4.5[21], um conjunto de *scripts* para facilitar a execução dos comandos previamente implementados no APTS e para extrair métricas a partir dos artefatos que ele gera. O primeiro *script*, chamado de `train_model`, é descrito em linguagem fish[22] e realiza o ciclo de treinamento descrito na Seção 2.5.

Esse *script* chama o comando `build-training-memory` com argumentos especificados pelo usuário. Ele deve definir o agente inteligente usado para orientar a simulação de partidas, o que requer sua geração por meio do comando anterior. Seu algoritmo então começa a simular uma série de partidas jogadas segundo o método de MCTS adaptada pelo *AlphaZero*. Ao final, ele guarda, em uma pasta aninhada dentro da pasta do modelo de ResNet, as memórias de partidas e de treinamento geradas.

Para esse comando, definimos a MCTS para realizar 512 ciclos de busca, a um coeficiente de exploração de 1.4, com o fim de explorar suficientemente o espaço de busca, sem comprometer drasticamente o tempo de execução do experimento. Além disso, o agente inteligente utilizou o método de *softmax* a um coeficiente de suavização de 1 para selecionar um movimento avaliado. Também definimos o parâmetro de *seed* como 1.

O segundo passo do ciclo gerenciado pelo *script* `train_model` é executar o comando `train` para gerar um novo modelo de rede neural com pesos e vieses melhor alinhados à memória de treinamento. Seu algoritmo utiliza as ferramentas disponibilizadas bela biblioteca TensorFlow.js para dividir a memória em conjuntos de tamanho fixo e randomizar a ordem desses, a fim de aumentar a variabilidade. Encontramos uma limitação na biblioteca, que não permitiu definir valor de *seed* para esse sorteio. Por fim, é exportada uma nova pasta que contém

[21] Acesso em: https://www.anthropic.com/claude/sonnet.
[22] Acesso em: https://fishshell.com/docs/current/language.html.

os arquivos da ResNet treinada. Sua localização é aninhada à pasta da memória de treinamento, o que forma uma estrutura de árvore de arquivos, em que cada ciclo gera mais um bloco.

Para esse comando, definimos o tamanho da janela de fornecimento dos dados sintéticos como 128 turnos, e determinamos que cada alinhamento deve ser realizado em 16 épocas. Dentro de cada época, 15% dos dados foram utilizados para validação do alinhamento. Por meio de alguns testes preliminares, identificamos que o valor de penalidade para movimentos inválidos não mostrou diferença significativa nas métricas de acurácia do processo de alinhamento, o que nos motivou a defini-lo como 0. Executamos o ciclo de treinamento continuamente, o que resultou na geração de 21 modelos de ResNet consecutivamente treinados até a data de fim do experimento.

O comando `train` salva junto a cada um o registro de métricas de treinamento aferido pelo TensorFlow. Com base nisso, criamos um *script* chamado `analyze_training_logs`, que acessa a pasta do primeiro modelo e busca seus descendentes, que naquela são aninhados. Esse algoritmo lê as métricas de acurácia da *policy head* e da *value head* associadas à última época de treinamento de cada modelo e as salva em memória. Então, ele organiza os modelos em duas listas, ordenadas de forma decrescente para cada uma das métricas. Cada lista é exportada para um arquivo do formato CSV.

Por meio deste artefato, identificamos os melhores modelos de ResNet de acordo com a *policy head*, como disposto na Tabela 1, e de acordo com a *value head*, listados na Tabela 2. Percebemos que o melhor resultado convergiu relativamente cedo, no 4º ciclo de treinamento, para uma acurácia de 0.667935 na predição de movimentos e de 0.557322 na estimativa da qualidade da partida. Após isso, houve leve piora nas métricas, que variaram próximas de 0.571429 para ambas as saídas por vários ciclos, comumente voltando a esse exato valor.

Tabela 1 — Melhores modelos de ResNet ordenados por acurácia da *policy head*.

Ciclo	***Policy head***	***Value head***
4º	0.667935	0.557322
7º	0.571429	0.571429
8º	0.571429	0.571429
13º	0.571429	0.571429
15º	0.571429	0.571429

Fonte: elaborado pelo autor (2026).

Tabela 2 — Melhores modelos de ResNet ordenados por acurácia da *value head*.

Ciclo	***Policy head***	***Value head***
7º	0.571429	0.571429
8º	0.571429	0.571429
13º	0.571429	0.571429
15º	0.571429	0.571429
16º	0.571429	0.571429

Fonte: elaborado pelo autor (2026).

Esses resultados parecem indicar que o agente inteligente é capaz de interpretar o cenário de um turno, ainda que não atinja uma compreensão tão expressiva como esperado. A proximidade das métricas com a faixa de 57% levanta preocupações sobre a capacidade do agente inteligente de reconhecer qual dos jogadores ele deve representar em cada turno. Outra percepção obtida é sobre a duração do processo. Para o jogo Ligue-4, que têm baixa complexidade, é razoável considerar que poucos ciclos de treinamento são necessários, uma vez que após o 7º ciclo não foram observadas melhoras na acurácia.

5.2 SIMULAÇÃO DE PARTIDAS

Após selecionarmos o melhor agente inteligente, avaliamos sua atuação em partidas simuladas do jogo Ligue-4. Para isso, executamos o comando `play-match-using-agent` do APTS, definindo o mesmo modelo de ResNet para orientar ambos os jogadores e configurando o coeficiente de suavização do método de *softmax* como 1.

A fim de diminuir a chance de obter um resultado pouco representativo, definimos um *script* para executar esse comando 100 vezes, variando o valor de *seed* de 1 a 100. Cada execução gera uma pasta que contém o arquivo de histórico da partida, da mesma forma como aquele gerado para construir a memória de treinamento.

Em seguida, criamos um *script* responsável por, para cada partida, ler seu histórico de turnos e coletar em um arquivo auxiliar os seguintes dados: (1) o índice do jogador vencedor, ou `null` caso haja empate; (2) a quantidade de turnos decorridos até o fim da partida; (3) a quantidade de vezes em que o primeiro jogador efetuou cada um dos 7 movimentos; e (4) essa mesma análise para as jogadas do segundo jogador.

Ao fim, esse *script* ainda compila os dados analíticos em um arquivo de estatísticas com um conjunto de informações. A primeira é o cálculo da média, da mediana e do desvio padrão da duração das partidas, como exibido no Quadro 1. Percebemos que a partida mais célere apresentou duração de 15 turnos, ao passo em que a mais longa decorreu por 40 turnos. Comparando todas as simulações, o jogo Ligue-4 tende a ser concluído numa média de 24.32 turnos, apresentando mediana de 23.00 e desvio padrão de 5.94. Considerando o tamanho do tabuleiro de 42 casas e a baixa complexidade do jogo, essas métricas parecem razoáveis.

Quadro 1 — Métricas acerca da duração em turnos de partidas simuladas do jogo Ligue-4.

Mínimo	**Máximo**	**Média**	**Mediana**	**Desv. pad.**
15	40	24.32	23.00	5.94

Fonte: elaborado pelo autor (2026).

Outro dado coletado por aquele *script* é a quantidade de empates e de vitórias de cada jogador ao fim de cada partida. O algoritmo relaciona esse dado com a duração das partidas em turnos, classificada em quatro faixas, como apresentado na Tabela 3.

Essa análise demonstra uma vantagem para o jogador “X” caso ele consiga encerrar o jogo em até 20 turnos, ocasiões em que ele teve 80% de chance de vitória. Caso a partida dure mais, as chances para o jogador “O” se equilibram, ainda que se demonstre uma vantagem notável para o jogador “X”. Análises como essa podem ser especialmente úteis para projetistas de jogos, que devem buscar um equilíbrio entre os jogadores além do número total de turnos.

Tabela 3 — Análise de vitórias dos jogadores segundo faixas de duração de partidas simuladas do jogo Ligue-4.

Duração	**Turnos**	**Jogador “X”**		**Jogador “O”**	
		N	%	N	%
T <= 20	25	20	80%	5	20%
20 < T <= 30	59	35	59%	24	41%

Duração	Turnos	Jogador “X”		Jogador “O”	
		N	%	N	%
30 < T	16	9	56%	7	44%
Total	100	64	64%	36	36%

Fonte: elaborado pelo autor (2026).

Por fim, a ferramenta também registra a frequência de jogada de cada movimento por cada um dos jogadores, como apresentado na Tabela 4. No jogo Ligue-4, esperávamos que os agentes inteligentes privilegiassem a 4ª coluna do tabuleiro, pois ela é a que permite formar mais linhas de peças adjacentes. Entretanto, percebemos que essa hipótese não se concretizou para o experimento simulado. Isso pode indicar vícios no processo de treinamento das ResNets, que não teriam explorado o suficiente estados em que tal coluna levou a vitórias.

Tabela 4 — Análise de movimentos mais jogados por cada agente inteligente em partidas partidas simuladas do jogo Ligue-4.

Jogador	Coluna						
	1ª	2ª	3ª	4ª	5ª	6ª	7ª
Jogador “X”	77	292	67	138	208	112	254
Jogador “O”	41	300	24	205	300	230	84
Total	118	592	91	343	508	342	338

Fonte: elaborado pelo autor (2026).

Quanto aos demais jogos implementados, o Jogo da Velha e o *Snowball*, não executamos o experimento de geração de agentes inteligentes e obtenção de estatísticas por meio de partidas sintéticas. Ainda assim, o Jogo da Velha foi útil para iniciarmos a modelagem da arquitetura do sistema e criarmos os testes de unidade. Em seguida, o *Snowball* teve a relevância de pôr à prova a capacidade de representação de jogos mais complexos e de uso de pontuação durante as partidas. Dessa forma, o APTS na atual versão permite ao usuário jogar esses jogos contra outras pessoas ou contra o algoritmo de MCTS clássica, que também afere a qualidade de movimentos viáveis a partir de um estado fornecido.

6 CONSIDERAÇÕES FINAIS

Este trabalho se tratou de uma pesquisa de natureza aplicada e exploratória que visou o uso dos métodos usados no projeto *AlphaZero* como ferramenta de auxílio no projeto de jogos. Seu objetivo específico é criar um ambiente de representação de protótipos de jogos de turnos com o fim de auxiliar pessoas criadoras de jogos a realizarem a fase de *play-test*. Esse sistema representa jogos de turnos arbitrários e permite a simulação de partidas. Além disso, o programa avalia os movimentos viáveis a partir de um estado por meio do método clássico de MCTS e por meio de agentes inteligentes orientados por ResNets.

A hipótese tomada é que os agentes inteligentes são capazes de realizar a fase de *play-test* de jogo de forma automatizada por meio da geração do histórico de partidas sintéticas, e destacar estatísticas que delas emergem. Dessa forma, os projetistas de jogos de turnos podem reduzir o uso de recursos humanos quando o interesse é realizar testes de estresse e balanceamento. Então, esse estudo pode oferecer perspectivas e ferramentas inovadoras ao cenário de criação de jogos autorais.

Para isso, foi desenvolvido o sistema Sistema de Teste de Jogabilidade Automatizado (APTS) que, de forma geral, foi capaz de viabilizar a representação de jogos de turnos de informação completa e organizados em tabuleiros. Como resultado, foi possível modelar o Jogo da Velha, uma variação autoral dele em um tabuleiro maior chamada de *Snowball*, e ainda o jogo Ligue-4, o que comprovou a viabilidade de representar diferentes estilos de jogo na plataforma.

Esse último jogo foi selecionado para criarmos um agente inteligente orientado pela MCTS clássica, que é executada pelo programa e gera como artefatos a estimativa de qualidade de jogar cada um dos movimentos disponíveis e uma imagem da árvore de busca construída. Essa tecnologia foi aprimorada ao substituir a busca em árvore pela predição de modelos de ResNets, usada para gerar o artefato de estimativa de qualidades do movimentos.

Como objetivo de viabilizar essa técnica, implementamos no APTS métodos responsáveis por ajustar os modelos de rede neural para que suas predições sejam mais acuradas. Nesse sentido, o sistema permite criar uma instância de ResNet inicial e fornecê-la novamente ao programa para que seja usada como método de orientação de um agente inteligente focado em geração de memória de treinamento. Ele usa a MCTS com adaptações que incorporam a ResNet, conforme o projeto *AlphaZero*, para escolher os melhores movimentos em uma série de partidas simuladas.

Algoritmos auxiliares usaram comandos disponibilizados pelo APTS para continuamente alinhar os modelos aos dados por eles próprios gerados num processo de aprendizado por reforço, comumente chamado de *self-play*. Esse processo permite ao usuário do sistema ajustar os parâmetros para criar agentes inteligentes com diferentes estratégias de jogo.

Então, ele é capaz de usar comandos do APTS alinhados a um *script* auxiliar para que tais agentes treinados se enfrentem em uma série de partidas, cujos dados podem ser extraídos em métricas úteis acerca da quantidade de vitórias de cada jogador, da duração do jogo e da predileção por certos movimentos.

Essas métricas demonstram a capacidade de uso do sistema construído para auxiliar no processo de *play-test*, reduzindo a necessidade de testadores humanos nessa fase, ainda

que tenha sido atestada a necessidade de novos estudos e encontradas possíveis melhorias a fazer. Dessa forma, acreditamos ter contribuído diretamente às pessoas projetistas de jogos de tabuleiro autorais, por fornecer uma ferramenta diretamente aplicável ao seu trabalho.

Numa perspectiva maior, esperamos que este trabalho tenha contribuído de forma positiva para o cenário de criação de jogos de turnos autorais, que se encontra em crescimento e requer o estudo de métodos inovadores. Isso se justifica por termos fornecido uma avaliação de hipótese promissora acerca dos métodos abordados, e termos aplicado conceitos de representação de jogos de forma genérica o suficiente para compreender uma variabilidade grande de estilos de jogos.

Contudo, encontramos possíveis problemas no processo de alinhamento das redes neurais aos dados de memória gerados, de forma que não está certo se os agentes inteligentes foram capazes de compreender plenamente como qual dos jogadores eles deveriam atuar em cada turno. Uma proposta de solução razoável é gerar um agente inteligente que atue apenas como um jogador. Entretanto, isso incorreria em maior gasto de recursos e dificultaria o uso do sistema para jogos com uma quantidade grande de jogadores. Essa perspectiva faz necessário investigar formas mais adequadas de representar dados de um estado no formato de canais de números binários, o qual é requerido como entrada da ResNet.

Também é relevante ressaltar a necessidade de mais experimentos variando os parâmetros utilizados em várias fases do processo. Durante a fase de criação de memórias, poderíamos testar valores diversos para a quantidade de ciclos realizados pela MCTS ou o coeficiente de exploração por ela utilizado. Ainda, seria interessante testar diferentes quantidades de simulações de partidas ao gerar as memórias, ou variar o coeficiente de suavização usado pelo método de seleção de movimento por roleta. Já durante a fase de alinhamento de pesos e vieses, é possível utilizar um tamanho diferente para o conjunto de turnos alinhado a cada passo ou, ainda mais relevante, a quantidade de épocas de treino e de ciclos de treinamento, que deixaram de variar significativamente depois de poucas iterações.

Outra questão que não ficou evidente é a determinação do parâmetro de penalização de movimentos inválidos, cujo valor foi dado como 0. Ao mesmo tempo em que seu uso poderia levar a uma convergência mais rápida para os movimentos úteis, um valor muito alto levaria a uma diferença expressiva entre os valores de qualidade calculados para os movimentos bons e o coeficiente, o que resultaria numa aferição alta para a função de perda.

Resta ainda uma reflexão acerca da construção da rede neural utilizada para o jogo simulado, o Ligue-4. Considerando o pequeno espaço de busca de seus movimentos, é possível que uma ResNet com menos blocos residuais e com menor largura de *backbone* compreenda melhor estratégias desse jogo. Nesse sentido, é interessante considerar também se uma estrutura de rede neural mais simples que a ResNet levaria a melhores resultados para espaços de busca pequenos.

Finalmente, destacamos que não foi possível definir um valor de *seed* para o método de alinhamento da rede neural disponibilizado pela biblioteca TensorFlow.js. O processo aleatório do qual ele depende é o sorteio do conjunto de entradas e saídas a alinhar em cada momento. Isso tornou essa etapa de execução não-determinística, o que prejudica a reprodutibilidade dos resultados. Quanto aos demais usos de valores pseudo-aleatórios, certificamo-nos de gerá-los

por meio da *seed* fornecida pelo usuário. Assim, também pode-se realizar mais experimentos variando seu valor.

Outro ponto a explorar é a avaliação da qualidade de um movimento realizada após o fim da construção da árvore de busca. É comum selecionar aquele que levou a mais visitas em seu ramo da árvore, mas isso prejudica movimentos que imediatamente levam a um estado vitorioso, o qual não pode mais ser visitado. Para resolver esse problema, criamos uma função de avaliação que alinha a qualidade estimada da partida com a quantidade de visitas em cada ramo. Contudo, sua especificação foi arbitrária e requer maiores experimentos ou uma mais intensa busca na literatura para substituí-la.

Acerca da representação de jogos, este trabalho apresentou como limitação o suporte apenas a jogos de turnos, o que se justifica pela tradução direta para código-fonte de componentes fundamentais que os definem. Os autores seguiram a convenção de que, a cada turno, pode existir apenas um estágio, no qual a única ação disponível é que o jogador do turno efetue um movimento. Em jogos mais complexos, cada turno pode se dividir em estágios com objetivos diferentes, como primeiramente comprar uma carta do baralho e depois escolher um movimento. Além disso, é possível que outros jogadores atuem dentro do turno que a princípio não está alocado a eles. Essas especificidades podem ser representadas em trabalhos futuros.

Nesse sentido, a necessidade de conhecimento da linguagem JavaScript para implementar as classes concretas e em seguida suas instâncias oferece uma restrição para parte dos usuários. Idealmente, os projetistas não deveriam precisar ter esse conhecimento específico, mas utilizariam uma plataforma com interface gráfica com suporte a navegadores *web*. Então, a descrição dos protótipos deveria ser completamente desconectada da base de código-fonte do sistema. Para isso, poderíamos adaptar o APTS para reconhecer linguagens específicas de domínio, como a *Game Description Language* (GDL)[23] ou a *Zillions by rules files* (ZRF)[24].

Ainda acerca da experiência do usuário, elenca-se como trabalho futuro implementar formas de extração e representação dos dados de *play-test* relevantes ao projetista de forma intuitiva e integrada no sistema. Para testar esse aprofundamento, pode-se utilizar a variante criada para o Jogo da Velha, o *Snowball*, que apresenta um espaço significativo de busca, de 81 movimentos possíveis. Suas regras levam à expectativa de que um jogador atue para prejudicar o domínio de área do oponente no tabuleiro. Assim, pode-se realizar um experimento para verificar se essa impressão se materializa.

Nesse contexto, o sistema atualmente foi testado apenas para jogos de tabuleiro, ainda que os autores tenham tomado o cuidado de estabelecer os componentes de forma abstrata o suficiente para implementar jogos de cartas. Contudo, existe uma complicação para esse tipo de jogo em relação à sua codificação em canais, uma vez que a entrada da ResNet requer uma matriz de três dimensões, o que comumente representa as linhas e colunas do tabuleiro e, em seguida, os canais de dados. Para jogos de cartas, não há uma relação direta entre esses conceitos, o que também abre uma linha de investigação futura.

Ademais, é uma característica comum de jogos de cartas que os jogadores não mostrem aos demais as cartas que seguram em cada turno. Isso os configura seus estados como de infor-

[23] Acesso em: http://logic.stanford.edu/ggp/notes/gdl.html.
[24] Acesso em: https://www.zillionsofgames.com/language.

mação incompleta, o que exige mais estudos acerca da representação desses na forma codificada para a entrada na ResNet.

Outrossim, existe uma preocupação quanto à necessidade de descrever todos os movimentos possíveis de um jogo no momento em que se realiza a sua representação. O jogo Ligue-4, usado no experimento deste trabalho, permitia executar apenas 7 movimentos, o que não constitui um problema. Já o jogo de Xadrez como implementado pelo projeto *AlphaZero* apresenta 4672 movimentos, os quais deveriam ter, cada um, um nome e descrição. Apesar de a maior parte desse número ser devido a combinações das mesmas peças em diferentes situações — cujas instâncias poderiam ser definidas por meio de *scripts* —, ainda é pouco ergonômico para um usuário pensar em todas essas possibilidades antes sequer de testar o protótipo. Por isso, é relevante pesquisar sobre a possibilidade de treinar a rede neural para atribuir qualidades apenas aos movimentos válidos dinamicamente gerados a cada turno.

GLOSSÁRIO

COMPUTAÇÃO

agente inteligente. Sistema capaz de interpretar um estado, tomar decisões autônomas e agir para atingir objetivos definidos, aprendendo a adaptar seu comportamento (Holmgård et al., 2019).

apelido. Em inglês, *alias*. Nome alternativo dado a um tipo de dado, função ou outro elemento de programação para referenciá-lo de forma mais conveniente.

AlphaZero. Algoritmo de autoaprendizado por reforço que combina MCTS e ResNets profundas para dominar jogos de tabuleiro, desenvolvido pelo laboratório Google DeepMind (Silver et al., 2018).

aprendizado de máquina. Em inglês, *machine learning*. Área da IA que desenvolve algoritmos capazes de aprender padrões a partir de dados sem programação explícita, melhorando seu desempenho através da experiência (GeeksforGeeks, 2025a).

aproveitamento. Em inglês, *exploitation*. Componente do critério UCT na MCTS que favorece nós com maior valor médio estimado, aproveitando recompensas já observadas para guiar a seleção (Kocsis; Szepesvári, 2006).

entropia cruzada categórica. Em inglês, *categorical cross-entropy*. Função de perda utilizada em problemas de classificação multi-classe que mede a divergência entre a distribuição de probabilidade prevista pelo modelo e a distribuição real das classes (Li et al., 2022).

erro quadrático médio. Em inglês, *mean squared error*. Função de perda que calcula a média dos quadrados das diferenças entre valores previstos e valores reais, utilizada em problemas de regressão (Li et al., 2022).

exploração. Em inglês *exploration*. Componente do critério UCT na MCTS que prioriza nós pouco visitados, ampliando a busca e evitando convergir cedo demais (Kocsis; Szepesvári, 2006).

fitness. Em português, avaliação. Métrica que quantifica a qualidade de um estado ou solução em relação aos objetivos, atribuindo um valor numérico que orienta a tomada de decisão ou o processo de aprendizado.

linter. Em português, analisador estático de código. Ferramenta que analisa código-fonte para identificar e corrigir problemas de sintaxe, estilo e potenciais defeitos sem executar o programa.

overfitting. Em português, sobre-ajuste. Fenômeno em que um modelo de aprendizado de máquina se ajusta excessivamente aos dados de treinamento, capturando ruído e padrões específicos em vez de generalizar para novos dados, resultando em baixo desempenho para dados não vistos (GeeksforGeeks, 2025b).

perda. Em inglês, *loss*. Métrica que quantifica a discrepância entre as predições de um modelo de aprendizado de máquina e os valores reais esperados (Li et al., 2022).

peso. Em inglês, *weight*. Parâmetro ajustável que pondera a conexão entre neurônios em uma rede neural, determinando a força da influência de uma entrada sobre a saída de uma unidade (Li et al., 2022).

pooling. Em português, agrupamento. Operação em CNNs que reduz a dimensionalidade espacial dos dados, preservando as informações mais relevantes ao selecionar valores representativos de regiões locais (Li et al., 2022).

rede neural. Em inglês, *neural network*. Modelo computacional composto por camadas de unidades interligadas que aprendem padrões em dados por meio de ajustes de pesos (Li et al., 2022).

seed. Em português, semente. Valor inicial fornecido a um gerador de números pseudo-aleatórios para garantir reprodutibilidade dos resultados.

self-play. Em português, autoaprendizado por simulação de partidas. Técnica em que um agente inteligente treina jogando contra versões de si mesmo para aprender estratégias por reforço sem dados externos (Silver et al., 2017).

softmax. Função de ativação que converte um vetor de valores reais em uma distribuição de probabilidade, na qual cada elemento é transformado num valor entre 0 e 1, e a soma de todos os elementos resulta em 1 (Li et al., 2022).

thread. Em português, linha de execução. Unidade básica de processamento que executa instruções de forma independente dentro de um processo. Um programa que opera com mais de uma thread permite que múltiplas tarefas sejam executadas concorrentemente.

vetor. Em inglês, *array*. Estrutura de dados que armazena uma coleção ordenada de elementos acessíveis por índices numéricos sequenciais.

viés. Em inglês, *bias*. Parâmetro aditivo em um neurônio de rede neural que ajusta o limiar de ativação, permitindo que o modelo se adapte melhor aos dados (Li et al., 2022).

JOGOS

casa. Em inglês, *slot*. Unidade discreta que compõe o tabuleiro e pode conter peças ou recursos.

estado. Em inglês, *state*. Representação completa da situação do jogo em um instante, incluindo o conteúdo das casas, os recursos, a pontuação dos jogadores e demais condições vigentes.

jogador. Em inglês, *player*. Participante que toma decisões e executa movimentos conforme as regras do jogo.

jogo. Em inglês, *game*. Sistema de regras que define objetivos, jogadores, movimentos e condições de vitória ou encerramento (Suits, 1967).

jogo de tabuleiro. Em inglês, *board game*. Jogo que utiliza um tabuleiro composto por casas para posicionar peças ou marcadores, onde os movimentos seguem regras espaciais definidas pelo layout do tabuleiro.

jogo de turnos. Em inglês, *turn-based game*. Jogo em que os jogadores atuam de forma alternada em turnos sequenciais, fazendo o estado avançar passo a passo. Neste tipo de jogo, não são permitidos movimentos simultâneos.

movimento. Em inglês, *move*. Ação tomada a partir de um estado que altera as condições atuais, levando a um novo estado.

partida. Em inglês, *match*. Sessão completa do jogo, iniciando nas condições iniciais e terminando quando uma condição de fim é atingida.

play-test. Em português, teste de jogabilidade. Avaliação prática de um jogo com participantes para observar a experiência e coletar feedback de melhoria.

pontuação. Em inglês, *score*. Valor que indica o desempenho de um jogador segundo as regras do jogo.

rodada. Em inglês, *round*. Ciclo completo de turnos no qual todos os jogadores têm a oportunidade de agir uma vez.

turno. Em inglês, *turn*. Período em que um único jogador realiza seus movimentos antes de passar a vez.

REFERÊNCIAS

ABADI, Martín *et al*. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 9 nov. 2016. Disponível em: https://arxiv.org/abs/1603.04467. Acesso em: 13 jan. 2026.

ARAKI, Davi Sadao; KNOP, Igor O. Testes de software e simulações como ferramentas para game design. *In:* **Brazilian Symposium on Computer Games and Digital Entertainment 2020 Proceedings**. Recife, Pernambuco: SBC, 2020.

BECKER, Alexander; GÖRLICH, Daniel. What Is Game Balancing? - An Examination of Concepts. **ParadigmPlus**, v. 1, n. 1, p. 22–41, abr. 2020. DOI: 10.55969/paradigmplus.v1n1a2. Disponível em: https://doi.org/10.55969/paradigmplus.v1n1a2. Acesso em: 16 set. 2023.

BOARDGAME.IO DEVELOPERS. **Concepts**. *[S.l.]:* boardgame.io, 11 out. 2022. Disponível em: https://boardgame.io/documentation/#/. Acesso em: 12 jan. 2026.

BOARDGAMEGEEK, LLC. **SPIEL'25 Preview**. Essen, Germany: BoardGameGeek, LLC, 2025. Disponível em: https://boardgamegeek.com/geekpreview/78/spiel-essen-25-preview. Acesso em: 9 jan. 2026.

BRITANNICA, The Editors of Encyclopaedia. **Go**. *[S.l.]: [S.n.]*, 23 maio 2023. (Nota técnica). Disponível em: https://www.britannica.com/topic/go-game. Acesso em: 3 set. 2023.

CAHN, Lauren. **How to Win Connect 4 Every Time, According to the Computer Scientist Who Solved It**. *[S.l.]:* Reader's Digest, 8 out. 2024. Disponível em: https://boardgamegeek.com/geekpreview/78/spiel-essen-25-preview. Acesso em: 11 jan. 2026.

COHN, David; ATLAS, Les; LADNER, Richard. Improving generalization with active learning. **Machine learning**, v. 15, p. 201–221, 1994. DOI: 10.1007/BF00993277. Disponível em: https://doi.org/10.1007/BF00993277. Acesso em: 31 jan. 2025.

COULOM, Rémi. Efficient selectivity and backup operators in Monte-Carlo tree search. *In:* **International conference on computers and games**. *[S.l.]: [S.n.]*, 2006. Disponível em: https://doi.org/10.1007/978-3-540-75538-8_7. Acesso em: 31 jan. 2025.

DUMONT, Alberto Santos. **O que eu vi, o que nós veremos**. 1. ed. São Paulo: Wikisource, 1918. Disponível em: https://pt.wikisource.org/wiki/O_que_eu_vi,_o_que_n%C3%B3s_veremos. Acesso em: 2 ago. 2025.

ESLINT CONTRIBUTORS. **Core Concepts**. Disponível em: https://eslint.org/docs/latest/use/core-concepts/. Acesso em: 9 jan. 2026.

FULLERTON, Tracy. **Game Design Workshop: A Playcentric Approach to Creating Innovative Games**. 4. ed. Boca Raton: CRC Press, 2019

FÖRSTER, Robert. **AlphaZero from Scratch**. *[S.l.]: [S.n.]*, 2023. Disponível em: https://github.com/foersterrobert/AlphaZeroFromScratch. Acesso em: 6 jan. 2026.

GEEKSFORGEEKS. **Machine Learning Algorithms**. *[S.l.]:* GeeksforGeeks, 18 nov. 2025a. Disponível em: https://www.geeksforgeeks.org/machine-learning/machine-learning--algorithms/. Acesso em: 12 jan. 2026.

GEEKSFORGEEKS. **Underfitting and Overfitting in ML**. *[S.l.]:* GeeksforGeeks, 10 dez. 2025b. Disponível em: https://www.geeksforgeeks.org/machine-learning/underfitting-and--overfitting-in-machine-learning/. Acesso em: 12 jan. 2026.

GUDMUNDSSON, Stefan Freyr *et al*. Human-like playtest with deep learning. *In:* **2018 IEEE Conference on Computational Intelligence and Games (CIG)**. *[S.l.]: [S.n.]*, 2018. Disponível em: https://doi.org/10.1109/CIG.2018.8490442. Acesso em: 31 jan. 2025.

HE, Kaiming *et al*. Deep Residual Learning for Image Recognition. 2015. DOI: 10.1109/CVPR.2016.90. Disponível em: https://doi.org/10.1109/CVPR.2016.90. Acesso em: 31 jan. 2025.

HOLMGÅRD, Christoffer *et al*. Automated playtest With Procedural Personas Through MCTS With Evolved Heuristics. **IEEE Transactions on Games**, v. 11, n. 4, p. 352–362, 2019. DOI: 10.1109/TG.2018.2808198. Disponível em: https://doi.org/10.1109/TG.2018.2808198. Acesso em: 31 jan. 2025.

KOCSIS, Levente; SZEPESVÁRI, Csaba. Bandit Based Monte-Carlo Planning. *In:* **Machine Learning: ECML 2006**. FÜRNKRANZ, Johannes; SCHEFFER, Tobias; SPILIOPOULOU, Myra (orgs.). Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. Disponível em: https://doi.org/10.1007/11871842_29. Acesso em: 7 jan. 2026.

LI, Zewen *et al*. A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects. **IEEE Transactions on Neural Networks and Learning Systems**, v. 33, n. 12, p. 6999–7019, 2022. DOI: 10.1109/TNNLS.2021.3084827. Disponível em: https://doi.org/10.1109/TNNLS.2021.3084827. Acesso em: 31 jan. 2025.

LIANG, Jiazhi. Image classification based on RESNET. **Journal of Physics: Conference Series**, v. 1634, p. 12110, 2020. DOI: 10.1088/1742-6596/1634/1/012110. Disponível em: https://doi.org/10.1088/1742-6596/1634/1/012110. Acesso em: 6 fev. 2025.

MALOSTO, Celso Gabriel Dutra Almeida; CAMPOS, Luciana Conceição Dias; KNOP, Igor de Oliveira. Moving towards automated game play-testing. *In:* **Anais Estendidos do XXIV Simpósio Brasileiro de Jogos e Entretenimento Digital**. Salvador, Bahia: SBC, 2025. Disponível em: https://sol.sbc.org.br/index.php/sbgames_estendido/article/view/37117. Acesso em: 6 jan. 2026.

MALOSTO, Celso Gabriel Dutra Almeida; KNOP, Igor Oliveira; CAMPOS, Luciana Conceição Dias. AlphaZero como ferramenta de playtest. **Revista ComInG - Communications and Innovations Gazette**, v. 7, n. 1, p. 39–50, 2023. DOI: 10.5902/2448190485269. Disponível em: https://doi.org/10.5902/2448190485269. Acesso em: 31 jan. 2025.

MALOSTO, Celso Gabriel; KNOP, Igor. **Repositório do projeto APTS**. *[S.l.]:* GitHub, 2026. Disponível em: https://github.com/ufjf-gamelab/apts. Acesso em: 9 jan. 2026.

MARCELO, Antonio; PESCUITE, Júlio. **Design de jogos: Fundamentos**. 1. ed. Rio de janeiro: Brasport, 26 mar. 2009. p. 188

NAIR, Vinod; HINTON, Geoffrey E. Rectified linear units improve restricted boltzmann machines. *In:* **Proceedings of the 27th International Conference on International Conference on Machine Learning**. ICML'10. Haifa, Israel: Omnipress, 2010. Disponível em: https://dl.acm.org/doi/10.5555/3104322.3104425. Acesso em: 31 jan. 2025.

NODE.JS. **Introduction to Node.js**. *[S.l.]:* OpenJS Foundation, 23 jul. 2025. Disponível em: https://nodejs.org/en/learn/getting-started/introduction-to-nodejs. Acesso em: 9 jan. 2026.

PLAAT, Aske *et al*. **A Minimax Algorithm Better Than Alpha-beta?: No and Yes**. Edmonton, Alberta, Canada: The University of Alberta, 6 jul. 1995. Disponível em: https://arxiv.org/abs/1702.03401. Acesso em: 15 jan. 2026.

RANANDEH, Vahid; MIRZA-BABAEI, Pejman. Beyond Equilibrium: Utilizing AI Agents in Video Game Economy Balancing. *In:* **Companion Proceedings of the Annual Symposium on Computer-Human Interaction in Play**. *[S.l.]: [S.n.],* 2023. Disponível em: https://doi.org/10.1145/3573382.3616092. Acesso em: 31 jan. 2025.

ROMERO, Brenda; SCHREIBER, Ian. **Game Balance**. 1st edition ed. Boca Raton: CRC Press, 2021

SALEN, Katie; ZIMMERMAN, Eric. **Rules of Play: Game Design Fundamentals**. Cambridge: MIT Press, 2003. p. 688. Disponível em: https://mitpress.mit.edu/9780262240451/rules-of-play/. Acesso em: 31 jan. 2025.

SILVER, David *et al*. Mastering the Game of Go with Deep Neural Networks and Tree Search. **Nature**, v. 529, n. 7587, p. 484–489, jan. 2016. DOI: 10.1038/nature16961. Disponível em: https://doi.org/10.1038/nature16961. Acesso em: 16 set. 2023.

SILVER, David *et al*. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. 2017. DOI: 10.48550/arXiv.1712.01815. Disponível em: https://doi.org/10.48550/arXiv.1712.01815. Acesso em: 16 set. 2023.

SILVER, David *et al*. A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go through Self-Play. **Science**, v. 362, n. 6419, p. 1140–1144, 2018. DOI: 10.1126/science.aar6404. Disponível em: https://doi.org/10.1126/science.aar6404. Acesso em: 16 set. 2023.

STAHLKE, Samantha; NOVA, Atiya; MIRZA-BABAEI, Pejman. Artificial players in the design process: Developing an automated testing tool for game level and world design. *In:* **Proceedings of the Annual Symposium on Computer-Human Interaction in Play**. *[S.l.]: [S.n.]*, 2020. Disponível em: https://doi.org/10.1145/3410404.3414249. Acesso em: 31 jan. 2025.

SUITS, Bernard. What is a Game?. **Philosophy of Science**, v. 34, n. 2, p. 148–156, 1967. DOI: 10.1086/288138. Disponível em: https://doi.org/10.1086/288138. Acesso em: 31 jan. 2025.

TEUBER, Klaus. **Colonizadores de Catan**. Disponível em: https://boardgamegeek.com/boardgame/13/catan. Acesso em: 31 jan. 2025.

TRZEWICZEK, Ignacy. **I play-tested it 100 times**. *[S.l.]:* Portal Games, 22 jun. 2017. Disponível em: https://trzewik.medium.com/i-play-tested-it-100-times-fcb142c38c80. Acesso em: 7 set. 2023.

TYPESCRIPT TEAM. **TypeScript for JavaScript Programmers**. *[S.l.]:* TypeScript, 7 jan. 2026. Disponível em: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html. Acesso em: 9 jan. 2026.

WALLNER, Günter; HALABI, Nour; MIRZA-BABAEI, Pejman. Aggregated visualization of playtest data. *In:* **Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems**. *[S.l.]: [S.n.]*, 2019. Disponível em: https://doi.org/10.1145/3290605.3300593. Acesso em: 31 jan. 2025.

WOODS, Stewart. **Eurogames: The Design, Culture and Play of Modern European Board Games**. Jefferson: McFarland, Incorporated, Publishers, 2012. p. 262

ZOOK, Alexander; FRUCHTER, Eric; RIEDL, Mark O. Automatic playtest for Game Parameter Tuning via Active Learning. 2019. DOI: 10.48550/arXiv.1908.01417. Disponível em: https://doi.org/10.48550/arXiv.1908.01417. Acesso em: 31 jan. 2025.

ŚWIECHOWSKI, Maciej *et al*. Monte Carlo Tree Search: a review of recent modifications and applications. **Artificial Intelligence Review**, v. 56, n. 3, p. 2497–2562, jul. 2022. DOI: 10.1007/s10462-022-10228-y. Disponível em: https://doi.org/10.1007%2Fs10462-022-10228--y. Acesso em: 16 set. 2023.