COMMUNICATIONS AND INNOVATIONS GAZETTE

AlphaZero como ferramenta de playtest

Celso Gabriel D. A. Malosto[1], Luciana Campos[1], Igor de Oliveira Knop[1]

[1]Grupo de Educação Tutorial em Sistemas de Informação,
Universidade Federal de Juiz de Fora (UFJF)
Juiz de Fora – MG – Brasil

gabriel.malosto@estudante.ufjf.br, luciana.campos@ice.ufjf.br,
igorknop@ice.ufjf.br

Abstract. *The tabletop gaming market has maintained rapid growth in recent years, reaching tens of billions of dollars. Balancing these games is a market demand and a challenging discipline that requires significant analytical skill from the game designer. This skill is built through the conduct and observation of hundreds of playtests in test groups, in a error-prone process, due to the difficulty in finding individuals willing to play prototypes repeatedly, with only minor rule iterations. Additionally, not every game designer keeps a complete record of the tests and can handle the cause-and-effect relationships that small rule changes have on the outcomes. This work investigates AlphaZero as a computational intelligence technique to alleviate the demand for human players in the game creation and testing process. While existing research in the field seeks a more efficient agent, this study applies the algorithm to generate a dataset to help the game designer identify points of imbalance and explore creativity. At the current stage, a version of AlphaZero is implemented for self-training. Finally, we discuss areas in which the method can assist the game design process using the data generated by the algorithm during training.*

Resumo. *O mercado de jogos de mesa tem mantido acelerado crescimento nos últimos anos e já atinge as dezenas de bilhões de dólares. O balanceamento desses jogos é uma demanda de mercado e é uma disciplina difícil, que exige uma grande habilidade analítica por parte do game designer. Essa habilidade é construída com a condução e observação de centenas de partidas em grupos de teste, em um processo propenso a erro, devido à grande dificuldade em encontrar pessoas dispostas a jogar protótipos repetidamente, apenas com pequenas iterações nas regras. Adicionalmente, nem todo game designer mantém um registro completo dos testes e consegue lidar com as relações de causa e efeito que pequenas mudanças nas regras causam nos resultados. Este trabalho investiga o AlphaZero, como técnica de inteligência computacional para aliviar a demanda por jogadores humanos no processo de criação e testes de jogos. Enquanto os trabalhos na área buscam um agente mais eficiente, este aplica o algoritmo para gerar um conjunto de dados para auxiliar o game designer a encontrar pontos de desequilíbrio e explorar a criatividade. No atual estágio, uma versão do AlphaZero é implementada para realizar o autotreinamento. Por fim, são discutidos pontos nos quais o método pode auxiliar o processo de game design utilizando os dados gerados pelo algoritmo durante o treinamento.*

1. Introdução

O mercado dos jogos de tabuleiros 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 estes tipos de jogos, sendo o maior o BoardGameGeek (BOARDGAMEGEEK, 2023), que registra mais de 140 mil itens entre jogos e expansões. Apesar de variarem, um muitos desses se destacam pelo seu perfil tático ou estratégico durante as partidas, com uma série de reações em cadeia para as ações escolhidas e diversas dinâmicas emergentes pelas decisões dos jogadores. Estes jogos, conhecidos como *designer's games* por trazerem o nome do autor na capa, são fruto de uma organização dos criadores que traz benefícios para um mercado baseado em novidades (WOODS, 2012). Anualmente, entre 500 e 600 novos jogos são apresentados nas maiores convenções do meio, além de reimpressões e reedições (BOARDGAMEGEEK, 2022).

O processo de criação de um jogo é iterativo. O *game designer* implementa a sua ideia em um protótipo, de preferência de baixo custo para facilitar as modificações necessárias. Assim que fica pronto, o jogo deve ser testado, realizando partidas para explorar o comportamento dos sistemas e encontrar possíveis desequilíbrios. Adicionalmente, são realizados testes de estresse, como realizar a mesma ação durante quase toda a partida caso aparente ser muito vantajosa (MARCELO; PESCUITE, 2009; FULLERTON, 2019). 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, mas com agrupamentos em equilíbrio matemático, de dificuldade, de progressão, de estratégias e imparcialidade. 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).

Esta etapa de *playtesting*, na qual o jogo é jogado repetidamente, tem alto custo de recursos humanos. É difícil manter o grupo de *playtest* ativo e focado, dado que se trata de um processo exaustivo e cujo objetivo nem sempre é claro para os jogadores (TRZEWICZEK, 2017). Outrossim, as ações dos próprios testadores podem influenciar nos resultados dos testes com suas expectativas, humores, excessos ou falta de concentração. Esses são pontos importantes a se observar em um teste (MARCELO; PESCUITE, 2009), exceto quando os objetivos são equilíbrios ou testes de estresse.

Este trabalho apresenta os primeiros passos de uma pesquisa exploratória para investigar relações de balanceamento em jogos durante sua criação. É proposto um ambiente de *playtest* simulado para auxiliar o autor a realizar as primeiras iterações do processo sem a necessidade de testadores humanos. Dessa forma, espera-se que a participação de pessoas seja empregada para investigar aspectos lúdicos e a experiência do jogador, ao passo em que os testes repetitivos e a geração de massa de dados para análise serão realizados majoritariamente por agentes inteligentes[1]. Tal objetivo é apoiado por heurísticas e técnicas de inteligência computacional, entre as quais se destacam a *Monte Carlo Tree Search* (MCTS) e o AlphaZero, que têm sido utilizadas por realizar buscas eficientes na árvore de decisão através de aprendizado profundo. Assim, a partir de aprendizado não informado, essas tecnologias poderão ser utilizadas para jogos em desenvolvimento.

[1]Na área de Inteligência Artificial, um agente inteligente é uma entidade autônoma capaz de observar um ambiente através de sensores e atuar sobre este através de atuadores.

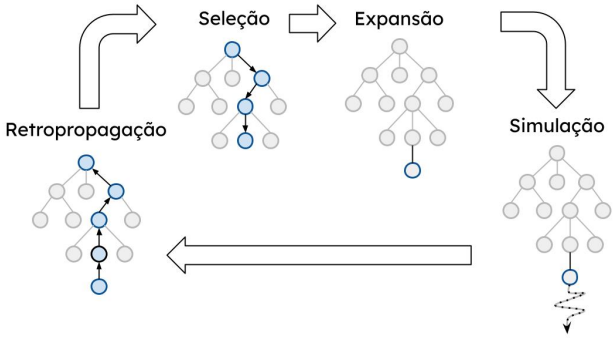


Figura 1: Ciclo da Árvore de Busca Monte-Carlo: seleção, expansão, simulação e retropropagação. Adaptado de Świechowski et al. (2022)

Esse trabalho é organizado da seguinte forma: na Seção 2 são apresentados conceitos importantes para o entendimento do trabalho; na Seção 3 é apresentado o ambiente de *playtests*; na Seção 4 são apresentados os resultados obtidos; e o trabalho se encerra com a Seção 5, que apresenta as considerações finais e sugestões de trabalhos futuros.

2. Fundamentação Teórica

2.1. Monte Carlo Tree Search - MCTS

O MCTS é um algoritmo de decisão que usa os nós de uma árvore para representar os estados de um jogo, e suas arestas para representar ações que realizam as transições entre estados (KOCSIS; SZEPESVÁRI, 2006; COULOM, 2006). A raiz da árvore mantém o estado inicial do jogo e cada nível representa alternadamente os possíveis estados daquele para cada ação tomada por um dos jogadores. Isso possibilita saber o movimento do oponente para prever a melhor ação futura (ŚWIECHOWSKI et al., 2022).

O processo de busca objetiva encontrar as melhores sequências de jogadas, e constitui-se de quatro etapas, conforme visto na Figura 1. A primeira é a de seleção, que procura, a partir da raiz, o melhor nó folha com base na diretriz definida. A mais comum é a *Upper Confidence Bounds* (UCB), que atribui a cada nó contadores de visita e de vitória, calculando uma equação cujo resultado alinha descoberta (*exploration*) e exploração (*exploitation*). Então, se o nó selecionado não representar fim de jogo, executa-se a fase de expansão, que cria o resultante da ação aplicada sobre aquele. Segue a fase de simulação, realizando ações aleatórias com base no novo nó até que o jogo termine. Invariavelmente, na fase de retropropagação, atualizam-se as estatísticas de todos os nós na trilha selecionada. A busca finaliza ao atingir o número de iterações determinado pelos parâmetros.

2.2. Modelo AlphaZero

Go é um jogo para 2 jogadores, originado na China, que é composto por um tabuleiro de 19 linhas verticais e horizontais, 180 peças brancas e 180 peças pretas. Uma partida se inicia com o tabuleiro vazio. Em cada turno, os jogadores se alternam colocando uma de suas peças nas intersecções das linhas horizontais e verticais. Qualquer intersecção não ocupada é válida para colocação das peças, que então não podem ser movidas. O objetivo é cercar totalmente as peças adversárias, pois, quando totalmente cercadas, essas são removidas do tabuleiro. Vence o jogador que, ao se esgotarem todos os movimentos, tiver a maior quantidade de peças no tabuleiro (BRITANNICA, 2023).

O *AlphaGo* é um modelo de rede neural desenvolvido pela empresa *DeepMind*, braço de pesquisa em IA da *Google*, cujo objetivo é jogar Go em nível competitivo. A primeira versão treinava o modelo ao jogar contra pessoas. Foi então desenvolvida a evolução *AlphaGo Zero*, que acumula dados de treinamento jogando contra si mesma, a partir de pesos e movimentos aleatórios, até alcançar desempenho excepcional (SILVER et al., 2016). O modelo foi então generalizado para aprender qualquer jogo de tabuleiro, dadas apenas as regras, o que se denominou AlphaZero. O programa recebe o estado atual do tabuleiro como *input*, e retorna como *output* um vetor de probabilidades para cada ação que pode ser tomada, e um escalar que representa o resultado estimado. Os pesos e vieses são lapidados ao fazer o agente jogar partidas contra si mesmo, em um processo de aprendizado por reforço (SILVER et al., 2017; SILVER et al., 2018).

Para tanto, o MCTS é associado a uma rede neural residual (do inglês, *Residual Neural Network* (ResNet)). Essa classe de modelos de IA é derivada das redes neurais convolucionais, e aplicada primariamente em domínios de reconhecimento de imagens. A ResNet tem o objetivo de gerar um modelo que responda bem a entradas diversas, ao criar uma rede de conexões profunda, mas garantindo baixa perda de precisão (HE et al., 2015). Sua estrutura consiste de sucessivos blocos de camadas convolucionais e normalizações, nas quais a função de ativação utilizada é a *Rectified Linear Unit* (ReLU). Ao final de cada bloco, é reaplicada a camada de entrada inicial, o que mantém intensos os detalhes da imagem. O AlphaZero utiliza uma adaptação desse conceito, ao representar o estado do jogo similarmente a uma imagem de três canais de cor.

2.3. Trabalhos Relacionados

Nos trabalhos de Gudmundsson et al. (2018) e Zook, Fruchter e Riedl (2020), o MCTS é utilizado junto a redes neurais convolucionais para avaliar a dificuldade de missões em jogos digitais *match 3*, respectivamente *Candy Crush* e *Jewels Star Story*. Os trabalhos conseguem reproduzir comportamentos de jogadores humanos e avaliar a dificuldade do nível proposto pelo *game designer* para uma melhor experiência de jogo. Zook, Fruchter e Riedl (2020) realizam um estudo combinando técnicas de regressão e classificação para realizar uma aprendizagem ativa de um jogo *shoot'em up* cuja mecânica é bem definida, mas os parâmetros como velocidades de jogador, inimigos e tiros são ajustados através de testes exaustivos, aqui substituídos pelo *playtest* automatizado.

Sob a ótica de comunicação dos dados gerados ao designer, Wallner, Halabi e 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. Similarmente, Stahlke, Nova e Mirza-Babaei (2020) usa 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 e Mirza-Babaei (2023).

Apesar dos trabalhos de testes serem em sua maioria referentes a jogos digitais (normalmente modelados sistemas em tempo contínuo), acreditamos que as mesmas técnicas podem ser aplicadas aos jogos físicos (modelados por sistemas discretos). A escassez de trabalhos nesta indústria nos motiva a realizar esta investigação, buscando avaliar as limitações ou ajustes necessários para sua implantação.

3. Ambiente de Playtests APTS

Este trabalho é precursor no desenvolvimento do sistema Auto Playtest System (APTS). Essa aplicação deve descrever e simular diversos jogos discretos, digitais ou de mesa, inicialmente para dois jogadores, a fim de coletar dados e ações de partidas jogadas. Estes serão utilizados para gerar informações estatísticas com o fim de auxiliar o *game designer* e diminuir o esforço humano em etapas de *playtest*, sobretudo as que envolvem testes de estresse e balanceamento, em que a experiência do jogador não é a variável principal.

A simulação computacional requer a implementação das regras na forma de um programa, que deve ser desacoplado do código do sistema, a fim de expor as ações disponíveis para um determinado estado do jogo devidamente representado. Os métodos necessários são descritos pela interface `Game`, na Figura 2. O sistema permite que o computador jogue contra si mesmo, evoluindo sua habilidade e construindo um histórico de jogadas, usado para treinar modelos de rede neural, a partir dos quais será possível obter características do jogo sob teste e demais informações relevantes para o desenvolvedor.

3.1. A aplicação do APTS

O propósito do APTS é criar um ambiente expansível e, por isso, o software e a modelagem das regras foram desenvolvidos na linguagem de programação JavaScript. Isso se justifica por esta ser popular e acessível, além de permitir executar parte da aplicação em navegador web futuramente. Nesta etapa do projeto, o objetivo do sistema é a geração de dados para subsequente análise, não sendo necessária uma interface com o usuário avançada. Decidiu-se então usar o ambiente de execução JavaScript Node.JS[2] para rodar o programa em terminal de linha de comando. Para representar os tabuleiros e as ações do jogo e realizar a exibição de dados de forma legível, foi utilizada a biblioteca *Ink*[3], que permite utilizar, em modo texto, métodos de desenvolvimento web, como programação em componentes React, o que garante uma boa visualização e controles de uso.

A implementação de referência do APTS foi baseada em uma ResNet aplicada ao MCTS, implementada com a biblioteca PyTorch, da linguagem de programação Python. Para este trabalho, foi escolhida a biblioteca de aprendizado de máquina TensorFlow.JS[4], que apresenta código aberto, é aplicável a uma variedade de tarefas, e em que se pode modelar e treinar redes neurais com a linguagem JavaScript. Assim, verifica-se ampla utilização no meio de ciência de dados, além de apresentar boa documentação, exemplos de aplicação, tutoriais e demais materiais relacionados (TENSORFLOW, 2023?).

3.2. Implementação do jogo modelado

Seguindo a implementação de referência (FöRSTER, 2023), o projeto teve início com a codificação do modelo do Jogo da Velha (em inglês, *Tic-Tac-Toe*). Este foi escolhido por gerar espaço de busca pequeno, ter regras simples, ser de informação completa e para dois jogadores, o que o torna excelente para validar a o sistema nesta primeira versão. A Figura 2 apresenta um resumo do diagrama de classes implementado no sistema APTS, com seus relacionamentos, interfaces, implementações, e estruturas auxiliares.

[2]Disponível em: ⟨https://nodejs.org⟩
[3]Disponível em: ⟨https://github.com/vadimdemedes/ink⟩
[4]Disponível em: ⟨https://www.tensorflow.org/js⟩.

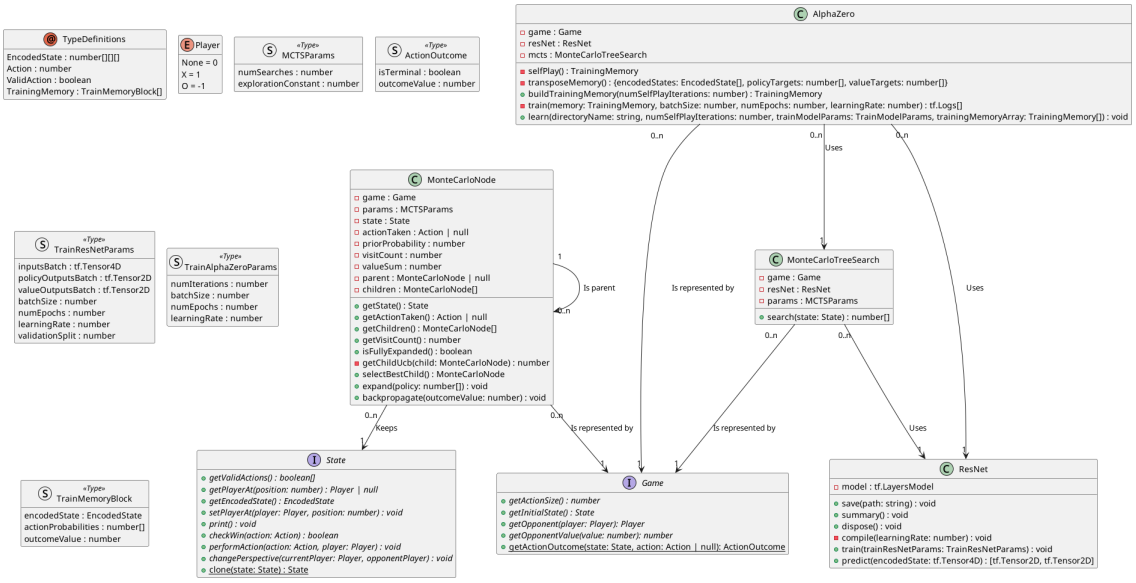


Figura 2: Diagrama de classes do Auto Playtest System (APTS).

Os classes concretas do jogo da velha definem atributos que guardam o tamanho do tabuleiro, específicas para este jogo. Além disso, elas implementam os métodos das interfaces `Game` e `State`, os quais manipulam um estado, e retornam informações segundo as regras codificadas. Percebe-se então que é possível utilizar a mesma lógica para implementar diferentes jogos. Para iniciar uma partida, deve-se chamar o método `getInitialState()`, que retorna uma matriz preenchida com zeros, cujas dimensões equivalem àquelas definidas pelos atributos da classe. Em seguida, pode-se realizar um movimento sobre determinado estado, ao chamar a função `performAction(action, player)`. Esta é responsável por colocar o símbolo de um jogador em determinada posição no tabuleiro, indicada pelo parâmetro `action`.

A cada turno da partida, deve-se verificar as ações válidas disponíveis. Neste caso, trata-se das posições em que um jogador pode marcar seu símbolo. Para isso, o método `getValidActions()` retorna um vetor de tamanho equivalente ao número de espaços do tabuleiro, em que cada posição é definida como verdadeiro, caso nenhum jogador tenha posicionado seu símbolo nela, ou falso, caso contrário. Sendo válida a ação escolhida, esta é executada, ao que se chama o método `getActionOutcome(state, action)`. Este é responsável por determinar se o jogo chegou a um estado terminal e qual o marcador de sucesso da ação, se vitória ou derrota, cuja estrutura é representada na Figura 2. Nessa implementação, um caso de empate é considerado como uma derrota.

3.3. Implementação da busca em árvore

A fim de que o computador sugira melhores ações, foi implementada a MCTS. Sua classe `MonteCarloNode` guarda: um estado do jogo; a ação que resultou nele; os nós filhos; e as ações disponíveis a partir do estado; além dos indicadores de visitas a si, e de vitórias e derrotas ocorridas ao percorrer um caminho no qual esteja incluso. Ademais, a classe `MonteCarloTreeSearch` é responsável por realizar a busca da MCTS, pelo método `search(state)`. Este realiza os passos da busca quantas forem as vezes determinadas pelos parâmetros, e retorna um vetor das probabilidades de vitória para cada ação.

A cada iteração, faz-se uma busca por um nó que ainda não tenha expandido em todas as ações possíveis. Isso é feito dentre os filhos, sendo selecionado aquele com o melhor UCB, utilizando a função `getChildUcb(child)`. Esta calcula a Equação 1a, que é dada pela soma de Equação 1b e Equação 1c. Após selecionado o filho, verifica se suas expansões já foram exauridas. Este processo é repetido até que se encontre um nó viável para expandir. Então, recupera-se o resultado da ação tomada para chegar ao estado daquele nó. Se a ação não finaliza a partida, são executadas as fases de expansão e simulação. Independentemente do resultado, é realizada a fase de retropropagação.

$$UCB = Exploitation + Exploration \tag{1a}$$

$$Exploitation = 1 - \frac{\frac{child.valueSum}{child.visitCount} + 1}{2} \tag{1b}$$

$$Exploration = expConst * \sqrt{\frac{\ln{(this.visitCount)}}{child.visitCount}} \tag{1c}$$

O método `expand()` clona o estado do nó, seleciona uma ação aleatória dentre aquelas ainda não tomadas sobre si, e a aplica sobre o estado copiado. Nota-se que os nós da MCTS não guardam qual jogador executou a ação. Essa informação é derivada de seu nível na árvore: os níveis K representam a perspectiva do primeiro jogador; e os níveis $K + 1$ representam a de seu oponente. Assim, para orientar os cálculos, um estado é codificado como na Figura 3b: o valor 1 indica as posições onde o jogador em questão posicionou seu símbolo, o valor 0 indica as posições sem marcações, e o valor -1 indica as posições ocupadas pelo oponente. Dessa forma, após aplicar a ação, todos os valores dessa matriz de estado devem ser invertidos, num processo chamado troca de perspectiva. Por fim, é criado um novo nó, que guarda o novo estado e a ação aleatória escolhida. Esse é adicionado à lista de filhos do nó cujo método de expansão foi chamado.

O	O	
	X	
	X	

(a) Estado do jogo.

-1	-1	0
0	1	0
0	1	0

(b) Estado pela perspectiva do jogador X.

Figura 3: Modelo do estado do jogo no MCTS.

Então, é chamado o método `simulate()` do nó filho. Inicialmente, obtém-se o resultado da ação escolhida, e troca a perspectiva do seu valor. Caso a ação termine o jogo, o indicador de vitória ou derrota é logo retornado para a retropropagação. Caso contrário, o estado do nó é copiado para um auxiliar, e sobre ele são efetuadas jogadas alternadas entre os dois jogadores, até que se chegue a um estado terminal. Da mesma forma, é retornado o indicador de vitória, cuja perspectiva é trocada caso seja o oponente quem finalizou o jogo. Este indicador é utilizado pelo método `backpropagate(outcomeValue)`, o qual recursivamente, da folha até a raiz, incrementa o contador de visitas e atualiza o índice de vitórias. Dada a propriedade alternada dos níveis da MCTS, o índice é invertido a cada passo recursivo, e esse valor é somado àquele guardado no nó.

Esse processo de busca é realizado repetidas vezes pois, a cada iteração, a árvore estará mais desenvolvida, o que deve levar a uma melhor exploração do espaço de busca.

Após findadas as iterações, é instanciado um vetor das probabilidades para cada ação. É tomada a raiz como referência, e cada posição é calculada como: o número de visitas ao filho resultante daquela ação, dividido pelo número total de visitas aos nove filhos (para o jogo da velha). Esse vetor é retornado como resultado do método `search(state)`.

3.4. Criação do modelo de ResNet

A fim de aprimorar a qualidade da busca e viabilizar a simulação de jogos mais complexos, optou-se pelo modelo de uma *Residual Neural Network* (ResNet). O APTS gera a rede de forma dinâmica, permitindo escolher o número de canais e a quantidade de blocos residuais para cada modelo treinado. Para isso, foi criada a classe `ResNet`, cujo objetivo é encapsular um modelo e servir como interface entre a rede neural e o programa que a utilizará. Assim, pode-se realizar alterações na tecnologia sem impactar a estrutura geral do código. Essa classe guarda um objeto do tipo `tf.LayersModel`, do TensorFlow.JS, que representa um modelo de rede neural estruturado em sucessivas camadas, sem ciclo, em que, por padrão, já são implementados métodos de ajuste e predição de dados.

`ResNet` disponibiliza o método `predict(state)`, que encapsula aquele homônimo do `tf.LayersModel`, para controlar o acesso aos dados, e garantir um bom gerenciamento de memória. Já a função `train(params)` recebe uma série de parâmetros de treinamento e ajusta o modelo ao conjunto de dados históricos fornecido, processo que é gerenciado em grande parte pelo TensorFlow, por seu método `fit()`. Para que um objeto `ResNet` seja criado, é necessário carregar algum modelo, ou especificar os parâmetros a fim de que um novo seja gerado. Neste segundo caso, é chamada a função `buildResNetModel(game, numResBlocks, numHiddenChannel)`. Esta, inicialmente, define o formato da camada de entrada de dados, codificada como a Figura 4. Os canais da imagem vermelho, verde e azul representam, respectivamente, o segundo jogador (símbolo O), as posições sem jogadas realizadas (espaços vazios), e o primeiro jogador (símbolo X). Assim, para o canal vermelho, o valor 1 representa que certa posição é ocupada pelo símbolo O do segundo jogador, enquanto as demais posições são preenchidas com 0. O mesmo é feito para as posições vazias e para o primeiro jogador.

1	1	0
0	0	0
0	0	0

(a) Canal vermelho para o jogador O.

0	0	1
1	0	1
1	0	1

(b) Canal verde para casas vazias.

0	0	0
0	1	0
0	1	0

(c) Canal azul para o jogador X.

Figura 4: Estado da partida codificado em mapa de cores pela perspectiva do jogador X.

Sucede-se então uma camada de transformação inicial, que aplica convolução e normalização, preparando o tensor para entrar na *backbone* da ResNet. Esta, por sua vez, é construída por sucessivas aplicações de blocos residuais, pela função `applyResBlock`. Cada bloco aplica convolução e normalização duas vezes, e então soma ao resultado uma cópia daquele tensor gerado logo após a transformação inicial. Por fim, são criadas duas camadas de retorno para o modelo, chamadas de `heads`. A *policy head* retorna um tensor de probabilidades de vitória para cada uma das ações possíveis, ou seja, para cada posição do tabuleiro. Já a *value head* retorna o índice de sucesso a partir do estado fornecido.

3.5. Aprimoramento da MCTS

Foi possível melhorar a eficiência da busca da MCTS, ao utilizar o modelo de ResNet. Depois de selecionar o melhor nó para expandir, seu estado é codificado, e utilizado como *input* do método `predict(encodedState)`, da `ResNet`. É aplicada uma máscara sobre o tensor de probabilidades, para definir com valor 0 as posições inválidas. Este então é utilizado na função adaptada `expand(policy)`, que cria todos os filhos viáveis para o nó selecionado, em vez de apenas um. Uma vez que o modelo treinado deve ser capaz de prever as melhores jogadas, não é necessário mais aplicar a fase de simulação. Então, aplica-se diretamente a retropropagação sobre o índice retornado pelo modelo.

Para que este processo funcione, foram realizadas algumas modificações na classe `MonteCarloNode`. Criou-se um novo atributo chamado `priorProbability`, que guarda a probabilidade dada pelo modelo de `ResNet` para aquele nó quando ele foi gerado. A fórmula do método `getChildUcb(child)` na Equação 2a também foi alterada, para levar em conta este parâmetro, como demonstrado na Equação 2b e Equação 2c.

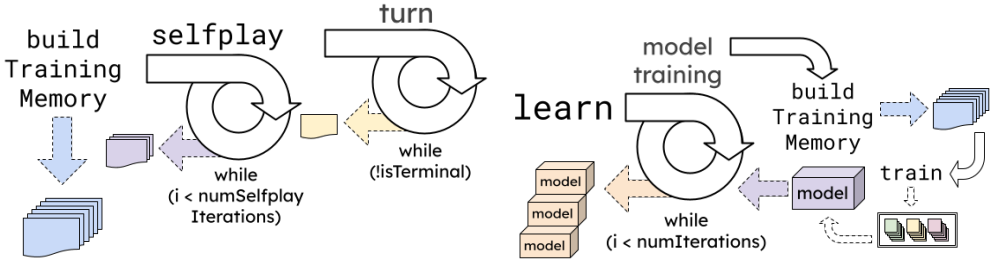
$$UCB = Exploitation + Exploration \tag{2a}$$

$$Exploitation = 1 - \frac{\frac{child.valueSum}{child.visitCount} + 1}{2} \tag{2b}$$

$$Exploration = expConst * child.priorProb * \frac{\sqrt{this.visitCount}}{child.visitCount + 1} \tag{2c}$$

3.6. Treinamento do modelo

Uma nova classe chamada `AlphaZero` foi criada para gerenciar o aprendizado de um modelo de ResNet. Para realizar o treinamento, deve-se construir uma memória de partidas jogadas. Isso é feito pelo método `buildTrainingMemory`, cujos processos são representados na Figura 5a. A função `selfplay` é chamada sucessivas vezes, conforme os parâmetros, para simular variadas partidas do jogo configurado. Em cada uma, sucedem-se turnos alternados entre dois jogadores até que se chegue a um estado terminal. É necessário fornecer um modelo já existente à classe `AlphaZero`, dado que é este que determina a ação escolhida a cada turno, a partir das probabilidades retornadas pelo seu método `predict`. Tal processo gera um conjunto de blocos de memória, formados por: o estado codificado para ResNet, as probabilidades de cada ação, e o resultado da jogada, como representado pelo tipo `TrainingMemoryBlock` na Figura 2.



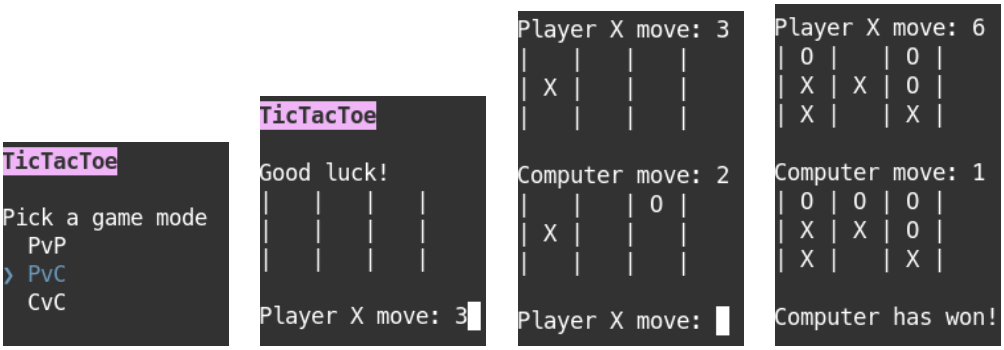
(a) Construção da memória. (b) Construção dos modelos.

Figura 5: Processo de treinamento dos modelos do APTS.

Por fim, o método `learn` gerencia essas etapas em um laço limitado pelo número de iterações, como mostrado na Figura 5b. Após memória ser construída, o método `train` a converte para o formato de entrada da ResNet, e encapsula o treinamento da rede. Ao fim de cada passo, o modelo e seus parâmetros são salvos no armazenamento.

4. Resultados

Com a implementação das estruturas do modelo ResNet descritas na Seção 3, além do sistema de treinamento do modelo, pôde-se implementar o ambiente de execução que rodam as rotinas de aprendizado de máquina. A interface do APTS, para diversas etapas é apresentada na Figura 6. A Figura 6a apresenta a escolha do modo de interação com o APTS, quais sejam: entre dois jogadores humanos (*Player versus Player* (PvP)), um humano contra o agente (*Player versus Computer* (PvC)) e dois agentes (*Computer versus Computer* (CvC)). Dessa forma, o usuário pode testar manualmente as regras de implementação, e verificar como o modelo treinado se comporta em uma partida. A Figura 6b apresenta o estado inicial e solicita uma jogada, a qual é realizada na Figura 6c. O resultado final da partida é apresentado na Figura 6d.



(a) Seleção de modo de jogo. (b) Estado inicial. (c) Andamento da partida. (d) Final da partida.

Figura 6: Interface do APTS para o Jogo da Velha contra o modelo treinado.

5. Considerações Finais, Limitações e Trabalhos Futuros

Neste trabalho foi apresentada uma implementação do método AlphaZero em JavaScript. Este é utilizado para o treinamento de um agente para jogos e é realizado pela modificação do algoritmo MCTS, ao incluir uma ResNet para avaliação e escolha de caminhamento na árvore de decisões, destacando-se por realizar uma série de partidas contra si mesmo, para o treinamento autônomo da rede. Essa implementação contribui para a criação do Auto Playtest System (APTS), que será utilizado para realizar um conjunto de testes de jogos em fase de desenvolvimento, os quais são importantes para a indústria, que deve fornecer experiências impactantes, com escolhas interessantes e bem balanceadas. Tal processo pode ser utilizado para diminuir a demanda por humanos em testes burocráticos e repetitivos, como os de estresse e mecanismos, além de descobrir estratégias dominantes.

No atual estado, o jogo da velha foi implementado por sua simplicidade. Trata-se de um jogo abstrato, de informação completa e exclusivo para dois jogadores. Espera-se utilizar a ferramenta em jogos com informação incompleta e multijogadores, nos quais a

árvore de decisão cresce significativamente. Isso exige estudos e observações subsequentes sobre o desempenho da implementação e abordagens para reaproveitar o treinamento entre as versões de desenvolvimento ou mesmo de camadas inteiras da rede. Este objetivo se consolida pelas interfaces mostradas na Figura 2: `Game` representa a lógica do jogo, e `State` define métodos de manipulação de estado. Dessa forma, torna-se possível mapear diferentes jogos para o *framework* desenvolvido. O processo limita-se a codificar as classes concretas correspondentes, as quais são responsáveis pela representação dos jogadores, pela obtenção de jogadas válidas a partir de um estado, pela verificação de vitória, e pela execução de dada ação sobre o espaço do jogo. Então, definir o jogo desejado e seus atributos nos parâmetros da aplicação. Logo, torna-se transparente para o motor de inteligência artificial e para os componentes de interface qual é o jogo carregado.

Outra limitação no estado atual é que o jogo necessita ser programado e a avaliação pelo autor é sobre dados brutos ou observação dos agentes. Até então, há duas formas de interação: pela interface em linha de comando, que permite jogar partidas contra o agente; e pelos *scripts* de teste e treinamento, que dão acesso a manipulações mais profundas, mas exigem conhecimento de programação. Ademais, a configuração dos parâmetros deve ser feita via código-fonte e análise das estruturas de dados. Portanto, ainda em aberto, deve-se disponibilizar uma interface mais amigável para o autor, que em se tratando de jogos físicos como de tabuleiro e cartas, não tem obrigação de ter contato com programação, como nos jogos digitais. Nessa linha, há a possibilidade de se utilizar uma linguagem específica de domínio ou algum padrão atual como a *Game Description Language* (GDL)[5] ou a ZRF Language[6]. Ademais, espera-se implementar ferramentas que facilitem a captura e observação das relações causais entre alterações nas regras e a avaliação das ações do AlphaZero ou a avaliação da rede com base nos estados intermediários da partida. Esse requisito torna-se mais crítico na medida em que jogos complexos são implementados e um banco de questões de design pode ser construído e avaliado em vários jogos diferentes.

Referências

BECKER, A. et al. What is Game Balancing? - An Examination of Concepts. *ParadigmPlus*, v. 1, n. 1, p. 22–41, abr. 2020. ISSN 2711-4627.

BOARDGAMEGEEK. *SPIEL'22 Preview*. 2022. Disponível em: ⟨https://boardgamegeek.com/geekpreview/55/spiel-22-preview?sort=hot⟩. Acesso em: 7 Setembro de 2023.

BOARDGAMEGEEK. 2023. Disponível em: ⟨https://boardgamegeek.com/⟩. Acesso em: 7 Setembro de 2023.

BRITANNICA, T. E. of E. *Go*. 2023. [Online; acesso em 3 setembro 2023]. Disponível em: ⟨https://www.britannica.com/topic/go-game⟩.

COULOM, R. Efficient selectivity and backup operators in monte-carlo tree search. In: SPRINGER. *International conference on computers and games*. [S.l.], 2006. p. 72–83.

FULLERTON, T. *Game design workshop : a playcentric approach to creating innovative games*. Fourth edition. [S.l.]: CRC Press, 2019. ISBN 9781315104300, 131510430X,

[5] http://logic.stanford.edu/ggp/notes/gdl.html
[6] https://www.zillionsofgames.com/language

9781351597685, 135159768X, 9781351597692, 1351597698, 9781351597708, 1351597701.

FöRSTER, R. *AlphaZero from Scratch*. 2023. [Online; acesso em 12 setembro 2023]. Disponível em: ⟨https://github.com/foersterrobert/AlphaZeroFromScratch⟩.

GUDMUNDSSON, S. F. et al. Human-like playtesting with deep learning. In: IEEE. *2018 IEEE Conference on Computational Intelligence and Games (CIG)*. [S.l.], 2018. p. 1–8.

HE, K. et al. Deep residual learning for image recognition. 2015.

KOCSIS, L. et al. Bandit based monte-carlo planning. In: SPRINGER. *European conference on machine learning*. [S.l.], 2006. p. 282–293.

MARCELO, A. et al. *Design de jogos: Fundamentos*. [S.l.: s.n.], 2009.

RANANDEH, V. et al. 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. p. 155–160.

ROMERO, B. et al. *Game Balance*. 1st edition. ed. Boca Raton: CRC Press, 2021. ISBN 978-1-4987-9957-7 978-1-03-203400-3.

SILVER, D. et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, v. 529, n. 7587, p. 484–489, jan. 2016. ISSN 0028-0836, 1476-4687.

SILVER, D. et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. Disponível em: ⟨https://doi.org/10.48550/arXiv.1712.01815⟩.

SILVER, D. et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, v. 362, n. 6419, p. 1140–1144, dez. 2018. ISSN 0036-8075, 1095-9203.

STAHLKE, S. et al. 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. p. 267–280.

ŚWIECHOWSKI, M. et al. Monte carlo tree search: a review of recent modifications and applications. *Artificial Intelligence Review*, Springer Science and Business Media LLC, v. 56, n. 3, p. 2497–2562, jul 2022. Disponível em: ⟨https://doi.org/10.1007%2Fs10462-022-10228-y⟩.

TENSORFLOW, G. *Get started with TensorFlow.js*. 2023? [Online; acesso em 4 setembro 2023]. Disponível em: ⟨https://www.tensorflow.org/js/tutorials⟩.

TEUBER, K. *Colonizadores de Catan*. 1995.

TRZEWICZEK, I. *I play-tested it 100 times*. [S.l.], 2017.

WALLNER, G. et al. Aggregated visualization of playtesting data. In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. [S.l.: s.n.], 2019. p. 1–12.

WOODS, S. *Eurogames: The design, culture and play of modern European board games*. [S.l.]: McFarland, 2012.

ZOOK, A. et al. Automatic playtesting for game parameter tuning via active learning. *arXiv preprint arXiv:1908.01417*, 2020.