DESENVOLVIMENTO E AVALIAÇÃO DE UMA METODOLOGIA PARA GERAÇÃO DE AGENTES GENÉRICOS PARA JOGOS DE TABULEIRO

MATEUS ANDRADE REZENDE

DESENVOLVIMENTO E AVALIAÇÃO DE UMA METODOLOGIA PARA GERAÇÃO DE AGENTES GENÉRICOS PARA JOGOS DE TABULEIRO

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais - Departamento de Ciência da Computação como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Luiz Chaimowicz

Belo Horizonte

Junho de 2017

Rezende, Mateus Andrade
R467d Desenvolvimento e Avaliação de uma Metodologia para Geração de Agentes Genéricos para Jogos de Tabuleiro / Mateus Andrade Rezende. — Belo Horizonte, 2017
xxii, 80 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais - Departamento de Ciência da Computação
Orientador: Luiz Chaimowicz

1. Computação - Teses. 2. General Game Playing. 3. Monte Carlo Tree Search. 4. Redes de Correlação em Cascata. 5. Método da entropia cruzada.
I. Orientador. II. Título.

CDU 519.6*82.9(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Desenvolvimento e avaliação de uma metodologia para geração de agentes genéricos para jogos de tabuleiro

MATEUS ANDRADE REZENDE

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. LUIZ CHAIMOWICZ - Orientador
Departamento de Ciência da Computação - UFMG

PROF. ADRIANO ALONSO VELOSO
Departamento de Ciência da Computação - UFMG

PROF. LEANDRO SORIANO MARCOLINO
Escola de Computação e Comunicações - Universidade de Lancaster

Belo Horizonte, 02 de junho de 2017.

Dedico este trabalho à minha família e amigos que sempre me dão apoio e sempre estão presentes.

Agradecimentos

Agradeço aos meus pais Maria José e Hélio pelo amor incondicional. Às minhas irmãs Carol e Bella pelo carinho e apoio. Aos meus amigos de longa data Acácio e Ândrik pelos incentivos e momentos de descontração. Ao professor Luiz Chaimowicz pela tranquilidade, compreensão, incentivos e conselhos. Aos meus amigos do trabalho, em especial Bruno e Johnny, pelo apoio e compreensão.

"Pensar é o trabalho mais difícil que existe. Talvez por isso tão poucos se dediquem a ele."

(Henry Ford)

Resumo

Um agente de *General Game Playing* (GGP) deve ser capaz de jogar efetivamente diferentes jogos talvez com algum processo inicial de aprendizagem. Os principais agentes de GGP se originaram da competição AAAI de GGP e em sua maioria se baseiam no algoritmo *Monte Carlo Tree Search* (MCTS). Geralmente esses agentes implementam melhorias no controle da busca associando valores de qualidade estatística a simples *features* aprendidas durante as partidas. Um desafio maior seria, dadas as regras de um jogo qualquer, como gerar um agente inteligente de forma completamente não supervisionada e que seja competitivo em comparação a agentes específicos para o jogo.

Neste trabalho, propomos um método denominado UCT-CCNN para o aprendizado *off-line* de função de valor para estados de jogos com dois jogadores, de soma zero, de informação perfeita, determinísticos, discretos e sequenciais. De forma mais prática, o método UCT-CCNN aceita qualquer jogo que possa ser modelado como uma árvore, como jogos de tabuleiro. Para jogos com essas características, o MCTS é um excelente algoritmo de propósito geral capaz de estimar utilidades para os estados de jogo independente de conhecimento específico de domínio, o que é essencial para tratar o problema GGP. Para uma boa estimativa da utilidade dos estados, e consequentemente decisões mais competitivas para o agente, o algoritmo MCTS precisa ser executado por mais tempo, o que não é possível em uma partida onde os turnos de cada jogador não podem demorar. No método UCT-CCNN inúmeras partidas são jogadas pelos agentes MCTS com política da árvore conhecida como *Upper Confidence Bounds for Tree* (UCT) em um processo *off-line* que gera uma base de dados de exemplos de estado-utilidade. A partir desses exemplos uma função de valor para os estados de jogo é aprendida com o uso de redes neurais construtivas denominadas *Cascade Correlation Neural Networks*, capazes de iterativamente construir uma arquitetura que se adapta ao problema que são submetidas, permitindo assim a característica GGP deste trabalho.

O método UCT-CCNN foi executado com os jogos Othello e Trilha e os agentes obtidos foram capazes de ganhar de agentes específicos do domínio. Com o UCT-CCNN

foi possível controlar a força do agente obtido através do controle do tempo de simulação MCTS para a geração dos exemplos usados no treino da rede neural, garantindo um método flexível capaz de gerar agentes inteligentes com diferentes níveis de dificuldade. É também mostrado que a função de valor obtida pelo método UCT-CCNN pode ser facilmente integrada em qualquer algoritmo, como o Minimax com Poda Alfa-Beta e o próprio MCTS, levando neste último a maiores taxas de vitória em comparação ao UCT padrão com o mesmo número de simulações.

Palavras-chave: *General Game Playing*, *Monte Carlo Tree Search*, *Upper Confidence Bounds for Trees*, *Cascade Correlation Neural Networks*, *Resilient Backpropagation*, *Cross-Entropy Method*.

Abstract

A General Game Playing (GGP) agent should be able to effectively play different games, sometimes with some initial learning process. The main GGP agents were originated from AAAI GGP competition and are mostly based on Monte Carlo Tree Search (MCTS) algorithm. Usually, those agents implement improvements in search-control by associating statistical quality values to simple features learned during the matches. A greater challenge would be, given the rules of any game, how to generate in a completely unsupervised way an intelligent agent that is competitive compared to specific agents for the game.

In this work, we propose a method called UCT-CCNN for the off-line learning of state value function of games with two players, zero-sum, perfect information, deterministic, discrete and sequential. In a practical way, the UCT-CCNN method accepts any game that can be modeled as a tree, such as board games. For a game with those characteristics, the MCTS is an excellent general purpose algorithm capable of estimating utilities for game states independently of specific domain knowledge, which is essential for dealing with the GGP problem. For a good estimate of the states' utility, and consequently more competitive decisions for the agent, the MCTS algorithm needs to run longer, which is not possible in a match where the turns of each player cannot take long. In the UCT-CCNN method, many matches are played by MCTS agents using the tree policy known as Upper Confidence Bounds for Tree (UCT) in an off-line process that generates a database of state-utility examples. From those examples, a value function for the game states is learned through the use of constructive neural networks known as Cascade Correlation Neural Networks, which are networks capable of iteratively building architectures that adapt itself to the applied problem, thus allowing the GGP characteristic of this work.

The UCT-CCNN method was executed with the games Othello and Nine Man's Morris and the obtained agents were capable of winning matches against specific domain agents. With the UCT-CCNN it was possible to control the strength of the obtained agent by controlling the MCTS simulation time for the generation of the

examples used in the neural network training, ensuring a flexible method capable of generating intelligent agents with different levels of difficulty. It is also shown that the value function obtained by the UCT-CCNN method can be easily integrated into any algorithm, like the Alpha-Beta Pruning Minimax and even the MCTS itself, leading the latter to higher winning rates when compared to the standard UCT with the same number of simulations.

Keywords: General Game Playing, Monte Carlo Tree Search, Upper Confidence Bounds for Trees, Cascade Correlation Neural Networks, Resilient Backpropagation, Cross-Entropy Method.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Seres humanos são capazes de aprender e jogar diferentes jogos, porém os agentes inteligentes normalmente são especializados em apenas um tipo de jogo, a fim de tirar um maior proveito do conhecimento específico do domínio relacionado. Um problema disso é que para cada jogo diferente deve ser desenvolvido um novo agente. Idealmente um agente inteligente seria capaz de jogar de maneira eficaz uma grande variedade de jogos, talvez com algum processo inicial de aprendizado. O desenvolvimento de agentes capazes de jogar mais do que um único jogo é uma área de estudo denominada *General Game Playing* (GGP) [Yannakakis & Togelius, 2015]. Essa área está relacionada com a *Artificial General Intelligence*, cujos esforços se concentram em propor e discutir conceitos e métodos focados em eventualmente alcançar uma inteligência artificial de domínio geral, algo parecido com a inteligência do ser humano [Goertzel & Pennachin, 2007]. O presente trabalho propõe um método capaz de gerar um agente inteligente genérico para jogos de tabuleiro, dadas as regras do jogo, sem nenhum conhecimento específico de domínio e de forma completamente não supervisionada. Neste capítulo é apresentado o contexto atual da área de GGP e seus desafios, as contribuições desta dissertação e como este documento é organizado.

1.1 Contexto Atual de GGP e Definição do Problema

O nome GGP advém da competição AAAI GGP na qual os agentes submetidos são testados em diferentes instâncias de jogos descritos em uma linguagem formal de definição de jogos denominada *Game Description Language* (GDL). A GDL caracteriza jogos finitos, discretos, determinísticos, multiagentes e de informação perfeita [Love et al., 2008],

características comuns a simples jogos de tabuleiro. Além da GDL outros *frameworks* surgiram para incentivar a pesquisa em GGP com uma gama maior de jogos. Um exemplo é o *Arcade Learning Environment* (ALE), baseado em emulação de jogos clássicos de Atari 2600 [Bellemare et al., 2013]. Outro exemplo é a *Video Game Description Language* (VGDL), que foca em simples jogos de duas dimensões [Ebner et al., 2013] [Schaul, 2013]. A VGDL é utilizada na competição *General Video Game AI* (GVG-AI) [Perez-Liebana et al., 2016]. Tais plataformas facilitam e promovem a pesquisa na área, ao prover um ambiente de simulação para as partidas e uma linguagem de descrição para facilitar a criação de várias instâncias de jogos diferentes, facilitando assim a validação de agentes de GGP.

Jogos podem ser muito diferentes entre si. Alguns necessitam de planejamento estratégico e a longo prazo, como em jogos de tabuleiro como GO e Xadrez. Outros exigem bons reflexos e *timing* como muitos jogos de Atari 2600. Há ainda aqueles que demandam boa memória e detecção de padrões como alguns jogos de *puzzle*. Certamente pode-se citar jogos que englobam todas essas características necessárias para um agente inteligente, como é o caso de jogos como StarCraft e Portal. O que esses jogos compartilham é a presença de um estado de jogo, que armazena todas as informações referentes ao ambiente e aos agentes, e um conjunto de ações que cada agente pode executar. Muitos implementam aleatoriedade, seja no resultado das ações dos agentes ou em eventos que alteram o estado do ambiente. Jogos simples como pequenos jogos de tabuleiro podem apresentar um espaço de busca tratável por técnicas tradicionais de Inteligência Artificial (IA), como algoritmos de busca. Infelizmente muitos jogos possuem um espaço de estados exponencial e técnicas como busca parcial e abstração do espaço de estados podem ser utilizadas, mas a forma com que se aplica tais técnicas depende da especificidade de cada jogo. Nesse contexto, a pergunta que queremos responder é: dadas as regras de um jogo qualquer, como gerar um agente inteligente capaz de jogá-lo, de forma completamente não supervisionada e que seja competitivo em comparação a agentes específicos para o jogo em questão?

1.2 Objetivos

Neste trabalho limitaremos nosso escopo a jogos genéricos de tabuleiro com dois jogadores, de soma zero, de informação perfeita, determinísticos, discretos e sequenciais. Tais características descrevem um jogo combinatório. Dentre tais jogos estão inclusos Go, Xadrez, Reversi, entre outros. São domínios excelentes para experimentos de IA por possuírem um ambiente controlado definido por simples regras, mas que tipica-

mente apresentam grande profundidade e estratégias complexas, podendo se tornar grandes desafios de pesquisa. A proposta deste trabalho no contexto de GGP é definir um método que dadas as regras de um jogo qualquer, nas características descritas anteriormente, seja capaz de gerar um agente inteligente para tal jogo. Isso equivale a dizer que para o agente aprender a jogar um jogo novo, o mesmo deve passar por um processo inicial de aprendizagem.

1.3 Contribuições

Neste trabalho, propomos um método denominado UCT-CCNN que recebe como entrada as regras de um jogo de tabuleiro e gera como saída uma função de valor para os estados de jogo. De forma mais prática, o método UCT-CCNN aceita qualquer jogo que possa ser modelado como uma árvore. Para jogos com essas características, o *Monte Carlo Tree Search* é um excelente algoritmo de propósito geral capaz de estimar utilidades para os estados de jogo independente de conhecimento específico de domínio, o que é essencial para tratar o problema GGP. Para uma boa estimativa da utilidade dos estados, e consequentemente decisões mais competitivas para o agente, o algoritmo MCTS precisa ser executado por mais tempo[1], o que não é possível em uma partida onde os turnos de cada jogador não podem demorar.

No método UCT-CCNN inúmeras partidas são jogadas pelos agentes MCTS com política da árvore conhecida como *Upper Confidence Bounds for Tree* (UCT) em um processo off-line que gera uma base de dados de exemplos de estado-utilidade. Os parâmetros do algoritmo MCTS são otimizados para o jogo em questão através do algoritmo conhecido como *Cross Entropy Method* (CEM). A base de dados de exemplos gerada passa por um processo de filtragem para eliminar utilidades que provavelmente não possuem a acurácia necessária para garantir boas decisões. A partir desses exemplos uma função de valor para os estados de jogo é aprendida com o uso de redes neurais construtivas denominadas *Cascade Correlation Neural Networks*, capazes de iterativamente construir uma arquitetura que se adapta ao problema que são submetidas, permitindo assim a característica GGP deste trabalho.

UCT-CCNN é um método de aprendizado de GGP, pois não se utiliza de conhecimento específico de domínio. Diferentemente dos agentes participantes da competição AAAI GGP, o UCT-CCNN necessita de uma fase anterior de processamento *off-line* antes do agente ser capaz de jogar um jogo novo. Dessa forma o agente gerado apre-

[1] O MCTS é um algoritmo *anytime*, ou seja, sua execução pode ser interrompida a qualquer momento e uma resposta válida será retornada, entretanto quanto maior for o tempo de execução melhor será a resposta retornada.

sentará decisões "fortes" desde o início das partidas, mas não será capaz de aprender durante as mesmas ou de jogar sem a execução da fase anterior de aprendizado. Essa é uma limitação do método UCT-CCNN que impede a participação de agentes na competição AAAI GGP. Outra diferença é que para gerar um agente para um jogo qualquer, o mesmo deve ter suas regras descritas através da implementação de interfaces de jogo. Os agentes participantes da competição AAAI GGP recebem as regras de um jogo através de sua descrição GDL.

Os jogos Othello e Trilha foram submetidos ao método UCT-CCNN e as funções de valor obtidas foram integradas ao algoritmo Minimax com Poda Alfa-Beta. Os agentes obtidos foram capazes de ganhar de agentes específicos do domínio, sendo que o agente Othello obteve melhores taxas de sucesso do que o agente Trilha, levando a diferentes conclusões para cada jogo. Com o UCT-CCNN foi possível controlar a força do agente obtido através do controle do tempo de simulação MCTS para a geração dos exemplos usados no treino da rede neural, garantindo um método flexível capaz de gerar agentes inteligentes com diferentes níveis de dificuldade. É também mostrado que a função de valor obtida pelo método UCT-CCNN pode ser facilmente integrada em qualquer algoritmo, como o Minimax com Poda Alfa-Beta e o próprio MCTS, levando neste último a maiores taxas de vitória em comparação ao UCT padrão com o mesmo número de simulações para o jogo Othello.

1.4 Organização

A dissertação é organizada da seguinte forma: no Capítulo 2 são introduzidos os conceitos básicos para o entendimento do método apresentado. O Capítulo 3 apresenta os principais trabalhos relacionados da área. No Capítulo 4 a metodologia proposta é descrita em detalhes. Já no Capítulo 5 a análise experimental e os resultados obtidos são mostrados. Por fim, o Capítulo 6 traz a conclusão do trabalho com uma discussão do método e dos resultados, além de sugestões para trabalhos futuros.

Capítulo 2

Background

Neste capítulo é apresentado uma visão geral das técnicas utilizadas neste trabalho. Dentre elas estão o algoritmo *Monte Carlo Tree Search* (MCTS) e sua variante *Upper Confidence Bounds for Tree* (UCT), *Cascade Correlation Neural Networks* (CCNN) e *Cross-Entropy Method* (CEM). O foco está em uma descrição mais geral das técnicas, os detalhes e adaptações relativos a este trabalho são apresentados no Capítulo 4.

2.1 Monte Carlo Tree Search

Nos últimos anos, o MCTS tem alcançado grandes avanços em muitos jogos específicos e jogos gerais, além de também ser utilizado em outras áreas de planejamento complexo no mundo real, como problemas de controle e otimização. Desta forma, o MCTS tem se tornado uma importante ferramenta para qualquer pesquisador de IA [Browne et al., 2012].

Dado um estado de jogo, o MCTS deve retornar a ação a ser executada naquele estado. O MCTS mantém uma árvore de estados do jogo que é construída de forma incremental e assimétrica. No início de sua execução, o MCTS recebe um estado de jogo e cria uma árvore contendo apenas um nó raiz representando o estado recebido. Depois desse processo de inicialização, o MCTS entra em um processo iterativo dividido em três etapas denominadas: Seleção, Simulação e *Backpropagation*.

A primeira fase do algoritmo é denominada fase de seleção, onde os nós da árvore construída até o momento são selecionados com base em uma política[1] denominada política da árvore. A política da árvore tenta balancear considerações de exploração justa (visitar áreas das quais poucas amostras foram feitas) e exploração focada (procurar em áreas que parecem mais promissoras). A partir da raiz da árvore, os nós

[1] política é uma função que especifica qual ação tomar em um dado estado de jogo, por exemplo.

são selecionados com base na política descrita anteriormente até atingir um nó para o qual ainda existam filhos que não foram expandidos. Antes de começar a segunda fase, dentre os filhos ainda não expandidos é escolhido um para expansão aleatoriamente de maneira uniforme. O filho expandido é adicionado à árvore.

A segunda fase do algoritmo é denominada fase de simulação e é executada a partir do nó expandido na fase anterior. Os movimentos, ou jogadas, são realizados durante a simulação de acordo com uma política denominada política padrão que, em sua forma mais simples, seleciona movimentos conforme uma distribuição uniforme. Apenas o valor real de utilidade[2] associada ao nó terminal é necessário no final de cada simulação.

A terceira e última fase do algoritmo se inicia ao atingir um nó terminal na fase de simulação. Essa fase final é denominada *backpropagation* e nela a árvore de busca é atualizada de acordo com os resultados obtidos na simulação. Isso envolve a atualização dos valores estatísticos dos nós no caminho de volta desde o nó expandido até a raiz da árvore. Esses valores estatísticos são o número de visitas ao nó e as *utilidades* de cada jogador acumuladas durante a fase de *backpropagation* no nó. Todas essas etapas do algoritmo MCTS estão contempladas na Figura 2.1. Na fase de seleção, os nós destacados foram selecionados pela política da árvore *Upper Confidence Bounds for Tree* (UCT) sendo o último um nó expandido. Na fase de simulação, os pequenos nós representam estados alcançados através da seleção de ações pela política padrão, até que um nó final de jogo seja alcançado. Na fase *backpropagation*, os nós destacados têm seus valores estatísticos atualizados.

Se uma simulação terminar com uma utilidade igual a zero não significa que o estado expandido é ruim. Estatisticamente falando, existe um intervalo de confiança para a utilidade esperada de um estado dado o número de vezes que esse estado foi selecionado na fase de seleção. Políticas otimistas exploram o limite superior desse intervalo de confiança a fim de encontrar a melhor ação a ser tomada. O algoritmo *Upper Confidence Bounds 1* (UCB1) determina uma política de escolha de ações que garante estar dentro de um fator constante do melhor limite do crescimento do valor denominado *regret* [Auer et al., 2002], que é a perda de utilidade por não tomar a melhor ação. Além disso, UCB1 é simples e eficiente. No algoritmo *Upper Confidence Bounds for Tree* (UCT), o UCB1 foi incorporado na política da árvore para selecionar uma dentre as possíveis ações a partir de um nó, com base em seus valores estatísticos. No algoritmo UCT, um nó filho (j) é selecionado na fase de seleção a fim de maximizar

[2]utilidade é um conceito abstrato associado às preferências dos agentes para a escolha da estratégia a ser adotada, que no caso do MCTS é um valor real que representa o resultado (ou *outcome*) associado ao estado sendo avaliado.

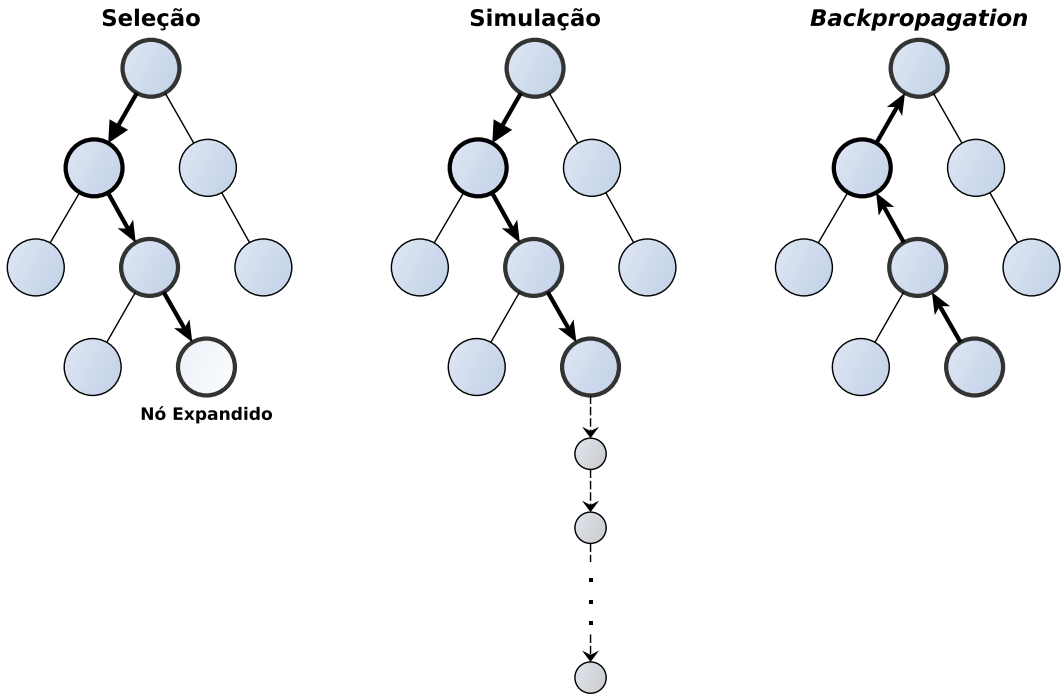


Figura 2.1: Etapas do algoritmo MCTS.

a utilidade UCT:

$$\text{UCT} = \overline{X_j} + 2C_p\sqrt{\frac{2\ln n}{n_j}},$$

onde $\overline{X_j}$ é a utilidade média observada para o nó j, n é o número de vezes que o nó pai do nó j foi visitado, n_j o número de vezes que o nó filho j foi visitado e $C_p > 0$ uma constante. É considerado que para $n_j = 0$ o valor UCT para o estado j seria infinito a fim de que estados nunca antes explorados tenham prioridade para serem expandidos. Estados com valores UCT iguais devem ser selecionados randomicamente. A constante C_p é denominada constante de exploração e pode ser ajustada para aumentar ou diminuir a prioridade na exploração em detrimento a priorizar estados com maior utilidade média observada. O valor ideal para a constante de exploração depende do problema sendo tratado e das melhorias implementadas nas políticas do MCTS. Cada nó armazena um valor $N(s)$ que corresponde ao número de vezes que o nó foi visitado e também armazena um valor $Q(s)$ que corresponde ao total de utilidade acumulada ao passar pelo estado s, portanto $\frac{Q(s)}{N(s)}$ é uma aproximação para a utilidade do estado s.

É conhecido que dados o tempo de execução e a quantidade de memória suficientes para o algoritmo MCTS com UCT, a árvore de estados converge para uma árvore

minimax ótima [Kocsis & Szepesvári, 2006] [Kocsis et al., 2006]. Dentre as principais características do MCTS estão a de que ele é independente de conhecimento específico de domínio, podendo ser utilizado em diferentes contextos com diferentes limites de tempo de execução (maior tempo de execução leva a melhores estimativas) e o crescimento da árvore de estados é assimétrico favorecendo regiões mais promissoras do espaço de estados. Outras políticas podem ser utilizadas no lugar da política da árvore UCT. Neste texto, MCTS-UCT é uma abreviação para o algoritmo MCTS que usa UCT como política da árvore. Sempre que a abreviação MCTS for mencionada sem especificar a política, ela não é relevante ao contexto.

No MCTS para se obter bons resultados muitas iterações do algoritmo devem ser executadas. Infelizmente, no contexto de jogos, a resposta da ação do agente em partidas reais não pode demorar muito. Em GGP, principalmente na competição AAAI GGP [Genesereth et al., 2005], o agente deve ser capaz de jogar qualquer jogo com um tempo restrito de preparação para a partida e de limite de tempo para efetuar uma jogada (*startclock* e *playclock*). Caso o servidor não receba uma resposta até o tempo limite, uma ação aleatória é selecionada para o agente. Devido a esse tempo restrito, os principais agentes vencedores das últimas competições se utilizam de mecanismos de aprendizagem durante as partidas oficiais do torneio, e a informação aprendida é usada para guiar a busca MCTS, a qual é paralelizada para conseguir efetuar o máximo de simulações possíveis no curto intervalo de tempo permitido.

A abordagem deste trabalho será um pouco diferente do contexto dos agentes envolvidos na competição GGP, pois o foco é gerar um agente que tenha decisões fortes logo no começo da partida, caso o oponente também seja um agente forte. O agente continua sendo de GGP pois o mesmo não se utiliza de conhecimento específico do domínio para aprender, apenas as regras do jogo. A diferença está em um processo prévio de aprendizagem off-line, antes de partidas oficiais do agente. O ponto positivo dessa abordagem é que o agente é forte desde o início da partida, e a sua força pode ser controlada pelo processo de aprendizagem, caso exista o interesse de gerar agentes com níveis variáveis de força (modo fácil ou difícil como existe em muitos jogos). O ponto negativo seria que o agente não aprende durante as partidas oficiais.

Sem estratégias pré-definidas e sem exemplos de partidas reais surge um problema de como criar um processo de aprendizagem off-line para um jogo qualquer. Uma possível solução para esse problema é utilizar o MCTS com sua capacidade de estimar valores de utilidades para estados de jogo sem conhecimento específico de domínio. Gerar e armazenar a árvore de busca completa através do MCTS é impraticável para problemas combinatoriais. Uma alternativa seria aprender através de uma Rede Neural Artificial (*Artificial Neural Network* - ANN) uma função de avaliação dos es-

tados baseada nos valores de utilidade esperada processados pelo MCTS. ANN é uma técnica que, assim como MCTS, recentemente tem apresentado excelentes resultados em *General Game Playing*.

2.2 *Cascade Correlation Neural Network*

Em uma Rede Neural Artificial (*Artificial Neural Network* - ANN) se aplica o princípio de aproximação de função por exemplos, ou seja, uma função é "aprendida" a partir de exemplos de entrada e saída. Uma ANN é composta por um conjunto de neurônios artificiais organizados em camadas. Cada neurônio possui conexões de entrada e de saída e todas as conexões em uma rede neural que se conectam em outro neurônio possuem um peso associado. As entradas de um neurônio passam por uma soma ponderada, onde cada entrada está associada com um peso, e o resultado é atribuído a uma função de ativação, que determinará o valor transmitido nas saídas.

Na Figura 2.2 é representado um neurônio artificial (k) que recebe cinco entradas, sendo uma delas um bias. A entrada do bias está associada com o peso b_k e as entradas x_i estão associadas com os pesos w_{ki} e φ representa uma função de ativação qualquer. A função de ativação mais comum é a *sigmoide* $\left(\varphi = \frac{1}{1+e^{-x}}\right)$, que é um caso especial da função logística. Sem o bias, o valor na saída sempre seria zero caso todos os valores de entrada fossem iguais a zero, independente dos valores dos pesos. Além disso o bias, que sempre está associado com o valor constante 1, garante maior flexibilidade para o modelo se ajustar aos dados, pois permite o deslocamento da curva gerada pela função de ativação para a esquerda ou para a direita.

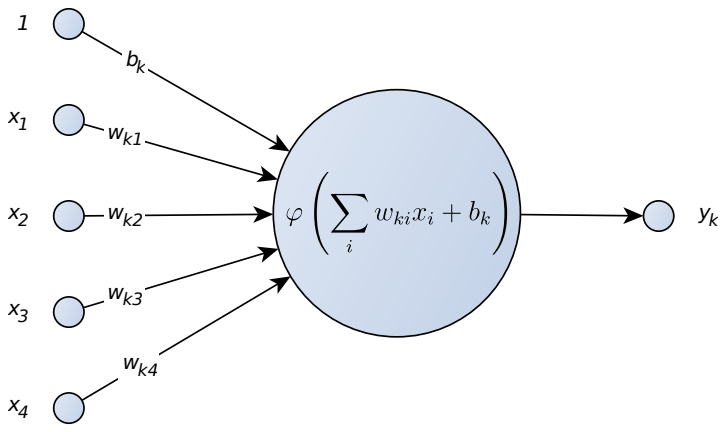


Figura 2.2: Representação de um neurônio artificial.

Antes de tratar um problema com uma ANN é necessário especificar uma grande quantidade de exemplos numéricos de entradas e saídas da função a ser aprendida. O aprendizado de uma ANN está relacionado ao ajuste dos pesos associados às conexões. A Figura 2.3 mostra uma arquitetura de rede neural com uma camada de entrada, uma camada escondida e uma camada de saída, todas com quatro neurônios mais as conexões relativas aos bias na camada escondida e na camada de saída.

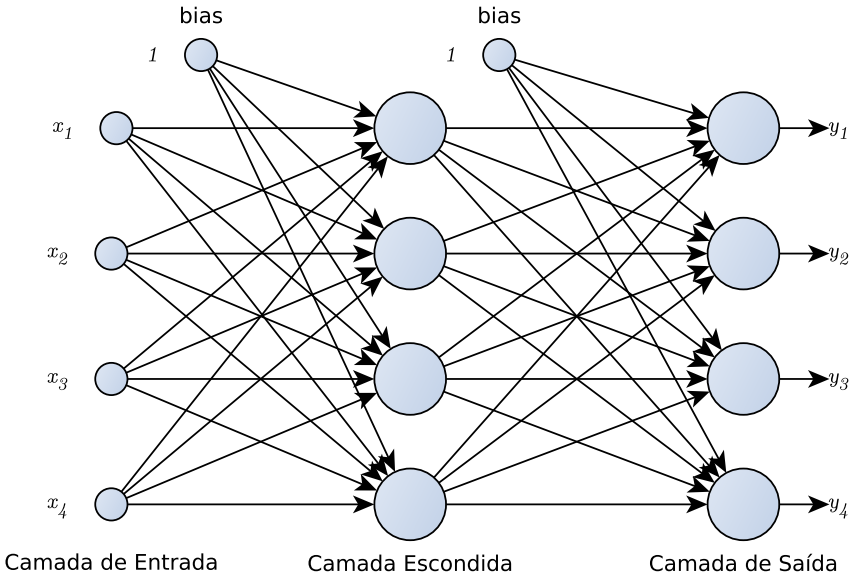


Figura 2.3: Arquitetura básica de uma ANN.

Criar uma rede neural cuja arquitetura seja capaz de generalizar diferentes problemas é uma tarefa desafiadora. Para conseguir uma convergência mais satisfatória da rede neural, de forma a generalizar corretamente entradas nunca antes vistas com o mínimo de erro, é necessário um conhecimento profundo do problema sendo tratado a fim de se efetuar os devidos ajustes nos parâmetros da rede neural. Dentre estes parâmetros, os mais críticos são relacionados à arquitetura da rede, que definem como os neurônios escondidos são organizados e como se conectam entre a camada de entrada e saída da rede, e os parâmetros do algoritmo de aprendizado, como é o caso da taxa de aprendizado para o algoritmo *Backpropagation*.

No algoritmo *Backpropagation*, um erro é calculado na saída da rede, conforme o exemplo fornecido na entrada e seu valor esperado, e então este erro é propagado de volta ajustando os pesos dos neurônios das camadas anteriores, conforme o gradiente associado aos pesos. Esse processo pode levar a convergência em um mínimo local, o que é evitado com a inicialização randômica dos pesos iniciais da rede. Além disso, múltiplos treinos são efetuados e a rede que convergiu melhor é escolhida. O parâ-

metro *step-size*, conhecido como taxa de aprendizado, determina o tamanho do ajuste efetuado nos pesos e tal parâmetro pode ser ajustado para controlar a convergência da rede. Os parâmetros da rede neural e do algoritmo de aprendizagem influenciam na generalização da função representada pela rede, e não ajustá-los corretamente pode levar a um *overfitting* ou sobreajuste, que significa um erro baixo para o conjunto de dados utilizado no treino, mas um erro alto para os dados do conjunto de teste e qualquer outro exemplo nunca antes visto pela rede (que não foram utilizados no treino da rede). Não existe uma maneira óbvia para ajustar esses parâmetros e não há garantias envolvidas.

Os problemas descritos anteriormente podem ser evitados ao se utilizar redes neurais construtivas, como é o caso das redes do tipo *Cascade Correlation Neural Network* (CCNN). A CCNN tem uma arquitetura especial que cresce para se adaptar ao problema, e também possui um processo de aprendizagem especial que reduz os custos computacionais e resolve muitos problemas do algoritmo *backpropagation* [Fahlman & Lebiere, 1990] [Balázs, 2009]. A CCNN começa com uma arquitetura mínima, apenas com as camadas de entrada e saída.

Os neurônios são adicionados na rede um por vez. Cada neurônio novo é inserido em uma nova camada escondida, ligada a todas as camadas anteriores. A Figura 2.4 mostra uma CCNN com dois neurônios escondidos adicionados. Os losangos pequenos e claros correspondem aos pesos associados às conexões de entrada nos neurônios de saída da rede e esses pesos são treináveis. Os quadrados pequenos e escuros correspondem aos pesos associados às conexões de entrada nos neurônios escondidos, já adicionados à rede, e esses pesos estão congelados (não são mais alterados depois de adicionados à rede). A seguir será mostrado como os pesos dos neurônios adicionados à rede são definidos.

O treinamento de uma rede CCNN consiste em um processo iterativo no qual um neurônio é treinado e adicionado à rede a cada iteração. Antes de ser adicionado à rede, um neurônio é treinado e, após adicionado, têm seus pesos das conexões de entrada congelados. Para treinar um novo neurônio, são calculados através de gradiente ascendente os pesos de entrada do neurônio candidato que maximizam a covariância C entre a saída do neurônio candidato e a saída da rede formada até o momento. Todos os pesos de entrada na camada de saída, incluindo os pesos de saída do neurônio candidato a ser adicionado, são treinados depois que um neurônio candidato é adicionado à rede. A covariância C é definida conforme a fórmula a seguir:

$$C = \sum_{o \in O} \left| \sum_{s \in S} (y_s - \overline{y})(e_{o,s} - \overline{e_o}) \right|,$$

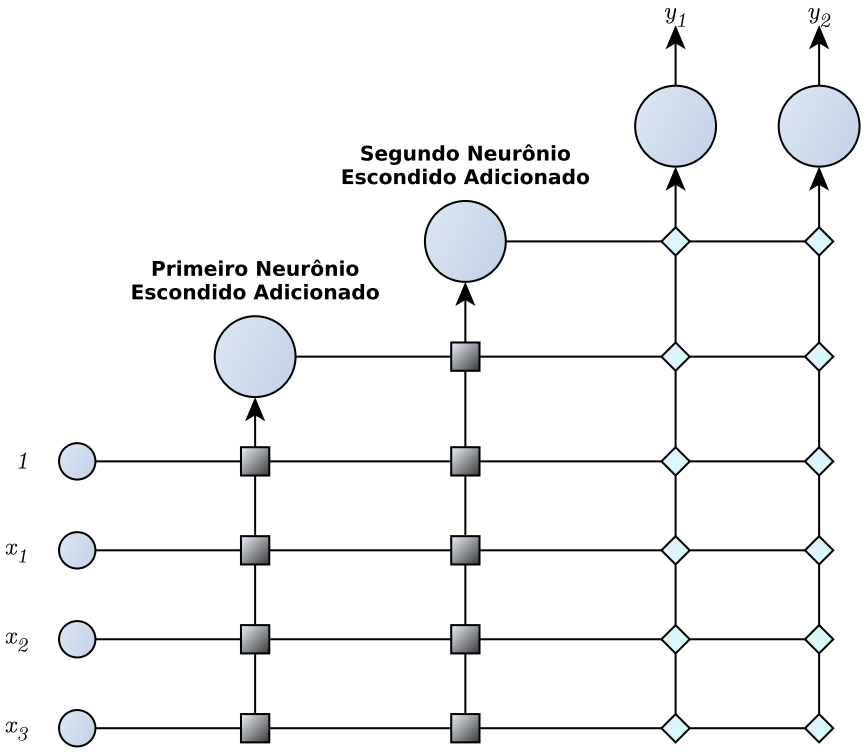


Figura 2.4: Arquitetura básica de uma CCNN.

onde O é o conjunto dos neurônios de saída da rede, S é o conjunto dos exemplos de treino, y_s é a saída do neurônio candidato para o exemplo s, $e_{o,s}$ é o erro na saída do neurônio de saída o para o exemplo s, $\overline{y}$ e $\overline{e_o}$ são as médias dos valores y_s e $e_{o,s}$ sobre todos os exemplos em S, respectivamente.

Uma outra abordagem para o treino de neurônios candidatos foi implementada por Fahlman, o criador da arquitetura CCNN, conhecida como Cascade 2 [Nissen, 2007]. Nessa abordagem o neurônio candidato é treinado para minimizar através de gradiente descendente a diferença *C2* entre o erro dos neurônios de saída e as entradas dos neurônios de saída recebidas do candidato sendo treinado. A diferença *C2* é representada a seguir pela fórmula:

$$C2 = \sum_{o \in O} \sum_{s \in S} (e_{o,s} - y_s \cdot w_{y,o})^2,$$

onde $e_{o,s}$ é o erro de saída do neurônio de saída o para o exemplo s, y_s é a saída do neurônio candidato para o exemplo s e $w_{y,o}$ é o peso do neurônio candidato y para o neurônio de saída o. Tanto os pesos de entrada do neurônio candidato quanto os pesos de saída do neurônio candidato são atualizados na minimização da diferença *C2*.

Com a minimização da diferença *C2* a ativação do neurônio candidato ponderada terá um valor próximo do erro da rede, portanto os pesos de saída do neurônio candidato devem ter o sinal invertido para contribuir com a minimização do erro. A abordagem Cascade-Correlation que usa a maximização de C é melhor para problemas de classificação, enquanto a abordagem Cascade 2 que usa a minimização de *C2* é melhor para problemas de regressão [Nissen, 2007].

Vários neurônios candidatos podem ser treinados independentemente com funções de ativação diferentes e diferentes inicializações aleatórias de pesos. O neurônio com maior covariância C ou menor diferença *C2* é selecionado, descartando-se os demais. Ao adicionar um neurônio na rede os pesos de suas conexões de entrada são congelados e, no caso da maximização da covariância C, todos os pesos de todos os neurônios conectados à camada de saída são treinados novamente. O ajuste de pesos em qualquer etapa é realizado através de algum algoritmo de aprendizado como *Backprop*, *Quickprop* [Fahlman, 1989] ou *Rprop* [Riedmiller & Braun, 1993]. O nome do algoritmo remete à correlação porque, no trabalho original, em uma primeira tentativa a correlação foi utilizada no treino dos neurônios candidatos, mas depois ficou decidido que a covariância seria a melhor opção, pois a mesma funcionou melhor em inúmeras situações [Fahlman & Lebiere, 1990].

Para este trabalho o algoritmo de aprendizado escolhido para treinar os pesos da CCNN foi o *iRprop* [Igel & Hüsken, 2000], que é uma variante melhorada do algoritmo original *Rprop*. No algoritmo *Rprop* a atualização de cada peso é baseada no sinal da derivada parcial do erro em relação ao peso, tornando o parâmetro *step-size* independente do valor absoluto da derivada parcial. A grosso modo, sendo w_{ij} o peso da conexão ligando o neurônio j ao neurônio i e E uma media de erro diferenciável em relação aos pesos, se a derivada parcial $\partial E/\partial w_{ij}$ possui o mesmo sinal para os passos consecutivos de atualização de peso, o *step-size* é incrementado, se o sinal dessa derivada mudar o *step-size* é decrementado. Cada peso possui um *step-size* associado.

A constante de salto do ajuste do *step-size*, o valor inicial dos *step-sizes* e os limites máximo e mínimos para os *step-sizes* são parâmetros do algoritmo. Como o valor *step-size* é ajustado dinamicamente pelo algoritmo, essa técnica de aprendizado dispensa o parâmetro conhecido como taxa de aprendizado, tornando o algoritmo *Rprop* ideal para ser utilizado em arquiteturas construtivas como a CCNN. Na variante *iRprop* atualizações anteriores de pesos são revertidas em caso de mudança de sinal da derivada parcial apenas se o erro geral da rede tiver aumentado. Além disso quando a derivada parcial muda de sinal, o valor da mesma é ignorado na próxima iteração, ou seja, não será verificada a mudança ou permanência de sinal na próxima iteração, o que levará apenas a uma atualização dos pesos conforme o *step-size* atualizado no passo atual.

2.3 *Cross-Entropy Method*

Cross-Entropy Method (CEM) foi originalmente desenvolvido como um método de simulação para estimar probabilidades de eventos raros e posteriormente também passou a ser utilizado como um método de otimização estocástica [Rubinstein & Kroese, 2013]. Neste trabalho o CEM é utilizado para otimizar a constante de exploração da política da árvore UCT, e assim garantir melhores estimativas para as utilidades dos estados no MCTS [Chaslot et al., 2008].

O CEM envolve um procedimento iterativo dividido em duas etapas. Na primeira etapa é realizada uma amostragem randômica de exemplos através de alguma distribuição de probabilidade parametrizada. Na segunda etapa os parâmetros da distribuição utilizada na geração dos exemplos são atualizados baseado nos dados produzidos, a fim de gerar melhores exemplos na próxima iteração do algoritmo. Uma importante característica do CEM é a sua convergência assintótica, onde sob leves condições de regularidade o processo é finalizado com probabilidade 1 em um número finito de iterações [de Mello & Rubinstein, 2002].

A distribuição g utilizada é escolhida de uma família G de distribuições que melhor se adapta ao problema. Alguns exemplos de distribuição são Gaussiana, Binomial, Bernoulli, etc. A partir da distribuição g, N exemplos são extraídos independentemente em uma amostra X, onde $X = \{x_i | 1 \leq i \leq N\}$. Seja $f(x_i)$ uma função que retorna a utilidade de um exemplo x_i qualquer. A ideia é encontrar os valores dos parâmetros ideais da distribuição selecionada que levará a exemplos de alta utilidade. Considerando um valor de utilidade γ, define-se um conjunto "elite" L_γ, onde $L_\gamma = \{x_i | f(x_i) \geq \gamma, 1 \leq i \leq N\}$. O valor da utilidade mínima para definir o conjunto elite determinará os exemplos utilizados no processo de estimativa dos parâmetros de uma melhor distribuição, sendo assim o valor de tal utilidade não pode ser muito alto porque senão um número pequeno de exemplos seria gerado prejudicando o aprendizado. Normalmente o valor de γ é definido de forma a inserir uma quantidade mínima de exemplos no conjunto elite, proporcional ao número de exemplos gerados. Uma forma de fazer isso é definir $\gamma = f(x_{\rho \times N})$, onde ρ é a taxa de seleção e os exemplos estão ordenados em ordem decrescente de suas utilidades. Na prática ρ é escolhido do intervalo $[0{,}01; 0{,}1]$ e determinará o conjunto elite constituído dos $\rho \cdot N$ melhores exemplos.

A ideia do CEM é identificar uma nova distribuição de G que melhor se aproxima de uma distribuição empírica sobre o conjunto elite gerado. Isso é feito identificando uma distribuição g que minimize a distância *cross-entropy* D_{CE} [Kullback, 1959] em relação à distribuição uniforme h sobre os exemplos do conjunto elite, conforme fórmula

definida a seguir:

$$D_{CE}(g||h) = \int g(x) log \frac{g(x)}{h(x)} dx.$$

Para uma distribuição Gaussiana $g\prime$ qualquer, os parâmetros que caracterizam uma distribuição com a distância *cross-entropy* mínima podem ser facilmente calculados a partir das estatísticas do conjunto elite, conforme mostrado nas fórmulas a seguir:

$$\mu' = \frac{\sum\limits_{x \in L_\gamma} x}{\rho \cdot N}$$

e

$$\sigma^{2\prime} = \frac{\sum\limits_{x \in L_\gamma} (x - \mu')^2}{\rho \cdot N},$$

nas quais os parâmetros de $g\prime$ com distância *cross-entropy* mínima são obtidos a partir das médias e variância do conjunto elite.

O CEM começa com uma distribuição para cada parâmetro a ser otimizado no problema alvo. Cada distribuição no CEM começa com os parâmetros iniciais previamente definidos. Também são previamente definidos o número de amostras geradas em cada iteração, a taxa ρ de seleção do conjunto elite, o parâmetro *step-size* que controla a taxa de convergência do processo e o número máximo de iterações. Em cada iteração, inicialmente são gerados exemplos com os parâmetros atuais de cada distribuição, e dos exemplos gerados para cada distribuição é extraído o conjunto elite de acordo com o parâmetro ρ. São então obtidos os novos parâmetros para cada distribuição minimizando a distância *cross-entropy* em relação à distribuição uniforme sobre os exemplos do conjunto elite. Por fim é realizada uma atualização dos parâmetros de cada distribuição em direção aos valores dos parâmetros obtidos na minimização da distância *cross-entropy*. Essa atualização é feita considerando um *step-size* previamente informado. Na iteração seguinte novos exemplos serão gerados com os os parâmetros atualizados das distribuições e esse processo irá se repetir até a convergência ou até que o número máximo de iterações seja atingido.

No término da execução do CEM, o valor a ser utilizado para cada parâmetro do problema alvo é o valor final da média da distribuição associada ao parâmetro, visto que as atualizações dos parâmetros das distribuições durante as iterações do CEM levaram cada uma delas a gerar exemplos em torno de regiões mais promissoras do espaço de busca.

Capítulo 3

Trabalhos Relacionados

Na competição GGP, os recentes vencedores são todos baseados em MCTS-UCT [Browne et al., 2012], como é o caso do CadiaPlayer, que venceu a competição nos anos de 2007, 2008 e 2012 [Finnsson, 2012]. O vencedor mais recente (2016) explora um método, também aplicado em MCTS, denominado *Upper Confidence Bounds* (UCB) que é relacionado ao problema conhecido como *Multi-armed Bandit*, em conjunto com o algoritmo de satisfação de restrições *Maintaining Arc Consistency* (MAC), caracterizando assim uma nova técnica de *Stochastic Constraing Programming* denominada MAC-UCB [Koriche et al., 2016]. Esse último agente, conhecido como *WoodStock*, suporta jogos não determinísticos com a nova versão da GDL, denominada GDLII, que dá suporte a jogos multiagentes, com incerteza e informação incompleta, apesar do agente WoodStock focar em jogos estocásticos completamente observáveis.

Em relação ao *Arcade Learning Environment*, técnicas envolvendo redes neurais têm alcançado bons resultados. Uma delas é a *Neuroevolution*, onde os pesos e a arquitetura de uma rede neural são evoluídos através de algoritmos genéticos [Hausknecht et al., 2014]. Em uma outra abordagem denominada DQN, dentre as técnicas utilizadas estão o *Double Q-learning*, um algoritmo voltado para aprendizado por reforço, e Redes Neurais Convolucionais Profundas, que são particularmente úteis para extração de características de uma matriz de *pixels*, no caso, um *frame* do jogo Atari relacionado. Essa abordagem se tornou o estado da arte atualmente envolvendo o ambiente de jogos Atari ALE [van Hasselt et al., 2016]. Um outro trabalho recente com resultados competitivos em relação ao DQN usa engenharia de características dos jogos ALE para mostrar que é possível alcançar bons resultados se as características certas dos jogos forem consideradas, o que ficou conhecido como Aprendizado por Reforço Raso, colocando um questionamento na necessidade de técnicas de alta demanda computacional como o Aprendizado Profundo [Liang et al., 2016]. Algoritmos

de Aprendizado Profundo como as Redes Convolucionais são eficazes para aprender características relevantes dos dados de entrada de forma automática, porém exigem uma grande quantidade de dados e um grande poder computacional.

O que pode atrair um maior interesse para os agentes de GGP é se os mesmos forem competitivos em relação a agentes específicos. Os agentes específicos são favorecidos por utilizar heurísticas para reduzir o número de estados visitados durante uma busca, além de poder aplicar abstrações e se utilizar de *features* específicas de cada jogo, e assim focar em regiões mais promissoras do espaço de busca. Mesmo com o uso de conhecimento específico de domínio, muitos jogos ainda são um grande desafio para a IA, como são o caso de Go e do jogo Starcraft. A melhor IA criada para Go, denominada AlphaGo, utiliza técnicas de propósito geral como Redes Neurais de Aprendizado Profundo e MCTS, e foi o primeiro algoritmo capaz de ganhar de um jogador experiente de Go para o tamanho de tabuleiro padrão de 19x19. Isso mostra o poder do algoritmo MCTS caso a busca possa ser direcionada com o uso de técnicas auxiliares, como as Redes Neurais aplicadas ao MCTS no AlphaGo. Apesar da utilização de técnicas de propósito geral, o AlphaGo recorre ao uso de conhecimento específico de domínio junto a essas técnicas a fim de obter um agente mais forte. As redes neurais são treinadas a partir de exemplos de jogadas de partidas reais entre jogadores profissionais, que usam conhecimento específico de domínio para determinar as melhores jogadas. Algumas *features* do jogo Go também são definidas manualmente a fim de garantir a avaliação das mesmas e melhorar o aprendizado da rede. Além disso os dados utilizados no treino são de um tamanho específico de tabuleiro, o que pode prejudicar o agente obtido para tabuleiros com tamanhos diferentes.

Uma das restrições no uso de MCTS para GGP é o tempo de simulação, que é restrito ao tempo máximo do turno para jogos sequenciais ou o tempo máximo de processamento para um agente tomar uma decisão em jogos simultâneos, o que nesse último caso é prejudicial ao agente uma demora na escolha da ação. Outro problema é que em grande parte do tempo de simulação do MCTS porções não interessantes da árvore de busca são exploradas, sendo necessário maior tempo de simulação ou técnicas para direcionar a busca MCTS.

Técnicas modernas para jogos com espaços de busca com ampla exploração impraticável, como é o caso do Go, são utilizadas para remediar esse problema. Dentre tais técnicas estão *First-Play Urgency* [Gelly & Wang, 2006], técnicas *All Moves As First* (AMAF) como a técnica *Rapid Action Value Estimation* (RAVE) [Gelly & Silver, 2007], melhorias de seleção como *Progressive Bias* [Chaslot et al., 2007], e técnicas de *Pruning*. Essas e outras técnicas estão listadas em Browne et al. [2012], e a maioria delas foram utilizadas em agentes vencedores da competição de GGP. Muitas dessas

técnicas não garantem melhores taxas de vitórias para quaisquer jogos, apesar de melhorar a performance do agente em muitos deles [Finnsson, 2012]. Além disso, algumas dessas técnicas demandam conhecimento específico do domínio, o que não é interessante para o foco em GGP.

Os principais trabalhos relacionados dos quais o método apresentado nesta dissertação tirou inspirações são o CadiaPlayer [Finnsson, 2012] e a sua estratégia adotada para a generalização do algoritmo MCTS para GGP e de melhorias não específicas de domínio; e o AlphaGo [Silver et al., 2016] e a sua estratégia de integração do MCTS com redes neurais. Nas próximas seções deste capítulo, uma visão geral desses algoritmos é apresentada.

3.1 AlphaGo

Go é considerado um grande desafio para a Inteligência Artificial devido ao seu grande espaço de estados e pela dificuldade de criar avaliações heurísticas para um estado de jogo. Avaliações heurísticas de estados de um jogo qualquer são utilizadas para reduzir o espaço de busca em algoritmos como o Minimax com altura limitada, que interrompe a busca em profundidade quando a altura máxima é alcançada e usa a avaliação heurística para avaliar o estado, o que levou ao sucesso agentes inteligentes para os jogos Xadrez, Damas e Reversi. Outra forma de reduzir o espaço de busca seria efetuar uma poda, ou seja, limitar a busca em largura ao invés da profundidade. MCTS limita a largura de busca através de uma política de seleção de ações, o que levou ao sucesso agentes inteligentes para os jogos Gamão e Palavras Cruzadas. Os melhores agentes Go antes do AlphaGo utilizam MCTS com melhorias em suas políticas, o que levou a um nível amador forte do agente. Tais melhorias são "rasas" no sentido de que se utilizam de combinação linear de características simples extraídas do estado de jogo.

AlphaGo [Silver et al., 2016] utiliza, em sua implementação do MCTS, políticas que se utilizam de redes neurais de aprendizado profundo. São utilizadas redes que geram na saída uma distribuição de probabilidades sobre as ações disponíveis a partir de um estado qualquer e redes que retornam o provável resultado do jogo a partir de um estado qualquer.

Primeiramente uma rede neural convolucional foi treinada por aprendizado supervisionado com exemplos (um exemplo equivale a um estado como entrada e uma distribuição de probabilidades sobre as ações possíveis como saída) extraídos da base de dados *KGS Go Server* de 30 milhões de posições de jogo, sendo que os exemplos eram

selecionados randomicamente durante o treino. A rede alcançou uma taxa de acerto[1] de 57%. Tal rede representa a política p_σ. A rede da política p_σ alterna entre camadas convolucionais e retificadores não-lineares. Uma camada final de *softmax*, representada pela função $\varphi(z^{(i)}) = \frac{e^{z^{(i)}}}{\sum_{j=0}^{k} e^{z^{(j)}}}$, onde $z^{(i)}$ representa o valor recebido pela softmax no neurônio de saída i e k representa a quantidade de neurônios de saída. Dessa forma a saída representa uma distribuição de probabilidades sobre todas as ações legais a partir do estado recebido como entrada. Uma política mais simples e mais rápida, denominada p_π, também foi treinada usando um classificador linear softmax a partir de pequenos padrões de características de um estado de jogo Go, a qual atingiu uma acurácia de 24.2%, mas executa muito mais rapidamente.

Em um segundo estágio, uma rede neural convolucional de arquitetura idêntica à rede da política p_σ é melhorada através de aprendizado por reforço de gradiente de política, gerando uma política mais forte p_ρ. Os pesos inicialmente são iguais aos da rede da política p_σ. As partidas são realizadas entre a rede da política atual e iterações anteriores de redes selecionadas randomicamente a fim de evitar overfitting. A recompensa é zero para todos os passos de tempo não terminal. Quando o fim do jogo é alcançado a recompensa é $+1$ para vitória e -1 para derrota. A política p_ρ ganhou 80% dos jogos contra a política p_σ e ganhou 85% dos jogos contra o algoritmo de código aberto mais forte denominado Pachi, que executa 100.000 simulações por jogada.

Por fim, uma função de valor denominada v_θ é treinada para, diferentemente das redes treinadas anteriormente, prever o resultado do jogo a partir de um estado qualquer. Exemplos de estados com seus respectivos resultados associados são extraídos de partidas distintas jogadas entre redes da política p_ρ, para evitar o *overfitting* ocasionado por usar estados sequenciais de uma mesma partida. A função de valor v_θ é treinada por uma rede neural de arquitetura semelhante à rede da política p_σ mas com apenas uma saída ao invés de uma distribuição de probabilidades. A rede da função de valor v_θ se aproximou da acurácia de MCTS em conjunto com a política p_ρ, utilizada na seleção de ações durante as simulações, mas com uma redução de 15.000 vezes no tempo de computação. Na figura 3.1 está ilustrado o pipeline de aprendizado de máquina descrito anteriormente.

AlphaGo combina as redes das políticas e da função de valor em um algoritmo MCTS para selecionar ações através de pesquisa com *lookahead*. Na árvore de busca, cada aresta caracterizada pelo par de estado e ação (s,a) armazena um valor de ação $Q(s,a)$, um contador de visitas $N(s,a)$ e uma probabilidade prévia $P(s,a)$. A cada passo

[1] porcentagem das distribuições de probabilidades sobre as ações de jogadas profissionais previstas corretamente de um conjunto de testes cujos exemplos não foram utilizados no treino

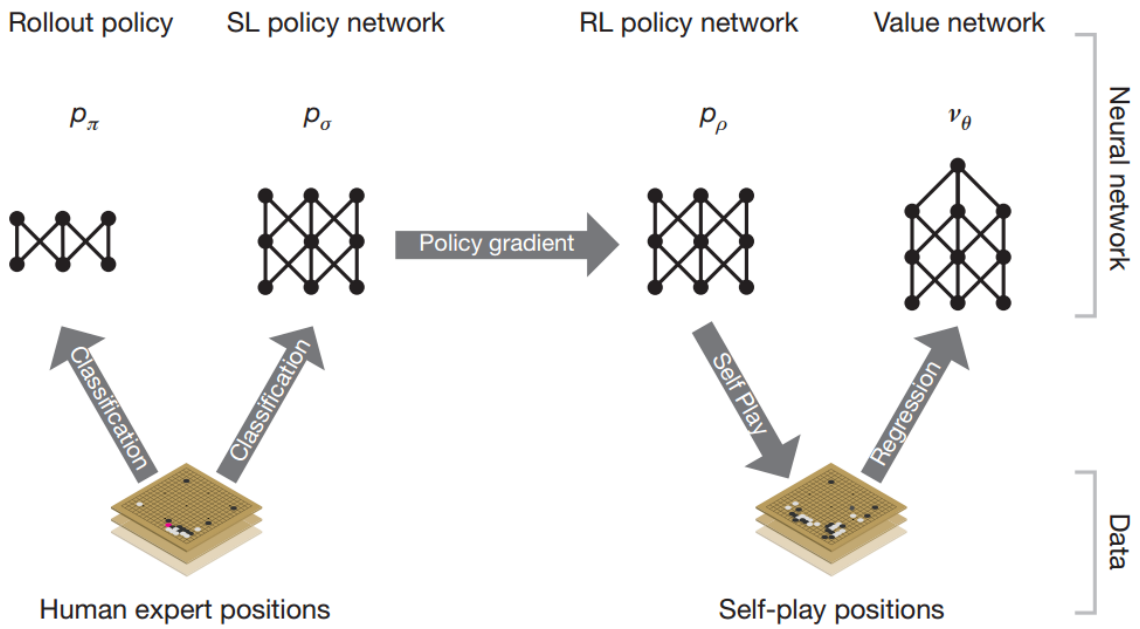


Figura 3.1: Pipeline de Treinamento e Arquitetura de Rede Neural [Silver et al., 2016].

t na árvore MCTS, uma ação a_t é selecionada de um estado s_t a fim de maximizar o valor de ação mais um bônus:

$$u(s_t,a) \propto \frac{P(s,a)}{1+N(s,a)},$$

que é proporcional à probabilidade prévia mas decai conforme ocorrem visitas repetidas, a fim de incentivar a exploração.

Quando um nó folha s_L é atingido em um passo L, o mesmo é processado pela rede da política p_σ e a distribuição de probabilidade é armazenada em forma de probabilidades prévias para cada ação legal a, ficando $P(s,a) = p_\sigma(a|s)$. O nó folha é avaliado pela rede da função de valor $\mathrm{v}_\theta(s_L)$, posteriormente um resultado z_L é calculado executando ações até o final da partida usando a política rápida p_π. Esses valores são combinados usando um parâmetro de mistura λ em uma função de avaliação $V(s_L)$

$$V(s_L) = (1-\lambda)v_\theta(s_L) + \lambda z_L.$$

No final da simulação, os valores de ação e contadores de visitas de todas as arestas percorridas são atualizados. Quando o algoritmo termina, a ação mais visitada a partir da raiz é escolhida.

Um torneio foi organizado entre o AlphaGo e suas variantes contra os mais fortes programas comerciais como *Crazy Stone* e *Zen*, e os programas mais fortes open-source

como *Pachi* e *Fuego*. Todas essas implementações utilizam algoritmos MCTS de alta performance. Também foi inserido nos torneios o algoritmo *GnuGo* que usa métodos estado da arte que precedem MCTS. O AlphaGo de uma única máquina foi capaz de ganhar 99.8% das partidas realizadas contra outros programas Go. A versão distribuída de AlphaGo ganhou de 5 a 0 contra Fan Hui, um jogador profissional que ganhou o Campeonato Europeu de Go nos anos de 2013, 2014 e 2015, sendo a primeira vez que um programa de computador ganha de um jogador profissional.

Vale notar que a rede da política p_σ apresentou melhor performance na política da árvore do que a rede da política mais forte p_ρ, presumivelmente porque humanos avaliam de um diverso conjunto de ações promissoras, enquanto a política p_ρ extraída de aprendizado por reforço foca em uma única melhor ação. Lembrando que as políticas p_σ e p_ρ retornam uma distribuição de probabilidades sobre as ações possíveis dado um estado qualquer.

Em contrapartida, a função de valor v_θ, que retorna o resultado provável do jogo dado um estado qualquer, extraída da política p_ρ teve melhor performance na política padrão do que a função de valor derivada da política p_σ.

Enfim, o que o AlphaGo mostrou é que o MCTS, um algoritmo de propósito geral, pode ser melhorado também com técnicas de propósito geral ao ponto de resolver problemas específicos considerados difíceis melhor do que qualquer outra técnica específica. Vale notar que apesar das técnicas aplicadas serem de propósito geral, parâmetros foram melhorados para o jogo específico Go de tamanho de tabuleiro 19x19 e a rede neural foi inicialmente treinada com aprendizado supervisionado a partir de exemplos de jogadas reais, além das arquiteturas das redes também serem montadas e melhoradas especificamente para o jogo Go. Idealmente um método de aprendizado para GGP não deveria depender de exemplos de jogadas reais, nos quais estaria embutido o conhecimento específico de domínio do jogador que executou as jogadas.

3.2 CadiaPlayer

Cadiaplayer [Finnsson, 2012] é um importante agente de GGP que foi o único competidor a vencer três vezes a competição de *General Game Playing*, nos anos de 2007, 2008 e 2012. Seu sucesso está relacionado com o uso de MCTS-UCT e uma série de melhorias que guiam a busca com base no conhecimento adquirido do jogo em tempo real. Tal conhecimento extraído durante as partidas pode ser usado para avaliar estados de jogo e para guiar a busca nas simulações do MCTS. Todas as melhorias apresentadas levaram a um aumento nas taxas de vitórias na maioria dos jogos testados. Algumas delas se

saíam melhor em determinados tipos de jogos. Alguns poucos jogos não apresentavam melhora e até mesmo algumas extensões atrapalhavam um pouco a performance do agente. Para resultados mais detalhados consulte Finnsson [2012].

Posteriormente será apresentada uma visão geral das melhorias implementadas no agente Cadiaplayer. Um primeiro conjunto de melhorias utilizadas no MCTS estão relacionadas com o controle de simulação. São um total de cinco mecanismos para guiar a execução da simulação no MCTS. Outras duas extensões existentes para o MCTS foram generalizadas para o contexto de GGP e também estão apresentadas.

3.2.1 *Move-Average Sampling Technique* (MAST)

Com esta técnica se aprende automaticamente conhecimento do domínio para melhorar a estratégia de seleção de ação durante a fase de simulação. Dessa forma o agente foi capaz de ganhar a competição de 2008. A informação de controle é aprendida durante a fase de *backpropagation*. Posteriormente, esse conhecimento adquirido é utilizado nas futuras simulações para enviesar a seleção randômica de ações na direção de ações mais promissoras.

Diferentemente do MCTS padrão, na melhoria MAST todos os estados de uma simulação são visitados na fase de *backpropagation*, além dos estados expandidos na árvore de estados. Para cada ação a no caminho, uma média global de vitórias $Q_h(a)$ é mantido em uma tabela de pesquisa e incrementalmente atualizada conforme o resultado final das simulações. $Q_h(a)$ representa a qualidade da ação. Ações com boa média, independente do estado de jogo, terão maior valor de qualidade. A ideia é que tais ações serão provavelmente melhores sempre quando estiverem disponíveis, como colocar uma peça no centro do tabuleiro no Jogo-da-velha ou nos cantos como no Reversi. Nas simulações, as escolhas das ações são enviesadas para escolher aquelas provavelmente melhores, através da amostragem de *Gibbs*, conforme a fórmula a seguir:

$$P(a) = \frac{e^{Q_h(a)/\tau}}{\sum_{b=1}^{n} e^{Q_h(b)/\tau}},$$

onde a é uma ação qualquer, n é o número de ações disponíveis e τ é um parâmetro que quanto maior for, mais homogênea será a distribuição gerada.

3.2.2 *Tree-Only MAST* (TO-MAST)

Tudo nesta técnica ocorre como na anterior, porém são atualizadas as médias $Q_h(a)$ durante o *backpropagation* apenas para as ações executadas na parte já adicionada

da árvore de busca, ou seja, o *backpropagation* executa conforme o algoritmo MCTS padrão, visitando apenas o caminho relativo aos estados já expandidos na árvore de estados. A ideia é que para tais ações, as estimativas de qualidade possuem menos variância, devido à maior chance de ações presentes na parte superior da árvore de busca serem selecionadas. Dessa forma TO-MAST prefere maior qualidade no conhecimento em detrimento da quantidade de informação utilizada para controlar a busca.

3.2.3 *Predicate-Average Sampling Technique* (PAST)

Nesta técnica o contexto é levado em consideração, ou seja, uma ação para ser boa depende do estado no qual a mesma é realizada. No contexto de GGP em jogos que utilizam a GDL, as características de um estado são representadas por predicados. Dessa forma os valores médios de qualidade, ao invés de serem armazenados para cada ação, são armazenados para pares de ação-predicado em $Q_p(p,a)$, onde p pertence ao conjunto de predicados do estado sendo percorrido na fase de *backpropagation*. Na amostragem de *Gibbs* é utilizado o par ação-predicado de maior valor médio para o estado sendo avaliado. Em PAST a amostragem de *Gibbs* somente é considerada quando um número suficiente de amostras é alcançado para o par sendo avaliado, caso contrário os pares apresentam valores igualmente neutros.

3.2.4 *Features-to-Action Sampling* (FAST)

Para aumentar a gama de jogos em que agentes de GGP conseguem jogar bem, alguns conceitos de mais alto nível devem ser compreendidos, como tipos de peças e a geometria do tabuleiro. Agentes de GGP que usam simples técnicas de controle de busca são prejudicados por não levarem em conta esses conceitos, muito importantes em jogos como o Xadrez, por exemplo. Tais *features* podem ser identificadas em uma descrição GDL utilizando-se de *template matching*. Utilizando essa estratégia são identificados tipos de peças e células do tabuleiro. Peças somente são consideradas no conjunto de *features* se houver mais de um tipo. Para aprender a importância relativa das *features* detectadas é utilizado o algoritmo de Aprendizado por Diferença Temporal $TD(\lambda)$.

A cada simulação um episódio é gerado e utilizado no algoritmo $TD(\lambda)$. Dois valores distintos para cada estado de jogo são calculados, um usando $TD(\lambda)$ que retorna o valor R_t^λ e outro usando uma função de valor $V(s_t)$ que é a soma ponderada de features observadas no estado. Os pesos associados a cada feature são aprendidos por uma regra delta para que a função de valor $V(s_t)$ se aproxime do valor retornado pelo $TD(\lambda)$ para

o estado no tempo t, conforme mostrado nas fórmulas a seguir:

$$V(s) = \sum^{|\vec{f}|} \theta_i \times f_i(s)$$

$$\vec{\delta} = \vec{\delta} + \alpha \times [R_t^{\lambda} - V(s_t)] \times \nabla_{\vec{\theta}} V(s_t),$$

onde α é um parâmetro *step size* e $\nabla_{\vec{\theta}} V(s_t)$ é o gradiente da função de valor $V(s_t)$. O resultado da regra delta é utilizado entre os episódios para atualizar os valores dos pesos θ_i associados a cada *feature* f_i. Para jogos com diferentes tipos de peças, cada *feature* $f_i(s)$ representa o número de peças desse tipo dado um estado s. Se o jogo não possui peças de mais um tipo então as *features* consideradas são as posições do tabuleiro que apresentam um valor binário, com peça ou sem uma peça, para cada jogador.

Durante as simulações, a função de valor do estado não é utilizada pois é caro efetuar a transferência para cada um dos estados filhos. Para evitar isso é utilizada a mesma estratégia de MAST, onde a qualidade da ação $Q_h(a)$ é calculada com base no tipo de *feature* sendo considerada. Para *features* associadas a tipo de peça a seguinte equação é utilizada:

$$Q_h(a) = \begin{cases} -(2 \times \theta_{\text{destino}} + \theta_{\text{origem}}), & \text{se ação de captura} \\ -100, & \text{caso contrário} \end{cases}$$

Dessa forma ações de captura são favorecidas, ainda mais aquelas em que a peça capturada é mais importante do que a que efetuou a captura. Para *features* associadas a posições no tabuleiro a seguinte equação é utilizada:

$$Q_h(a) = c \times \theta_{p,to}$$

onde $\theta_{p,to}$ é o peso da *feature* do jogador p colocar uma peça na posição to, e c é uma constante. Dessa forma ações são escolhidas pela distribuição $P(a)$ conforme ocorre em MAST.

Ao longo do jogo várias simulações são efetuadas e os pesos das features são constantemente atualizados, porem os mesmos somente são utilizados quando seus valores estão suficientemente longe de zero.

3.2.5 *Rapid Action Value Estimation* (RAVE)

RAVE é um método que acelera o processo de aprendizado dentro da árvore de jogo. Em Go este método é conhecido como uma das heurísticas *All-Moves-As-First* (AMAF) porque ele usa valores associados a ações realizadas mais abaixo no caminho simulado, e assim favorece mais amostras para ações similares disponíveis a partir da raiz de jogo. Quando o *backpropagation* do valor da simulação é realizado, além dos valores associados ao par estado-ação $Q(s,a)$, são atualizados as ações vizinhas $Q_{\text{rave}}(s,a')$, apenas se a' ocorrer abaixo a partir do caminho que o *backpropagation* é realizado. A medida que o número de simulações aumenta e os valores $Q(s,a)$ apresentam mais confiança, os mesmos devem ser mais relevantes do que os valores $Q_{\text{rave}}(s,a)$, e para tratar isso ambos são ponderados linearmente conforme mostrado a seguir:

$$\beta(s) \times Q_{\text{rave}}(s,a) + (1 - \beta(s)) \times Q_{\text{UCT}}(s,a)$$

$$\beta(s) = \sqrt{\frac{k}{3n(s) + k}}$$

O parâmetro k é denominado parâmetro de equivalência e controla quantas visitas em um estado são necessárias para que ambas as estimativas tenham pesos iguais. $n(s)$ retorna quantas vezes o estado s foi visitado.

3.2.6 *Early Cutoffs*

Sem um conhecimento para guiar as simulações MCTS, grande quantidade de computação é gasta em jogadas irrelevantes. Sem um conhecimento para que cortes possam ser realizados, primeiramente é necessário agrupar dados observando as simulações. Dentre esses dados estão a altura mínima e máxima de estados terminais. Se o intervalo entre a altura mínima e máxima for suficientemente longo significa que um corte antecipado da simulação poderá economizar mais tempo de computação. Portanto, com base nesse intervalo e em quantos estados já foram visitados na simulação atual, é heuristicamente determinada uma altura máxima para que uma simulação seja cortada.

Outra forma de interromper simulações é aproveitar a avaliação de um estado que não seja final, caso as regras do jogo em questão disponibilizem essa funcionalidade, e com base nessas avaliações analisar a variação delas ao longo da simulação. Uma medida denominada estabilidade é obtida a partir da média das variações observadas na pontuação fornecida ao longo da simulação. É importante observar que quanto mais próximo de zero estiver o valor da estabilidade, mais estável estará a avaliação dos estados. A estabilidade somente é considerada a partir do estado em que ocorre

a primeira mudança no valor da avaliação, dessa forma uma estabilidade não é considerada nos estados iniciais de jogo. Portanto, se o valor de estabilidade for menor do que um valor previamente determinado como limiar de estabilidade, mas não zero, então a pontuação é considerada estável e um corte é realizado a partir do estado no qual a pontuação se estabiliza mais o número de jogadores. Somar o número de jogadores permite considerar na avaliação do estado onde o corte será efetuado um possível contra-ataque dos oponentes.

3.2.7 *Unexplored Action Urgency*

No MCTS-UCT, enquanto todos os filhos de um estado não são explorados cada um uma vez, a fórmula UCT não é utilizada para controlar a busca nesse estado. Se conhecimento do domínio estivesse disponível, ações boas poderiam ser exploradas e as ruins ignoradas. O sucesso dessa abordagem depende claramente da precisão do conhecimento utilizado.

Nessa estratégia o agente deve escolher entre aplicar a estratégia de seleção UCT no subconjunto de ações exploradas no estado, ou efetuar jogadas simuladas no subconjunto de ações não exploradas. O valor associado às ações não exploradas é denominado *urgency*. São comparados o maior valor UCT do conjunto de ações já exploradas com o valor *urgency* das ações não exploradas. A fórmula para o valor *urgency* é praticamente a fórmula UCT, como se as ações tivessem sido exploradas uma única vez com resultado de simulação de empate (no caso do GGP esse valor é 50, visto que o valor de um estado final varia de 0 a 100):

$$urgency = 50 + C_p * \sqrt{\ln n(s)} * discount.$$

O parâmetro *discount* tem o propósito de diminuir o valor *urgency* conforme as ações vão sendo exploradas e tem seu valor calculado como a razão entre o número de ações não exploradas e o número total de ações disponíveis. Dessa forma quanto mais ações são exploradas, maiores são as chances de que uma ação boa já tenha sido encontrada.

Essa extensão deve estar relacionada com outras que favoreçam a seleção de boas ações o mais cedo possível para que a mesma tenha um efeito benéfico. Uma extensão que pode ser utilizada junto com a *Unexplored Action Urgency* é a MAST, por exemplo.

Capítulo 4

Metodologia

Neste capítulo é apresentada uma metodologia para a geração de agentes inteligentes genéricos para jogos de tabuleiro com dois jogadores, de soma zero, de informação perfeita, determinísticos, discretos e sequenciais. Primeiramente apresentamos uma visão geral da modelagem. Depois introduzimos a arquitetura do modelo implementado. Posteriormente cada elemento da arquitetura é detalhado para um completo entendimento do modelo.

4.1 Visão Geral

O algoritmo *Monte Carlo Tree Search* (MCTS) traz consigo muitas vantagens sendo a principal delas a não dependência de conhecimento específico de domínio, mas também apresenta alguns pontos negativos que devem ser considerados. Para se aplicar o MCTS em um problema, obrigatoriamente deve existir a possibilidade de modelar o mesmo como uma árvore, o que nem sempre é uma tarefa fácil, principalmente para domínios com ambientes dinâmicos e ações não determinísticas. Outro ponto problemático é que por não se apoiar em conhecimentos específicos do domínio, a exploração não necessariamente irá focar em porções interessantes do espaço de busca. Esse problema ocorre principalmente nas simulações iniciais devido à alta variância estatística dos dados, o que demanda um tempo de execução maior e ajustes na constante de exploração no caso da implementação *Upper Confidence Bounds for Tree* (UCT), a fim de aumentar as chances de explorar porções promissoras do espaço de busca.

Utilizar diretamente o MCTS-UCT para estimar a melhor ação a ser tomada dado um estado, apesar de possível, pode não ser o ideal. Em alguns jogos, principalmente

aqueles com uma alta complexidade da árvore de jogo[1], para estimar bons valores de utilidade para os estados filhos e escolher a ação que leva para aquele de maior utilidade, o MCTS-UCT deve efetuar um grande número de simulações, ou seja, deve rodar por mais tempo, o que não é prático ou mesmo permitido em jogos. Executar o MCTS e armazenar as utilidades processadas em uma espécie de tabela *hash* para cada estado de jogo também não é uma boa estratégia pelo fato de que muitos jogos possuem um espaço de estados exponencial, como é o caso do Xadrez e do GO.

Para contornar esse problema, a estratégia adotada neste trabalho é de generalizar uma função de avaliação a partir de exemplos de pares de estado-utilidade de um jogo qualquer. Esses exemplos seriam gerados pelo MCTS rodando o máximo de tempo que for possível. Quanto mais tempo, melhores serão os exemplos de pares gerados. Executar o MCTS por mais tempo para gerar bons exemplos de estado-utilidade não é o suficiente, visto que há outro parâmetro crítico que determinará a força do agente obtido. Tal parâmetro é a constante de exploração UCT. O valor dessa constante decide a importância dada ao bônus de exploração para estados poucos visitados. Quanto maior for a constante, mais serão valorizados estados poucos visitados. Não existe uma maneira óbvia para ajustar a constante de exploração sendo normalmente determinada experimentalmente. Para resolver este problema de forma automática, e assim garantir o mínimo de intervenção por parte do usuário, um processo estocástico de otimização determinado *Cross-Entropy Method* (CEM) será utilizado para estimar a melhor constante para o jogo em questão. A execução do CEM deve ser finalizada para que uma constante de exploração possa ser determinada, antes de executar as partidas MCTS para a geração de exemplos. O método CEM foi escolhido visto que o mesmo já foi utilizado com sucesso para estimar parâmetros do algoritmo MCTS e se mostrou uma das melhores e mais rápidas opções com convergência garantida [Chaslot et al., 2008].

Para generalizar a função de avaliação a partir de exemplos de estado-utilidade um algoritmo de aprendizado de máquina, como redes neurais, pode ser utilizado. O problema de redes neurais é que cada tipo de rede converge melhor para problemas específicos. Muitos parâmetros são ajustados, como taxa de aprendizagem e a arquitetura da rede, a fim de melhor adaptar o algoritmo para o problema em questão. No contexto deste trabalho não é possível realizar esse tipo de ajuste porque a proposta é aplicar o mesmo algoritmo para qualquer tipo de jogo que possa ser modelado como uma árvore. Uma alternativa seria parametrizar o algoritmo e deixar a cargo do usuário o ajuste dos parâmetros, com base no jogo para o qual o agente será gerado. A proposta deste trabalho é reduzir ao máximo o trabalho do usuário, para que o mesmo

[1] a medida de complexidade da árvore de jogo tem como limite inferior o fator de ramificação médio elevado à altura média da árvore de jogo.

possa gerar um agente para o seu jogo sem ter que implementar algoritmos de IA e nem realizar análise experimental para ajustar parâmetros.

Na tentativa de contornar as restrições no uso de redes neurais, é proposta a utilização do algoritmo *Cascade Correlation Neural Network* (CCNN), para o qual não é necessário especificar uma arquitetura de rede, ou seja, quantas camadas escondidas ou quantos neurônios em cada uma dessas camadas a rede deve apresentar. O algoritmo CCNN utiliza uma técnica de aprendizado construtivo, onde um neurônio é adicionado por vez em uma nova camada escondida. O algoritmo de aprendizado escolhido para ser utilizado no treinamento da rede CCNN foi o iRprop, por ser um algoritmo adaptativo para o qual não é necessário especificar parâmetros críticos como a taxa de aprendizado. A implementação CCNN escolhida foi a Cascade2 por ser melhor para problemas de regressão. As implementações do iRPROP e CCNN Cascade2 utilizadas neste trabalho são as da biblioteca *Fast Artificial Neural Network* (FANN) [Nissen, 2003] [Nissen & Nemerson, 2000].

Por fim, as redes neurais obtidas podem ser utilizadas diretamente, ou em algoritmos de busca como uma função de avaliação no caso do Minimax ou como uma política da árvore ou política padrão no caso do MCTS.

4.2 Arquitetura

Neste capítulo é proposto um método denominado Aprendizado Offline UCT-CCNN de Funções de Valor para Jogos de Tabuleiro (*Off-line UCT-CCNN Value Fuction Learning For Board Games*), ou resumidamente, Aprendizado UCT-CCNN (UCT-CCNN Learning). Esse método é utilizado para a geração offline de uma função de valor a ser utilizada em um agente inteligente para um jogo qualquer de soma zero, de informação perfeita, determinístico, discreto e sequencial. Tais características garantem a modelagem em árvore do jogo. Na Figura 4.1 é apresentado um esquema geral do processo de aprendizado UCT-CCNN.

Os elementos do método de aprendizado UCT-CCNN são discutidos neste trabalho em um contexto de linguagem de programação orientada a objetos. Apesar do método apresentado ter sido implementado em C++, as ideias serão apresentadas sem discutir detalhes de implementação relativos à essa linguagem. A seguir é apresentada a definição dos elementos presentes na Figura 4.1.

- **GeneralAgentTrainingConfig**: classe onde são definidos os parâmetros de configuração do algoritmo, os quais influenciam diretamente no tempo de treinamento e na precisão da função de valor a ser aprendida.

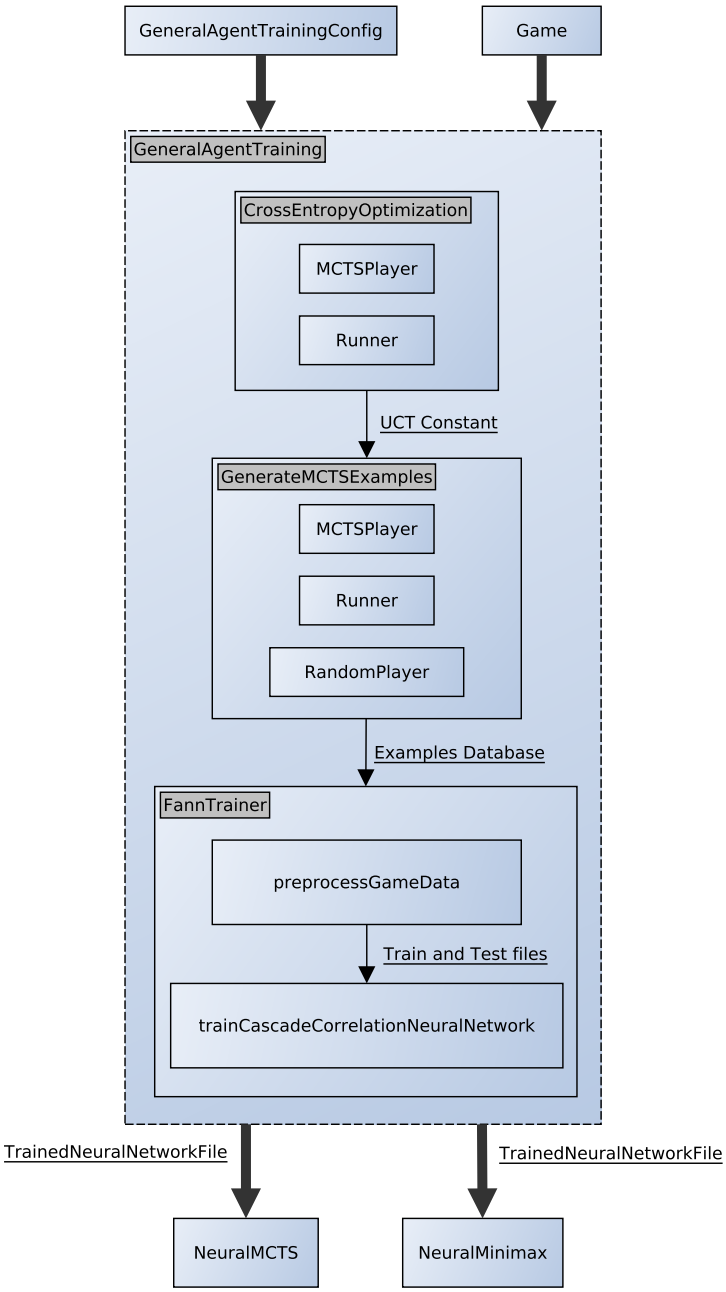


Figura 4.1: Arquitetura do Método de Aprendizado UCT-CCNN.

- **Game**: *interface* que define o contrato de como especificar as regras de um jogo, dentre as quais estão incluídas qual é o estado inicial, quais são as ações disponíveis dado um estado, dado um estado e uma ação qual o estado resultante, etc. São regras necessárias para modelar o jogo como uma árvore.

- **GeneralAgentTraining**: contém todo o processo do método de aprendizado UCT-CCNN. Deve receber instâncias dos tipos GeneralAgentTrainingConfig e

Game. Retorna uma rede neural como uma função de valor dos estados do jogo recebido como entrada.

- **CrossEntropyOptimization**: caracteriza um processo iterativo de otimização estocástica utilizado para encontrar o melhor valor para a constante de exploração UCT para o jogo recebido como entrada. Utiliza agentes do tipo MCTSPlayer e a classe de simulação de jogo Runner para executar múltiplas partidas em paralelo. Múltiplas partidas são executadas para obter o valor de utilidade esperada para cada exemplo da amostra de constantes de exploração gerada durante uma iteração do método *cross-entropy*.

- **GenerateMCTSExamples**: recebe uma constante de exploração UCT para utilizá-la no algoritmo MCTS. Múltiplas partidas são executadas entre dois agentes do tipo MCTSPlayer, a partir do estado inicial e também a partir de estados gerados aleatoriamente. Os estados aleatórios são extraídos de partidas executadas entre dois agentes do tipo RandomPlayer. Todas as jogadas e dados estatísticos são salvos em uma base de dados de jogo.

- **FannTrainer**: efetua um pré-processamento sobre a base de dados de jogo, prepara os dados de treino e de teste e treina a rede neural CCNN.

- **NeuralMinimax e NeuralMCTS**: agentes inteligentes nos quais a rede neural gerada pode ser utilizada como função de avaliação no caso do algoritmo Minimax ou como uma política da árvore no caso do algoritmo MCTS.

4.3 *Interface* de Descrição de Jogo

A *interface* de descrição de jogo é representada na Figura 4.1 como o elemento *Game*. Ela define o contrato de como representar as regras de um jogo qualquer em forma de árvore. As regras devem representar um jogo jogo genérico de dois jogadores, de soma zero, de informação perfeita, determinísticos, discretos e sequenciais. Um jogo é caracterizado por três interfaces: Ação, Estado e Jogo.

A implementação da interface Ação deve conter os dados necessários para representar uma ação no jogo relacionado. A implementação da interface Estado já inclui os dados referentes a um estado de jogo genérico, mas o usuário deve incluir informações adicionais necessárias para caracterizar um estado de jogo. A implementação da *interface* Jogo deve conter informações adicionais necessárias para controlar o fluxo de uma partida. Além disso, o usuário deve implementar a função de transferência do jogo,

a qual dado um estado e uma ação deve retornar um estado resultante com todas as informações atualizadas. Com a função de transferência é possível construir a árvore de jogo. A seguir, cada uma dessas interfaces é detalhada.

Uma Ação no contexto do método UCT-CCNN não possui nenhum dado prévio, ficando a cargo do usuário inserir dados em sua implementação conforme for necessário.

Já o tipo Estado armazena, independente do jogo, informações comuns a qualquer jogo aceito pelo método UCT-CCNN. Dentre elas estão: se o estado é terminal e caso positivo se alguém venceu e quem venceu ou se foi um empate; qual jogador efetuou a última ação; qual o jogador do turno atual; as possíveis ações do estado atual (agrupadas por simetria do tabuleiro resultante) e a ação executada no turno anterior. O usuário deve implementar em seu tipo Estado as seguintes funções: gerar uma cópia do objeto estado; gerar uma sequência binária para o objeto Estado (não é recomendado codificar as ações disponíveis na sequência binária, visto que tal sequência será utilizada pela rede neural para avaliar um estado resultante após uma ação ser realizada e codificar na sequência binária as ações disponíveis pode prejudicar o aprendizado da rede, pois quanto mais simples for tal sequência melhor). O aprendizado efetivo da rede neural CCNN depende muito de como um estado é codificado em uma sequência binária. Como a estrutura de um estado depende do jogo, essa codificação fica a cargo do usuário. A sequência binária deve refletir as informações básicas do estado, quanto mais simples for a sequência, sem redundância de informações, melhor será o aprendizado da rede.

O tipo Game não armazena nenhuma informação prévia, também ficando a cargo do usuário persistir dados em sua implementação. O usuário deve implementar as seguintes funções: retornar o estado inicial de jogo; dado um estado e uma ação retornar o estado resultante com todas as informações prévias do estado (listadas anteriormente) devidamente preenchidas.

Com a implementação dessas três *Interfaces* o jogo já estará formalmente caracterizado. Perceba que nenhuma estratégia de jogo foi exigida, apenas as regras básicas para modelar o jogo em forma de árvore. Como trabalho futuro pretende-se gerar a implementação de tais *Interfaces* a partir de uma linguagem de descrição de jogos, como a *General Description Language* [Love et al., 2008] utilizada na competição de *General Game Playing*.

4.4 Agentes de jogo

Antes de prosseguir para um detalhamento dos três elementos principais da arquitetura UCT-CCNN, apresentada na Figura 4.1, iremos apresentar nesta seção um detalhamento dos agentes de jogo utilizados no método UCT-CCNN. Neste trabalho foram utilizados quatro algoritmos diferentes de agentes de jogo que se dividem em dois grupos.

No primeiro grupo, relativos aos agentes utilizados no processo de aprendizagem, estão os agentes MCTSPlayer e RandomPlayer, presentes nos elementos *CrossEntropyOptimization* e *GenerateMCTSExamples* da Figura 4.1. No segundo grupo, relativos aos agentes utilizados depois do processo de aprendizagem, nos quais as funções de valor dos estados obtidas são aplicadas, estão os agentes NeuralMCTS e NeuralMinimax. Esses dois últimos agentes estão presentes na parte inferior da Figura 4.1, sendo que ambos recebem como entrada uma rede neural treinada, que representa a função de valor gerada para avaliar os estados de jogo.

4.4.1 MCTSPlayer

O agente MCTSPlayer é baseado na modalidade UCT do algoritmo MCTS, ou MCTS-UCT. Uma explicação em alto nível para o algoritmo MCTS-UCT foi dada na seção 2.1. No Algoritmo 1 é apresentada a implementação do agente MCTSPlayer. A seguir estão detalhadas as funções utilizadas no Algoritmo 1:

- **CriarNoEAdicionarNaArvore**(*state*) : cria um nó referente ao estado *state* e o adiciona como o nó raiz da árvore MCTS. Um nó da árvore MCTS contém os seguintes dados: se é um nó terminal, número de visitas, número de vitórias por jogador de simulações que passaram pelo nó, quem efetuou a jogada a partir do nó pai, nós filhos, ações não expandidas e o nó pai;
- **RecursosNaoExcedidos**(t, m, s) : atualiza as estatísticas de tempo de execução, utilização de memória e simulações executadas e encerra a busca MCTS caso alguma delas ultrapasse o tempo de execução t, o uso de memória m ou o número de simulações s;
- **CopiarEstado**(*state*) : função auxiliar que retorna uma cópia do estado *state*;
- **PoliticaDaArvore**(*node*, *estadoAuxiliar*) : Função com implementação na linha 16 do Algoritmo 1. A partir do nó raiz seleciona os nós usando a política da árvore UCT até atingir um nó ainda com ações não expandidas, escolhe uma aleatoriamente de maneira uniforme, expande o estado resultante e o retorna;

Algoritmo 1 MCTSPlayer

Require: *game* ▷ Implementação da *interface* de descrição de jogos
Require: t ▷ Tempo máximo em segundos de simulação para cada jogada
Require: m ▷ Uso máximo de memória em KB
Require: s ▷ Número máximo de simulações permitidas
Require: *uct* ▷ Constante de exploração UCT
Require: *generateNeuralExamples* ▷ Se exemplos de estado-utilidade serão gerados

```
▷ Recebe o estado de jogo atual e retorna ação a ser executada
procedure BuscaUCT(state)
    CriarNoEAdicionarNaArvore(state)
    while RecursosNaoExcedidos(t, m, s) do
        estadoAuxiliar ← CopiarEstado(state)
        node ← PoliticaDaArvore(ObterNoRaiz(), estadoAuxiliar)
        PoliticaPadrao(estadoAuxiliar)
        Backup(node, estadoAuxiliar)
    end while
    if generateNeuralExamples then
        GerarExemplosUtilidadeEstado(ObterNoRaiz())
    end if
    action ← ObterMelhorFilho(ObterNoRaiz(), 0)
    return action
end procedure

▷ Política da Árvore UCT para escolha de ações na árvore MCTS
procedure PoliticaDaArvore(node, estadoAuxiliar)
    while NaoETerminal(node) do
        if PossuiFilhoNaoExpandido(node) then
            node ← ExpandirProximoFilho(node)
            AtualizarEstadoAuxiliar(node, estadoAuxiliar)
            return node
        else
            node ← ObterMelhorFilho(node, uct)
            AtualizarEstadoAuxiliar(node, estadoAuxiliar)
        end if
    end while
    return node
end procedure
```

- **PoliticaPadrao**(*estadoAuxiliar*) : não adiciona nós na árvore, porém seleciona ações aleatoriamente a partir de uma distribuição uniforme até que um estado final seja alcançado. A cada escolha de ação o *estadoAuxiliar* é atualizado;

- **Backup**(*node*, *estadoFinal*) : A partir do nó expandido na fase de seleção, representado pelo nó *node*, volta nos nós pais até atingir a raiz, atualizando em cada nó desse caminho percorrido os valores de número de visitas e de número de vitória para cada jogador, com base no estado final de jogo, representado pelo *estadoFinal*;

- **GerarExemplosUtilidadeEstado**(**ObterNoRaiz**()) : percorre os primeiros nós filhos do nó raiz e armazena em uma base de dados, especifica para cada partida e jogador, exemplos contendo a representação binária do estado, o valor UCT calculado para o estado e o número de visitas ao estado;

- **ObterMelhorFilho**($node, uct$) : escolhe o melhor nó filho do nó pai *node* com base na fórmula UCT a seguir:

$$UCT = \frac{Q(child)}{V(child)} + uct \cdot \sqrt{\frac{2 \cdot \log V(parent)}{V(child)}},$$

  onde $Q(n)$ retorna a quantidade de vitórias menos a quantidade de derrotas referentes ao jogador que efetuou a ação no nó pai de n, $V(n)$ retorna o número de visitas do nó n e *uct* é a constante de exploração UCT;

- **NaoETerminal**($node$) : retorna verdadeiro se o nó não for referente a um estado de fim de jogo e falso caso contrário;

- **PossuiFilhoNaoExpandido**($node$) : retorna verdadeiro se o nó possuir ações de estados ainda não expandidos e falso caso contrário;

- **ExpandirProximoFilho**($node$) : seleciona aleatoriamente a partir de uma distribuição uniforme um dos filhos ainda não expandidos de *node* e o adiciona na árvore MCTS;

- **AtualizarEstadoAuxiliar**($node, estadoAuxiliar$) : atualiza o estado auxiliar armazenado para o próximo estado conforme ação efetuada pelo pai do nó *node*.

No final da execução do Algoritmo 1, na linha 14, o agente MCTSPlayer retorna a ação a partir do estado raiz que leva ao melhor estado filho. É importante observar que para retornar a melhor ação, o melhor estado filho é escolhido considerando a constante de exploração igual a zero. Dessa forma é retornado a ação que leva para o estado filho com maior utilidade média, representada pela fórmula $\frac{Q(child)}{V(child)}$.

4.4.2 RandomPlayer

O agente RandomPlayer simplesmente escolhe aleatoriamente uma ação dentre as disponíveis com base em uma distribuição uniforme. Ele é usado em partidas nas quais ambos os jogadores são do tipo RandomPlayer, para gerar aleatoriamente exemplos de estado de jogo válidos. Essa é uma forma de amostrar exemplos de estados de um jogo qualquer, que em muitos casos pode possuir um espaço de estados exponencial.

4.4.3 NeuralMCTS

O agente NeuralMCTS é bem parecido com o agente MCTSPlayer, porém utiliza em sua política da árvore uma rede neural como uma técnica de controle de busca, redi-

recionando o crescimento da árvore MCTS em direção a regiões mais interessantes do espaço de busca. A diferença principal em relação ao agente MCTSPlayer é que na função **PoliticaDaArvore**(*node*, *estadoAuxiliar*), com implementação na linha 16 do Algoritmo 1, não ocorre a verificação, presente na linha 18, da existência de nós ainda não expandidos, sendo executada apenas a parte do código *else* que começa na linha 24. No agente NeuralMCTS, quando é solicitada a escolha do melhor filho de um nó, todos os nós filhos são adicionados à árvore MCTS e o melhor filho é retornado. A função que avalia um nó, para a obtenção do melhor filho, é uma pequena variação da fórmula UCT, mostrada a seguir:

$$UCT = S(child) + B(child, parent, uct),$$

sendo

$$S(child) = \begin{cases} \frac{Q(child)}{V(child)}, & \text{se } V(child) > 0 \\ 1.0, & \text{caso contrário} \end{cases}$$

e

$$B(child, parent, uct) = \begin{cases} \frac{(C(child)+1)}{2} \cdot uct \cdot \sqrt{\frac{2 \cdot (1+\log V(parent))}{V(child)}}, & \text{se } V(child) > 0 \\ \frac{(C(child)+1)}{2}, & \text{caso contrário} \end{cases},$$

onde $S(child)$ representa a qualidade estatística do nó *child* e $B(child, parent, uct)$ representa o bônus de exploração para o nó *child*. No bônus de exploração, $C(child)$ é o valor do estado de jogo representado pelo nó *child*, dentro do intervalo $[-1,1]$, calculado por uma rede neural treinada pelo método de aprendizado UCT-CCNN. A função desse primeiro fator na fórmula do bônus de exploração é reduzir o bônus para estados menos vantajosos e representa uma probabilidade prévia calculada para o estado, assim como ocorre de maneira similar na estratégia adotada na política da árvore do algoritmo MCTS usado no AlphaGo, explicado na Seção 3.1. Outra mudança no bônus de exploração em relação à tradicional fórmula UCT é que o resultado do logaritmo do número de visitas do nó pai é acrescido de 1, para evitar que o bônus de exploração seja zero para estados visitados apenas uma vez. Não houve alteração na política padrão, dessa forma as ações são executadas aleatoriamente de maneira uniforme na fase de simulação.

4.4.4 NeuralMinimax

O agente do tipo NeuralMinimax é baseado no algoritmo de busca adversarial Minimax com poda Alfa-Beta [Knuth & Moore, 1975]. A diferença principal está na função de avaliação de um estado, que não é implementada manualmente. Para isso uma rede neural CCNN treinada pelo método UCT-CCNN é utilizada como função de avaliação para os estados de jogo.

4.5 Otimização da constante de exploração UCT com CEM

O algoritmo MCTS-UCT possui um parâmetro importante, denominado constante de exploração, presente na fórmula UCT. Ele é responsável por aumentar ou diminuir o valor relativo ao bônus de exploração. Ao aumentar o valor da constante de exploração, o bônus de exploração será consequentemente aumentado e a política da árvore irá favorecer mais estados pouco visitados. Infelizmente o valor ideal para esse parâmetro varia de acordo com o domínio, e normalmente uma análise experimental é realizada para determinar o melhor valor para o parâmetro. Como a proposta deste trabalho está relacionada à GGP, foi definido um método automático de otimização da constante de exploração que usa como base o algoritmo *Crosss Entropy Method* (CEM). O CEM já foi aplicado para identificar parâmetros do MCTS utilizando *self-play*, obtendo bons resultados [Chaslot et al., 2008].

Nesta seção primeiramente será apresentada a ideia em alto nível do processo de otimização da constante de exploração UCT com o método CEM. Posteriormente será apresentado um pseudocódigo desse processo e logo a seguir uma explicação mais detalhada dos procedimentos utilizados no pseudocódigo.

A primeira fase do processo UCT-CCNN consiste em identificar a melhor constante de exploração para o jogo caracterizado pela implementação da *interface* de descrição de jogo. Para isso, será utilizado o método de otimização estocástica *cross-entropy*, ou CEM. Essa etapa, identificada na Figura 4.1 como *CrossEntropyOptimization*, recebe do objeto *GeneralAgentTrainingConfig* os parâmetros relativos ao método de otimização *cross-entropy* que são: tamanho da população gerada a cada iteração; taxa de seleção para o conjunto elite de exemplos; o número máximo de iterações; *step-size* relativo à atualização das médias e desvio padrão da distribuição Gaussiana; número de repetições que define quantas partidas serão executadas para cada exemplo a fim de estimar a taxa de vitória, que seria a utilidade do exemplo; *threads* a serem

utilizadas, pois o método CEM demanda muito processamento por ter que executar múltiplas partidas para cada exemplo; o tempo de simulação dos agentes MCTS-UCT utilizados na execução das partidas.

O parâmetro a ser estimado é a constante de exploração UCT. Um exemplo gerado dessa constante deve ter seu valor no intervalo $[0{,}2; 2{,}0]$. Para gerar os exemplos uma distribuição Gaussiana será utilizada. Os parâmetros iniciais da distribuição devem ser definidos. Por padrão, a média inicial da distribuição é a média do menor e do maior valor do intervalo e o desvio padrão é a metade da distância entre o menor e o maior valor do intervalo. No caso da constante de exploração UCT a média da distribuição é $(0{,}2 + 2{,}0)/2 = 1{,}1$ e o desvio padrão é $(2{,}0 - 0{,}2)/2 = 0{,}9$. Dessa forma haverá maior probabilidade distribuída para os valores dentro do intervalo. Mesmo com probabilidades pequenas valores fora do intervalo podem ser gerados, estes são definidos como o valor do limite do intervalo mais próximo.

Para cada iteração do algoritmo são gerados n exemplos a partir da distribuição com os parâmetros de média e desvio padrão da iteração atual, onde n é o tamanho da população definido pelo usuário. Quanto maior for a população, maiores serão as chances de encontrar exemplos próximos do ponto ótimo. Para cada exemplo gerado, são executadas r partidas, sendo r o número de repetições a fim de se estimar a taxa de vitória. Quanto maior for o número de repetições melhor serão as estimativas. Uma partida é executada entre dois agentes do tipo MCTSPlayer, sendo que um deles utiliza a constante de exploração do exemplo gerado a ser avaliado e o oponente utiliza a constante de exploração correspondente à média da distribuição da iteração atual.

Depois de obtidas todas as taxas de sucessos associadas a cada exemplo, são selecionados os melhores exemplos para o conjunto elite, serão $n \cdot s$ elementos ao todo no conjunto elite, sendo s a taxa de seleção definida pelo usuário. Com base nos elementos do conjunto elite são calculados a média e o desvio padrão da distribuição com menor distância *cross-entropy* como parâmetros alvos, e então a média e desvio padrão atuais são atualizados em direção aos parâmetros alvos conforme o parâmetro *step-size*. Na próxima iteração, os novos valores da média e do desvio padrão serão utilizados como parâmetros da distribuição na geração dos exemplos e a média como constante UCT nas partidas para estimar a taxa de vitória dos exemplos. Esse processo se repete até que o número de iterações i seja alcançado ou até que o novo desvio padrão calculado seja muito próximo de zero, indicando uma convergência do processo.

Todo o processo descrito anteriormente é apresentado no Algoritmo 2. A seguir estão detalhadas as funções utilizadas no Algoritmo 2:

Algoritmo 2 Método Cross-Entropy para identificar melhor constante de exploração UCT

Require: *min* ▷ Valor mínimo para a constante de exploração UCT
Require: *max* ▷ Valor máximo para a constante de exploração UCT
Require: *i* ▷ Número máximo de iterações
Require: *n* ▷ Tamanho da população
Require: *r* ▷ Número de repetições para extração de taxa de vitória
Require: *s* ▷ Taxa de seleção para o conjunto elite
Require: *t* ▷ Tempo máximo em segundos de simulação MCTS para cada jogada
Require: *h* ▷ Threads utilizadas para executar partidas em paralelo
Require: *step* ▷ Parâmetro *step-size* para controlar convergência

1: $mediaAtual \leftarrow (min + max)/2$
2: $desvioPadraoAtual \leftarrow (max - min)/2$
3: **for** $x \leftarrow 1, i$ **do**
4: $populacao \leftarrow$ **GerarPopulacao**$(mediaAtual, desvioPadraoAtual, n)$
5: **for all** $p \in populacao$ **do**
6: **EfetuarPartidas**$(p, mediaAtual, r, t, h)$
7: **end for**
8: $tamanhoConjuntoElite \leftarrow$ **Floor**$(s \cdot n)$
9: $elite \leftarrow$ **ExtrairConjuntoElite**$(tamanhoConjuntoElite, populacao)$
10: $mediaAlvo \leftarrow \frac{\sum_{e \in elite}(e)}{tamanhoConjuntoElite}$
11: $desvioPadraoAlvo \leftarrow \sqrt{\frac{\sum_{e \in elite}(e - mediaAlvo)^2}{tamanhoConjuntoElite}}$
12: $mediaAtual \leftarrow step \cdot mediaAlvo + (1 - step) \cdot mediaAtual$
13: $desvioPadraoAtual \leftarrow step \cdot desvioPadraoAlvo + (1 - step) \cdot desvioPadraoAtual$
14: **if** $desvioPadraoAtual \approx 0$ **then return** $mediaAtual$
15: **end if**
16: **end for**
17: **return** $mediaAtual$

- **GerarPopulacao**$(mediaAtual, desvioPadraoAtual, n)$: gera n exemplos a partir de uma distribuição Gaussiana com os parâmetros $\mu = mediaAtual$ e $\sigma = desvioPadraoAtual$;

- **EfetuarPartidas**$(p, mediaAtual, r, t, h)$: executa partidas em paralelo a fim de se obter a taxa de vitórias para cada exemplo gerado executando r partidas contra o oponente do tipo MCTSPlayer com a constante UCT igual a média representada pelo valor *mediaAtual*, tempo máximo de simulação por jogada igual a t e h threads para execução de partidas em paralelo. Apesar de ser chamada para cada exemplo p da população, todas as partidas da população são executadas em paralelo;

- **ExtrairConjuntoElite**$(tamanhoConjuntoElite, populacao)$: espera o término das partidas para toda a população e ordena os exemplos em ordem decrescente por taxa de sucesso. Dos exemplos ordenados extrai os $s \cdot n$ primeiros exemplos para o conjunto elite, sendo s a taxa de seleção.

No final do algoritmo é retornado a *mediaAtual*, que será o valor da constante de exploração usada durante a fase de geração de exemplos para o treinamento da rede neural. Considere que uma partida de um jogo qualquer de tabuleiro com dois

jogadores tenha em média J jogadas, ou turnos. Sendo assim o processo de otimização cross-entropy irá demorar aproximadamente $\left(i \cdot \frac{n \cdot r \cdot J \cdot t}{h}\right)$ segundos, considerando que o número de threads não exceda o número de partidas a serem executadas durante as iterações. Perceba que somente é possível executar partidas em paralelo dentro de uma mesma iteração, pois é necessário o resultado de todas elas para obter as taxas de vitória e assim calcular os novos parâmetros para a próxima iteração.

Como exemplo, suponha um processo de otimização da constante de exploração para um jogo qualquer com parâmetros do CEM configurados da seguinte forma: iterações $i = 5$; população $n = 15$; repetições $r = 90$; jogadas por partida $J = 60$; tempo de simulação MCTS $t = 60$; threads $h = 64$. Com essa configuração de parâmetros o tempo de execução do processo de otimização CEM seria de $\left(5 \cdot \frac{15 \cdot 90 \cdot 60 \cdot 60}{64}\right) \cdot \frac{1}{3600} = 105{,}469$ horas, ou 4,394 dias. Conforme percebe-se nos cálculos do exemplo, o CEM é um processo bastante custoso exigindo uma máquina com muitas threads para viabilizar o método. Esse tempo de processamento relativo ao CEM pode ser eliminado se o usuário souber um bom valor de constante de exploração UCT para o jogo em questão. O método CEM somente precisa ser executado uma única vez, mesmo se o usuário tiver a intenção de gerar múltiplas redes neurais para agentes com forças diferentes, a fim de gerar diferentes níveis de dificuldade ou testar diferentes parâmetros das fases posteriores do processo UCT-CCNN.

4.6 Geração de exemplos off-line via MCTS-UCT

Nesta seção primeiramente será apresentada a ideia em alto nível do processo de geração de exemplos de estado-utilidade pela execução off-line de partidas entre agentes do tipo MCTSPlayer. Posteriormente será apresentado um pseudocódigo desse processo e logo a seguir uma explicação mais detalhada dos procedimentos utilizados no pseudocódigo.

A segunda fase do processo UCT-CCNN consiste em gerar exemplos de estado-utilidade a partir de partidas executadas entre agentes MCTSPlayer. A constante de exploração UCT a ser utilizada nas partidas é aquela obtida pelo método de otimização CEM. Se o usuário desejar, uma outra constante previamente informada pode ser utilizada, pulando a primeira fase referente à otimização CEM. Além disso nessa fase é crucial que o tempo de simulação dos agentes MCTSPlayer sejam o maior possível, o que exigirá mais recursos do sistema, a fim de gerar exemplos de estado-utilidade melhores.

Essa etapa, identificada na Figura 4.1 como *GenerateMCTSExamples*, recebe do objeto *GeneralAgentTrainingConfig* os seguintes parâmetros: número i de partidas

efetuadas a partir do estado inicial; número e de estados randômicos a serem gerados; número r de partidas efetuadas a partir de cada estado randômico gerado; o tempo t de simulação dos agentes MCTSPlayer utilizados na execução das partidas; o número h de *threads* a serem utilizadas para execução de partidas em paralelo. É possível especificar no *GeneralAgentTrainingConfig* se a etapa CEM será pulada, e neste caso é necessário especificar também a constante de exploração UCT a ser utilizada.

Primeiramente são executadas i partidas a partir do estado inicial padrão do jogo entre dois agentes MCTSPlayer, ambos com a constante de exploração UCT recebida como entrada. Quanto maior o tempo máximo de simulação t definido pelo usuário melhor serão os exemplos de estado-utilidade, no sentido das utilidades serem mais próximas da utilidade real do estado, devido a uma menor variância da estimativa. Tudo depende do tempo disponível do usuário, do poder computacional disponível em relação à quantidade de threads e memória, e da "força" alvo do agente, ou seja, se o usuário deseja um agente mais fácil ou difícil quando for jogar contra humanos.

Normalmente o agente MCTSPlayer apenas retorna a ação a partir do estado raiz que leva ao estado filho de maior utilidade média. Na fase de geração de exemplos, antes do agente retornar a melhor ação, os primeiros estados filhos da raiz são persistidos como exemplos em uma base de dados específica para cada partida e jogador. Todos os primeiros filhos da raiz são persistidos a fim de gerar exemplos tanto de estados bons quanto de estados ruins para uma melhor generalização da rede neural a ser treinada. Cada exemplo persistido contém o estado em sua forma binária, o número de visitas ao estado e a utilidade média calculada para o estado, representada pela fórmula $\frac{Q(child)}{V(child)}$.

A Figura 4.2 representa uma partida executada entre dois MCTSPlayer e os estados que são persistidos como exemplos. Antes do agente realizar uma jogada em um turno, os primeiros filhos do nó relativo ao estado atual de jogo, que é o nó raiz da árvore MCTS, são persistidos em uma base de dados específica para a partida e o jogador. Isso ocorre porque posteriormente uma filtragem é realizada nos exemplos persistidos e é necessário identificar os exemplos de uma partida relativos ao jogador que venceu a partida.

Quanto mais partidas são executadas maior será a quantidade de exemplos gerados, o que favorecerá o treinamento da rede neural. Múltiplas partidas são efetuadas por causa da natureza estocástica dos agentes MCTS, o que garante uma maior abrangência de exemplos gerados, visto que a cada partida os agentes MCTS exploram regiões diferentes do espaço de busca. Entretanto, como são dois agentes "fortes" jogando, será explorada uma porção do espaço de busca mais consistente com ações escolhidas por agentes "fortes". A fim de gerar exemplos para estados não muito comuns, princi-

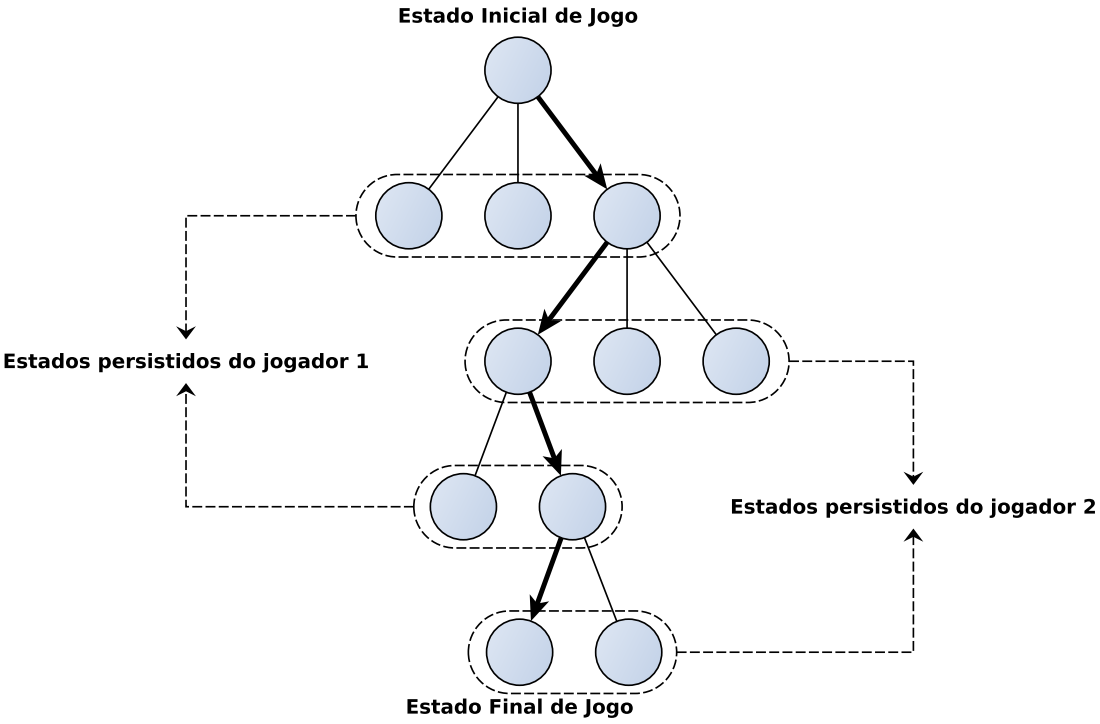


Figura 4.2: Estados que são persistidos como exemplos em uma partida.

palmente em situações nas quais o oponente não é experiente, também são efetuadas partidas que se iniciam em estados intermediários válidos, gerados aleatoriamente.

Para geração dos estados aleatórios partidas são efetuadas entre agentes do tipo RandomPlayer, que escolhem as ações dentre as ações válidas de forma homogeneamente distribuída. Os estados gerados durante a execução das partidas são armazenados em uma base de dados. Inicialmente são executadas $(e/J) + 1$ partidas, onde e é o número de estados aleatórios a serem gerados e J é a média de jogadas por partidas extraída das partidas efetuadas a partir do estado inicial. De todos os estados aleatórios gerados, são primeiramente recuperados ao longo das partidas os estados mais próximos do estado inicial, um por partida, até que e estados aleatórios únicos sejam recuperados. Se faltarem estados aleatórios, novas partidas com os agentes RandomPlayer são executadas até que a quantidade e seja alcançada.

Quando e estados aleatórios forem gerados, r partidas são efetuadas entre agentes do tipo MCTSPlayer para cada estado aleatório como estado inicial. Assim como as partidas a partir do estado inicial padrão, a constante de exploração utilidade é a calculada pelo método CEM ou informada pelo usuário e os exemplos gerados são persistidos na base de dados utilizando-se da mesma lógica.

O processo descrito no parágrafo anterior garante um maior foco nos estados mais próximos da raiz da árvore de jogo na geração de partidas que se iniciam a partir de estados intermediários aleatórios. Isso garante que uma gama maior de exemplos serão gerados por partidas, visto que o estado intermediário aleatório estará mais próximo do início da árvore.

Todo o processo de geração de exemplos está descrito no Algoritmo 3. A seguir estão detalhadas as funções utilizadas no Algoritmo 3:

Algorithm 3 Geração de exemplos estado-utilidade para treinamento da rede neural

Require: i ▷ Partidas executadas a partir do estado inicial
Require: e ▷ Estados aleatórios a serem gerados
Require: r ▷ Partidas executadas por estado aleatório como estado inicial
Require: t ▷ Tempo máximo em segundos de simulação MCTS para cada jogada
Require: h ▷ Threads utilizadas para executar partidas em paralelo
Require: uct ▷ Constante UCT informada pelo método CEM ou pelo usuário

1: *exemplosEstadoUtilidade* ← **CriarBaseDeDadosDeExemplos**()
2: *baseDeDados* ← **CriarBaseDeDadosDeJogo**()

3: ▷ A base de dados de jogo vazia faz as partidas começarem com estado inicial padrão.
4: ▷ Gera exemplos de estado-utilidade:
5: **EfetuarPartidasGerandoExemplos**(*baseDeDados*, i, *uct*, *exemplosEstadoUtilidade*)

6: ▷ Estima quantas partidas jogar para gerar e estados aleatórios:
7: *Jogadas* ← **ObterMediaDeJogadasPorPartida**(*baseDeDados*)
8: *partidasRandomicas* ← $(e/Jogadas) + 1$
9: *baseAuxiliar* ← **CriarBaseDeDadosDeJogo**()
10: **EfetuarPartidasRandomicas**(*baseAuxiliar*, *partidasRandomicas*)
11: *baseDeDadosRandom* ← **CriarBaseDeDadosDeJogo**()
12: *comEstados* ← *False*

13: ▷ Extrai estados de partidas randômicas previamente executadas:
14: **while ObterPartidas**(*baseDeDadosRandom*) $< e \cap$ *comEstados* **do**
15: *comEstados* ← *False*
16: **for** $x \leftarrow 1$, *partidasRandomicas* **do**
17: **if PartidaTemEstado**(x, *baseAuxiliar*) **then**
18: *estadoRandom* ← **RemoverProximoEstado**(x, *baseAuxiliar*)
19: **AdicionarPartidaComEstadoInicial**(*baseDeDadosRandom*, *estadoRandom*, r)
20: *comEstados* ← *True*
21: **end if**
22: **end for**
23: **end while**

24: ▷ A base deve ter $e \cdot r$ partidas com estado inicial randômico, completar se faltar:
25: **PreencherRestanteDePartidasRandom**(*baseDeDadosRandom*, e, r)

26: ▷ Partidas começam com estados aleatórios pois a base de dados está preenchida.
27: ▷ Gera exemplos de estado-utilidade:
28: **EfetuarPartidasGerandoExemplos**(*baseDeDadosRandom*, $e \cdot r$, *uct*, *exemplosEstadoUtilidade*)

29: **return** *exemplosEstadoUtilidade*

- **CriarBaseDeDadosDeExemplos**() : retorna uma nova base de dados onde serão armazenados os exemplos de estado-utilidade a serem utilizados para o treinamento da rede neural. No final do algoritmo essa base de dados preenchida é retornada, na linha 29;

- **CriarBaseDeDadosDeJogo**() : retorna uma nova base de dados que armazena os dados relativos às partidas;
- **EfetuarPartidasGerandoExemplos**(*baseDeDados*, i, *uct*, *exemplosEstadoUtilidade*) : efetua i partidas entre dois oponentes do tipo MCTSPlayer com a constante de exploração *uct*. A cada decisão dos agentes, os exemplos relativos aos primeiros estados filhos são salvos na base de dados *exemplosEstadoUtilidade*, onde os exemplos são separados em grupos por partida e por jogador. Se a *baseDeDados* fornecida estiver vazia partidas são executadas a partir do estado inicial padrão do jogo, caso a *baseDeDados* esteja preenchida as partidas "em andamento" serão resumidas a partir dos estados intermediários recuperados da base;
- **ObterMediaDeJogadasPorPartida**(*baseDeDados*) : recupera da *baseDeDados* informada o número médio de jogadas por partida conforme os dados cadastrados;
- **EfetuarPartidasRandomicas**(*baseAuxiliar*, *partidasRandomicas*) : efetua o número de partidas informado por *partidasRandomicas* entre dois agentes do tipo RandomPlayer e armazena os dados das partidas executadas na base de dados *baseAuxiliar* informada;
- **ObterPartidas**(*baseDeDadosRandom*) : recupera da base de dados *baseDeDadosRandom* o número de partidas finalizadas;
- **PartidaTemEstado**(x, *baseAuxiliar*) : verifica na base de dados *baseAuxiliar* se a partida com id x possui estados em sua lista de estados visitados;
- **RemoverProximoEstado**(x, *baseAuxiliar*) : remove e retorna o próximo estado da lista de estados da partida de id x na base de dados *baseAuxiliar*. Vale ressaltar que a lista de estados dessas partidas estão ordenadas por ordem de visita, portando os estados vão sendo recuperados do inicio da partida para o fim a cada chamada da função **RemoverProximoEstado**;
- **AdicionarPartidaComEstadoInicial**(*baseDeDadosRandom*, *estadoRandom*, r) : adiciona na base de dados de jogo *baseDeDadosRandom* r partidas pendentes com estado inicial *estadoRandom*, caso não exista nenhuma partida com esse estado já cadastrada. Partidas pendentes em uma base de dados são resumidas quando a base é utilizada para execução de partidas;
- **PreencherRestanteDePartidasRandom**(*baseDeDadosRandom*, e, r) : irá verificar se a *baseDeDadosRandom* possui $e \cdot r$ partidas pendentes registradas.

Caso tenha menos que isso, uma partida entre dois agentes do tipo *RandomPlayer* é executada, os estados são extraídos e inseridos na base de dados *baseDeDadosRandom* caso sejam estados novos. Novas partidas são executadas até que a base de dados *baseDeDadosRandom* possua $e \cdot r$ partidas pendentes com estados iniciais aleatórios (partidas que se iniciam a partir de estados intermediários aletatórios).

Considere que uma partida de um jogo qualquer de tabuleiro com dois jogadores tenha em média J jogadas, ou turnos. Sendo assim o processo de geração de exemplos irá demorar aproximadamente $\left(\frac{(i+e\cdot r)\cdot J\cdot t}{h}\right)$ segundos. Como exemplo, suponha um processo de geração de exemplos para um jogo qualquer com parâmetros configurados da seguinte forma: jogadas por partida $J = 60$; tempo de simulação MCTS $t = 200$ segundos; threads $h = 64$; partidas executadas a partir do estado inicial $i = 30$; partidas executadas a partir de estado intermediário aleatório $r = 5$; número de estados intermediários aleatórios $e = 300$. Com essa configuração de parâmetros o tempo de execução do processo de geração de exemplos seria de $\left(\frac{(30+300\cdot 5)\cdot 60\cdot 200}{64}\right) \cdot \frac{1}{3600} = 79{,}687$ horas, ou 3,32 dias. Essa é uma das fases mais importantes do método UCT-CCNN e a ênfase deve ser dada no tempo de simulação do agente MCTSPlayer, que deve ser o maior possível, conforme os recursos disponíveis.

4.7 Filtragem e Treinamento de rede CCNN

Nesta seção será apresentado o processo de preparação dos dados de teste e de treino e o treinamento da rede neural CCNN. As partidas executadas a partir de estados iniciais padrão do jogo, assim como as partidas executadas a partir de estados iniciais aleatórios, geram exemplos em uma base de dados onde os mesmos estão separados por partida e por jogador. Cada exemplo contém um estado de jogo em sua forma binária, a utilidade média calculada pelo algoritmo MCTS-UCT, e o número de visitas ao estado.

Essa etapa, identificada na Figura 4.1 como *FannTrainer*, recebe do objeto *GeneralAgentTrainingConfig* a porcentagem t de dados de teste a serem separados do total de dados gerados para o treino da rede neural. Esse conjunto separado de testes será usado para avaliar a capacidade de generalização da rede treinada.

Primeiramente a base de dados retornada pela etapa de geração de exemplos é percorrida e uma filtragem é efetuada nos exemplos a fim de considerar exemplos melhores para a rede neural, uma maneira de contornar a alta variância na estimativa das utilidades do algoritmo MCTS-UCT. De todos os exemplos gerados em uma partida,

são considerados apenas os exemplos do jogador que venceu a partida, porque provavelmente as utilidades estimadas dos estados levaram a escolhas mais próximas de uma política ótima.

Entre as diferentes partidas efetuadas, estados iguais podem ter sido gerados como exemplos. Para favorecer um melhor treino da rede neural é melhor considerar a utilidade que levará a uma política mais próxima da política ótima. A utilidade calculada por um maior número de simulações apresenta uma menor variabilidade estatística, se aproximando da utilidade real do estado (aquela que leva a uma política ótima). Sendo assim, em caso de estados repetidos é mantido aquele com maior número de visitas. Esse processo de filtragem é identificado na Figura 4.1 por *preprocessGameData*. Outra possibilidade seria computar um novo valor de utilidade com base nas utilidades médias obtidas para os estados repetidos.

Espera-se que com essa filtragem valores mais consistentes de utilidade para os exemplos sejam fornecidos para a rede, levando a um menor erro no teste. Terminada a filtragem de exemplos, a base de exemplos é separada em dois conjuntos de dados, conforme parâmetro t relativo à porcentagem do total de exemplos a ser utilizada no conjunto de teste. O valor padrão para esse parâmetro é 10%. Os exemplos de teste são escolhidos aleatoriamente a partir de uma distribuição uniforme sobre todos os exemplos, até que um conjunto de testes de $t\%$ seja alcançado. O restante dos exemplos são separados em um conjunto de treino. A rede não é treinada com os exemplos contidos no conjunto de teste.

A rede neural utilizada é do tipo *Cascade Correlation Neural Network* (CCNN). O funcionamento da rede CCNN é explicado na Seção 2.2. A implementação utilizada da CCNN é a Cascade2, própria para problemas de regressão. O algoritmo de treino utilizado no ajuste de pesos da rede é o iRPROP com funcionamento também descrito na Seção 2.2. As implementações iRPROP e Cascade2 utilizadas neste trabalho são as da biblioteca *Fast Artificial Neural Network* (FANN) [Nissen, 2003] [Nissen & Nemerson, 2000] [Nissen, 2007]. Toda configuração específica de parâmetros será listada, o restante utiliza a configuração padrão de parâmetros definida pela biblioteca FANN. O processo de treino é identificado na Figura 4.1 por *trainCascadeCorrelationNeuralNetwork*.

Mesmo sendo um algoritmo construtivo é necessário especificar a configuração de entrada e saída da rede neural, visto que nas redes CCNN são adicionados apenas neurônios escondidos. Portanto foi definido que o número de neurônios na camada de entrada é equivalente ao número de bits na representação binária do estado de jogo. Cada neurônio de entrada recebe 0 ou 1 conforme a sequência binária que representa o estado de jogo a ser avaliado. A camada de saída consiste de um único neurônio que resulta na utilidade calculada para o estado recebido como entrada.

O valor de utilidade estimada retornado pelo algoritmo MCTS-UCT também varia de [−1,1], pois quando a constante de exploração é zerada durante a geração de exemplos a fórmula UCT se reduz na equação $\frac{Q(child)}{V(child)}$, onde $Q(child)$ é o número de vitórias menos o número de derrotas do jogador que efetuou a jogada no estado pai e $V(child)$ é o número de visitas ao estado filho. Se em todas as n visitas ao estado filho o jogador que efetuou a ação ganhou no resultado da simulação a equação resultaria em $(n-0)/n = 1$, e se em todas as n visitas ao estado filho o jogador perdeu a equação resultaria em $(0-n)/n = -1$.

Portanto foi escolhida como função de ativação no único neurônio de saída da rede a função tangente hiperbólica $tanh(x) = \frac{1}{1+\exp^{-x}}$, cujo resultado está no intervalo [−1,1]. Um dos pontos negativos dessa abordagem é que a rede neural aprende a avaliar um estado isoladamente, sem uma transição representada por um estado mais uma ação. Dessa forma para avaliar um estado filho é necessário calcular a transição do estado pai com a ação executada para chegar ao estado filho. Devido a essa característica, a rede neural é melhor aproveitada em situações que a transição de um estado a outro seria calculada de qualquer maneira. Para situações em que deve-se apenas escolher a melhor ação a partir de um estado pai é necessário calcular a transição para todos os estados filhos, para aí sim avaliá-los com a rede neural treinada.

A cada adição de neurônio é calculado o erro médio quadrático no conjunto de treino e no conjunto de teste. A cada iteração uma cópia da rede construída até o momento é salva junto com o erro de teste obtido. O treinamento da rede continua até que o número máximo de neurônios seja alcançado.

Quando o treino da rede neural é finalizado é obtida a iteração da rede neural que obteve o menor erro no conjunto de teste, pois é a rede que melhor generalizou para os estados nunca antes vistos. O arquivo referente a melhor rede obtida, com menor erro no teste, é retornado pelo algoritmo. Esse arquivo pode ser utilizado nos agentes do tipo NeuralMinimax e NeuralMCTS. No Capítulo 5 serão mostrado quatro exemplos da evolução do erro no processo de treino e a utilização das redes obtidas nos dois tipos de agentes. Para saber como treinar uma rede CCNN usando a biblioteca FANN consulte o Apêndice A.

Depois de treinada, o arquivo referente a rede neural com menor erro no conjunto de teste é retornado. Essa rede neural treinada representa uma função de valor para avaliar estados do jogo cujas regras foram especificadas pelas classes de descrição de jogo. Isso caracteriza o processo UCT-CCNN como uma forma de gerar funções de valores $Q(s)$, onde s é um estado qualquer de jogo, para quaisquer jogos de tabuleiro de dois jogadores, de soma zero, de informação perfeita, determinístico, discreto e sequencial.

No Capítulo 5, análises experimentais são realizadas para determinar a "força" da função de valor $Q(s)$ gerada pelo método UCT-CCNN, ou seja, o quanto a função de valor se aproxima de uma função de valor real para um estado s, dada uma política ótima $P(s',a)$, onde s' é o estado pai de s e a é a ação que leva ao estado s. Medir essa proximidade não é trivial, visto que não se sabe qual é a política ótima para o jogo em questão. Portando é necessário comparar a função obtida com outros agentes ou funções para se ter uma ideia de sua força.

Capítulo 5

Análise Experimental

Neste capítulo iremos analisar o método UCT-CCNN através de dois jogos: Trilha (ou *Nine Men's Morris*) e Othello (ou *Reversi*). Primeiramente são definidas as implementações das *interfaces* de descrição de jogo para os dois jogos. Posteriormente são otimizadas as constantes de exploração UCT para cada um dos jogos através do método CEM e o comportamento da convergência das constantes é apresentado. Em seguida, na fase de geração de exemplos via MCTS-UCT, para cada jogo são gerados dois grupos distintos de exemplos: no primeiro o tempo limite de simulação para o MCTS é de 200 segundos e no segundo de 600 segundos. Por fim, redes neurais CCNN são treinadas uma para cada grupo de exemplos gerados na fase anterior, totalizando duas redes neurais para cada jogo. As redes neurais obtidas são utilizadas em agentes NeuralMinimax e NeuralMCTS para avaliar a força da função de valor associada a cada uma das redes.

5.1 Implementações das *Interfaces* de Descrição de Jogo

Para cada jogo são implementadas as *interfaces* de descrição de jogo Ação, Estado e Jogo. As implementações dessas *interfaces* definem as regras dos jogos em questão. Nesta seção apresentaremos uma visão geral de cada jogo seguida dos detalhes de implementação das *interfaces*.

5.1.1 Othello ou *Reversi*

Othello é um jogo de tabuleiro com tamanho 8x8, de dois jogadores, com 64 peças que são pretas de um lado e brancas do outro. O jogador que inicia a partida deve colocar

as peças no tabuleiro com o lado preto para cima. O jogo começa com a configuração de tabuleiro mostrada na Figura 5.1. Cada jogador deve colocar uma peça em uma posição do tabuleiro vizinha a uma peça do adversário e que resulte em pelo menos uma captura de peça. Quando um jogador coloca uma peça, as peças do oponente que ficarem entre duas peças do jogador, sendo uma delas a peça colocada, são capturadas e devem ser viradas. Essas regras valem na vertical, horizontal e diagonal. O jogo acaba quando nenhum dos jogadores conseguem efetuar jogadas válidas e o jogador com a maior quantidade de peças no tabuleiro ganha. A seguir, as regras do jogo Othello serão apresentadas em termos das interfaces de descrição de jogo.

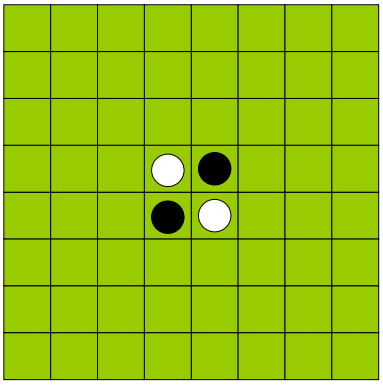


Figura 5.1: Jogo Othello em sua configuração inicial.

A implementação da *interface* Ação para o jogo Othello requer a criação de dois campos: um campo indicando se o jogador passará a vez devido a nenhuma ação válida disponível e outro com uma posição válida onde será inserida a peça do jogador.

A *interface* Estado é implementada com a criação de apenas um campo relativo a um estado do jogo Othello, que é o próprio tabuleiro de jogo, modelado como uma matriz de caracteres, como mostrado na Figura 5.2, onde: (*) representa um espaço vazio, (B) uma peça da cor preta e (W) uma peça da cor branca.

A representação binária de um estado é uma simples sequência binária de dois bits para cada posição do tabuleiro, da linha superior até a linha inferior da matriz, da esquerda para direita. Na Tabela 5.1 é mostrado o mapeamento binário para cada *feature* do jogo Othello. Em um estado de jogo, cada posição do tabuleiro apresenta uma *feature*. A *feature* relativa à "peça do jogador" diz respeito à peça do jogador que efetuou a ação que resultou no estado atual de jogo. Essa distinção entre quem efetuou a jogada e o outro jogador é importante para que a rede neural consiga aprender exemplos para qualquer estado de jogo. Essa representação binária é passada na camada de entrada da rede neural durante a fase de treino. Como o tabuleiro tem $8 \times 8 = 64$

posições, o número total de neurônios na camada de entrada será de $64 \times 2 = 128$ neurônios.

Feature	**Codificação Binária**
Espaço Vazio (*)	00
Peça do jogador	01
Peça do oponente	10

Tabela 5.1: Mapeamento binário das *features* do jogo Othello.

A *interface* Game é implementada sem a criação de novos campos, ou seja, nenhuma informação adicional é armazenada. O estado inicial de jogo, onde o jogador B começa, é definido pela matriz de caracteres representada na Figura 5.2.

	0	1	2	3	4	5	6	7
0	*	*	*	*	*	*	*	*
1	*	*	*	*	*	*	*	*
2	*	*	*	*	*	*	*	*
3	*	*	*	W	B	*	*	*
4	*	*	*	B	W	*	*	*
5	*	*	*	*	*	*	*	*
6	*	*	*	*	*	*	*	*
7	*	*	*	*	*	*	*	*

Figura 5.2: Matriz de caracteres representando o tabuleiro do jogo Othello.

As possíveis ações a serem executadas são aquelas nas quais uma peça do jogador é colocada junto a uma peça do adversário de forma que uma ou mais peças do adversário fiquem entre duas peças do jogador, tanto na vertical, horizontal e/ou diagonal, sendo uma destas peças a peça sendo colocada no tabuleiro. Todas as peças do adversário entre duas peças do jogador, sendo uma delas a peça colocada, são convertidas em peças do jogador. Caso não existam posições que possibilitam a captura de peças do adversário, a única ação disponível então é a ação nula, ou seja, onde o jogador passa a vez sem fazer nada.

Ao atualizar um estado com uma ação, o estado resultante é marcado como estado final se nas duas jogadas anteriores ambos os jogadores passaram a vez com ações nulas, sendo o jogador vencedor aquele com o maior número de peças ou empate em caso de mesmo número de peças. Na Figura 5.3 é mostrada uma atualização de estado com a ação do jogador de peça cor branca com a captura de duas peças do adversário.

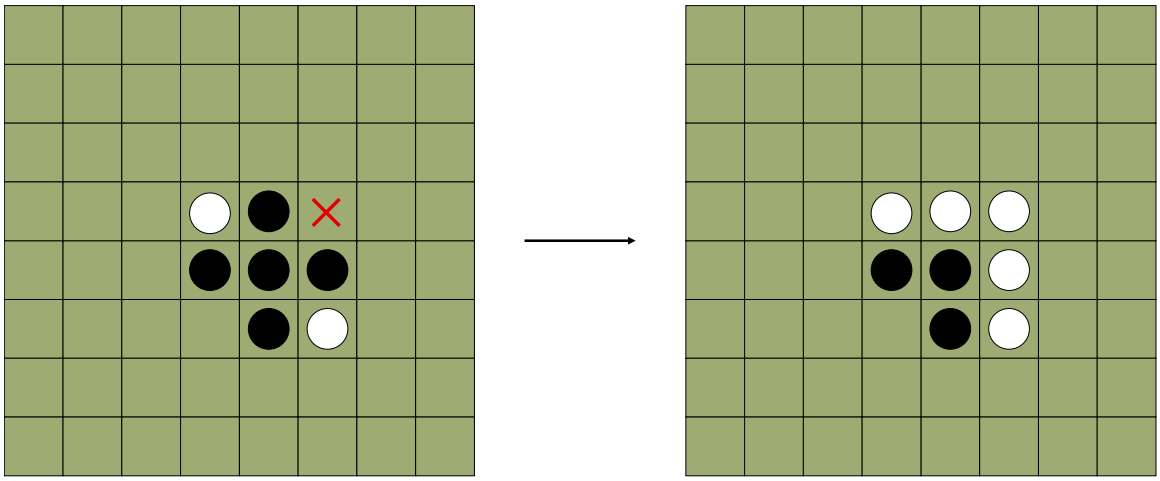


Figura 5.3: Representação de uma ação do jogador (W) no jogo Othello.

5.1.2 Trilha ou *Nine Men's Morris*

Trilha é um jogo de tabuleiro com 24 pontos de interseção, onde as peças são posicionadas, de dois jogadores. Cada jogador escolhe uma cor de peça. Uma possível situação de jogo é exibida na Figura 5.4. O jogo começa com o tabuleiro vazio e cada jogador possui 9 peças em mãos. Cada jogador começa na primeira fase, onde as peças devem ser colocadas no tabuleiro em qualquer posição livre. Quando um jogador colocar três peças alinhadas consecutivamente, na horizontal ou vertical, ele forma uma trilha e pode escolher uma peça do adversário para remoção. Obrigatoriamente, na remoção deve-se dar preferência para peças do adversário que não estejam em uma trilha. Quando um jogador colocar todas as suas peças, ele passa para a segunda fase. Na segunda fase um jogador pode mover peças para posições livres adjacentes. Um jogador entra na terceira fase quando restam apenas três de suas peças no tabuleiro. Na terceira fase um jogador pode mover peças para qualquer posição livre do tabuleiro. O jogador perde quando sobram apenas duas de suas peças no tabuleiro e o jogo termina em empate quando uma configuração de tabuleiro se repetir. A seguir, as regras do jogo Trilha serão apresentadas em termos das interfaces de descrição de jogo.

A implementação da *interface* Ação para o jogo Trilha requer a criação de quatro campos:

- O tipo de movimento, se é de colocação, de colocação seguida de captura, de movimento ou de movimento seguido de captura;
- Posição onde será colocada uma peça ou de onde será movida uma peça;
- Posição para a qual será movida uma peça;

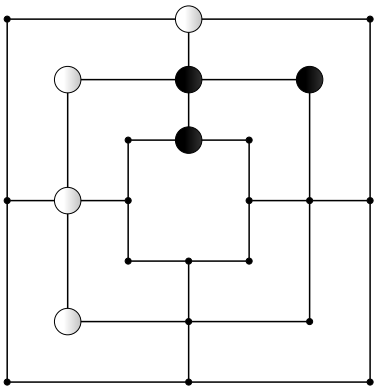


Figura 5.4: Uma possível configuração do jogo Trilha.

- Posição da qual será capturada uma peça do adversário.

A *interface* Estado é implementada com a criação de seis novos campos, relativos a um estado do jogo Trilha, que são:

- O tipo de ação a ser executada pelo jogador atual;
- O total de peças no tabuleiro para cada jogador;
- O total de peças nas mãos de cada jogador (ainda não colocadas no tabuleiro);
- O tabuleiro modelado como uma matriz de caracteres, como mostrado na Figura 5.5, onde: (o) representa um espaço vazio, (B) uma peça da cor preta, (W) uma peça da cor branca, (-) e (|) arestas que ligam as interseções onde as peças são colocadas e (+) o centro do tabuleiro;
- A fase de jogo de cada jogador, que são: Fase 1 de colocação de peças, Fase 2 de movimentação de peças ou Fase 3 de saltos de peças;
- Um histórico das últimas três ações executadas (para detectar uma sequência de jogada repetida ou um loop de jogo).

A representação binária de um estado é uma simples sequência binária de três bits para cada posição do tabuleiro, da linha superior até a linha inferior da matriz, da esquerda para direita, e de mais dois bits indicando a fase atual do jogador. Na Tabela 5.2 é mostrado o mapeamento binário para cada *feature* do jogo Trilha. Essa representação binária é passada na camada de entrada da rede neural durante a fase de treino. Como o tabuleiro tem $7 \times 7 = 49$ posições, e além do tabuleiro também devemos mapear a fase atual do jogador, o número total de neurônios na camada de

entrada será de $49 \times 3 + 2 = 149$ neurônios. Outras informações como quantidade de peças na mão do jogador e quantidade de peças capturadas não foram codificadas para manter a representação binária do estado o mais simples possível e a informação da fase de jogo já é suficiente para diferenciar estados do começo e do fim de jogo.

Tipo de *Feature*	***Feature***	**Codificação Binária**
De Tabuleiro	Espaço Vazio (o)	000
	Aresta Horizontal (-)	001
	Aresta Vertical (|)	010
	Centro do Tabuleiro (+)	011
	Peça do jogador	100
	Peça do oponente	101
De Jogador	Jogador na Fase 1	00
	Jogador na Fase 2	01
	Jogador na Fase 3	10

Tabela 5.2: Mapeamento binário das *features* do jogo Trilha.

A *interface* Game é implementada sem a criação de novos campos, ou seja, nenhuma informação adicional é armazenada. No estado inicial de jogo, ficou definido que o jogador W começa. Cada jogador possui em mãos 9 peças e ambos começam na Fase 1. Esse estado inicial é definido pela matriz de caracteres representada na Figura 5.5

	0	1	2	3	4	5	6
0	o	−	−	o	−	−	o
1	|	o	−	o	−	o	|
2	|	|	o	o	o	|	|
3	o	o	o	+	o	o	o
4	|	|	o	o	o	|	|
5	|	o	−	o	−	o	|
6	o	−	−	o	−	−	o

Figura 5.5: Matriz de caracteres representando o tabuleiro do jogo Trilha.

Ao atualizar um estado com uma ação, no estado resultante também devem ser atualizadas as ações válidas disponíveis, a fase de jogo de cada jogador e se o estado resultante é um estado final.

As ações válidas em um estado são: se o jogador estiver na Fase 1, serão válidas as ações de colocação de uma peça em um espaço vazio; se o jogador estiver na Fase 2, serão válidas as ações de movimentação de uma peça para posições adjacentes vazias; se o jogador estiver na Fase 3 serão válidas as ações de movimentação de uma peça para qualquer posição vazia do tabuleiro. Qualquer colocação ou movimentação com formação de uma sequência de três peças, na horizontal ou vertical, deve ser seguida da remoção de uma peça do adversário. As ações de remoção válidas devem obrigatoriamente dar preferência para peças do oponente que não estejam em uma trilha.

No processo de atualização das fases de jogo, o jogador que efetuou a jogada passa para a Fase 2 quando não houver mais peças em mãos. O jogador do próximo turno passa para a Fase 3 se o mesmo ficar com apenas três peças no tabuleiro.

O estado resultante é marcado como estado final se: o jogador do próximo turno não possuir ações válidas e assim perder o jogo; se o jogador do próximo turno ficar com menos de duas peças e assim também perder o jogo; se for detectado um loop com base no histórico das últimas três ações, resultando assim em um empate.

Na Figura 5.6 é mostrada uma atualização de estado com a ação do jogador de peça cor branca de movimento com formação de trilha, seguido da captura de uma peça do adversário.

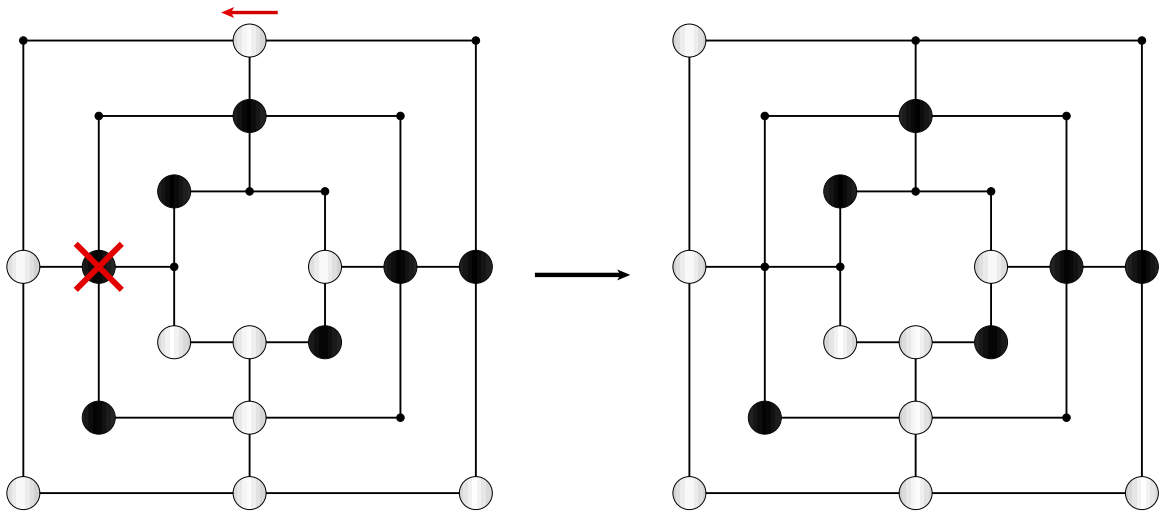


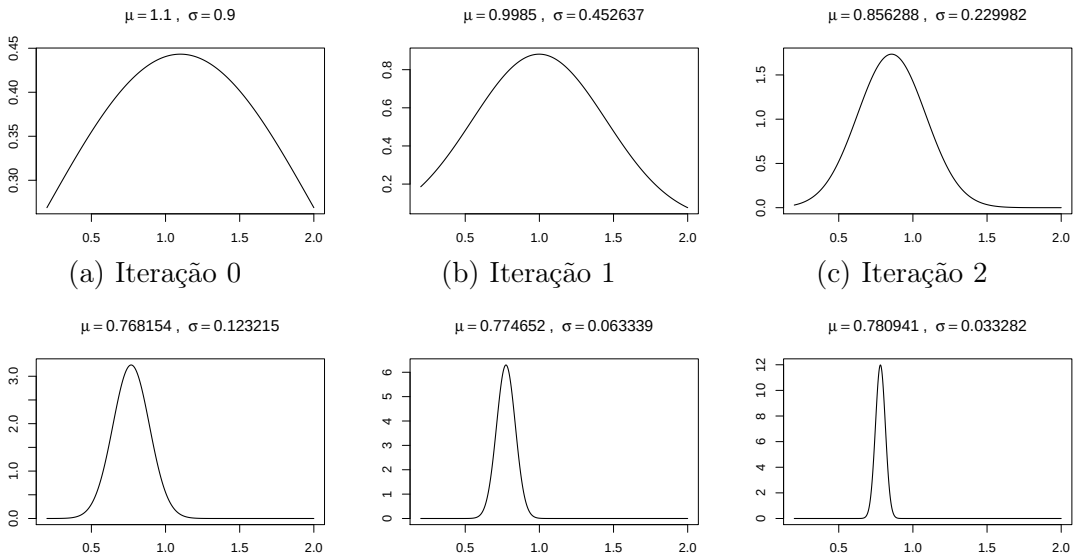
Figura 5.6: Representação de uma ação do jogador (W) no jogo Trilha.

5.2 Convergência no CEM da Constante UCT

Nesta seção é apresentado o comportamento da convergência da constante de exploração UCT usando o método CEM, explicado nas seções 2.3 e 4.5, para os dois jogos

descritos na seção anterior. A constante de exploração UCT deve estar dentro do intervalo [0,2; 2,0], e a distribuição inicial para o algoritmo CEM é uma distribuição normal com média $(0{,}2 + 2{,}0)/2 = 1{,}1$ e desvio padrão $(2{,}0 - 0{,}2)/2 = 0{,}9$, independente do jogo em questão. Isso garante altas probabilidades para todo o intervalo de possíveis valores, a fim de descobrir o ótimo global. Os parâmetros da fase CEM ficaram definidos da seguinte forma: tamanho da população igual a 15 exemplos; taxa de seleção de 0,134 para o conjunto elite de exemplos (o que leva a 2 exemplos no conjunto elite); número máximo de iterações de 5; *step-size* de 0,5; 90 repetições para a extração da taxa de vitória dos exemplos; 64 *threads* utilizadas; tempo de simulação dos agentes MCTSPlayer de 60 segundos.

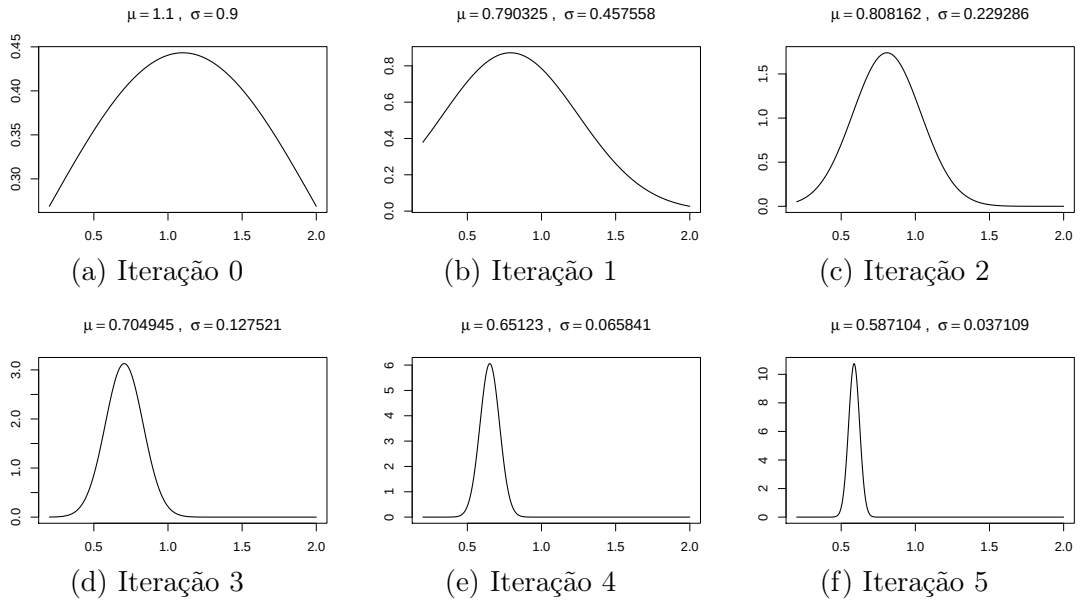
Considerando que o jogo Othello possui em média 60 turnos por partida, o tempo total da fase CEM foi de aproximadamente 4,4 dias, conforme discutido na seção 4.5. Já o jogo Trilha possui em média 50 turnos por partida, levando a um tempo total da fase CEM de aproximadamente 3,7 dias. Como se percebe, a fase CEM é muito custosa em termos de processamento, e por causa disso foram utilizados valores bem baixos para os parâmetros como população e tempo de simulação do algoritmo MCTS, pois foi escolhido que o foco de maior tempo de processamento seria dado à fase de geração de exemplos via MCTS. Mesmo com os baixos valores dos parâmetros o algoritmo conseguir convergir, como é mostrado na Figura 5.7 para o jogo Othello e na Figura 5.8 para o jogo Trilha.



(a) Iteração 0 (b) Iteração 1 (c) Iteração 2

(d) Iteração 3 (e) Iteração 4 (f) Iteração 5

Figura 5.7: Convergência do método CEM para o jogo Othello.



(a) Iteração 0 (b) Iteração 1 (c) Iteração 2

(d) Iteração 3 (e) Iteração 4 (f) Iteração 5

Figura 5.8: Convergência do método CEM para o jogo Trilha.

O jogo Othello convergiu para a constante de exploração UCT de valor 0,780941 e o jogo Trilha convergiu para a constante de exploração UCT de valor 0,587104, reforçando o fato de que jogos diferentes levam a diferentes constantes de exploração ótimas. Tais valores de constante de exploração foram utilizados em todas as fases posteriores do processo UCT-CCNN.

5.3 Geração de Exemplos via MCTS-UCT

Duas bases de dados de exemplos são geradas para cada jogo. Na primeira base de cada jogo o agente MCTSPlayer terá o tempo de simulação de 200 segundos e na segunda de 600 segundos. Quanto maior o tempo de simulação melhores serão os valores de utilidade estimados pelo algoritmo MCTS-UCT para os estados de jogo. Os parâmetros da fase de geração de exemplos estão definidos da seguinte forma: 30 partidas efetuadas a partir do estado inicial; 300 estados randômicos a serem gerados; 5 partidas efetuadas a partir de cada estado randômico gerado; 64 *threads* a serem utilizadas. A constante de exploração UCT utilizada foram as encontradas pelo método CEM.

Considerando que o jogo Othello possui em média 60 turnos por partida, o tempo total da fase de geração de exemplos foi de aproximadamente 3,32 dias para a base de dados com tempo de simulação de 200 segundos e 9,96 dias para a base de dados com tempo de simulação de 600 segundos, conforme discutido na Seção 4.6. O jogo Trilha possui em média 50 turnos por partida, o que levou a um tempo total da fase de

geração de exemplos de aproximadamente 2,77 dias para a base de dados com tempo de simulação de 200 segundos e 8,3 dias para a base de dados com tempo de simulação de 600 segundos. O ideal é aumentar esse tempo de simulação o máximo possível, conforme os recursos disponíveis.

5.4 Evolução do erro nas redes CCNN

Duas bases de dados de exemplos de estado-utilidade foram geradas para cada jogo. A partir dessas bases foram treinadas duas redes neurais para cada jogo. Na Tabela 5.3 é mostrado a quantidade de exemplos gerados para cada configuração, conforme as regras de filtragem de exemplos descritas na Seção 4.7, lembrando que em todas as configurações a quantidade de exemplos de teste foi definida para esta análise experimental como 10% dos exemplos filtrados, e o complemento de 90% para o conjunto de treino. Dessa forma no conjunto de treino não constam os exemplos do conjunto de teste, a fim de avaliar a evolução do erro na rede neural para exemplos nunca antes vistos.

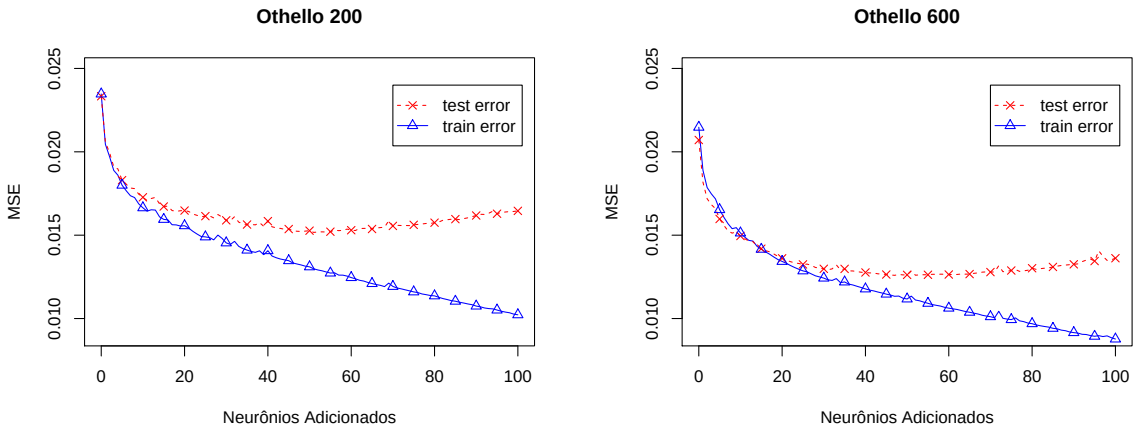
Jogo	**Tempo de Simulação MCTS**	**Exemplos de Treino**	**Exemplos de Teste**
Othello	200	149876	16653
Othello	600	144219	16024
Trilha	200	167360	18596
Trilha	600	258260	28696

Tabela 5.3: Quantidade de exemplos gerados por configuração.

A quantidade de exemplos é variável pois a quantidade de jogadas em uma partida e a quantidade de estados filhos válidos por jogada também são variáveis, sendo que os exemplos são extraídos dessas partidas, em cada jogada. Outra possibilidade seria definir o número de exemplos a serem gerados ao invés do número de partidas a serem executadas, mas por simplicidade decidiu-se pela segunda. Também para simplicidade, ficou determinado que as redes são treinadas até que o número de neurônios adicionados seja igual ao número de neurônios na camada de entrada, visto que nos experimentos realizados a rede começou a apresentar *overfitting* com uma quantidade inferior de neurônios. Durante o treino uma cópia da rede é salva a cada adição de neurônio e, no final, é escolhida aquela com menor MSE (*Mean Square Error*), que é a medida de precisão utilizada. Portanto, sempre quando for mencionado sobre o erro da rede, estará implícito que a medida de erro referenciada é o MSE. A rede neural treinada para o jogo Othello com exemplos limitados a 200 segundos de simulação é denominada **Othello:200** e com exemplos limitados a 600 segundos é denominada **Othello:600**.

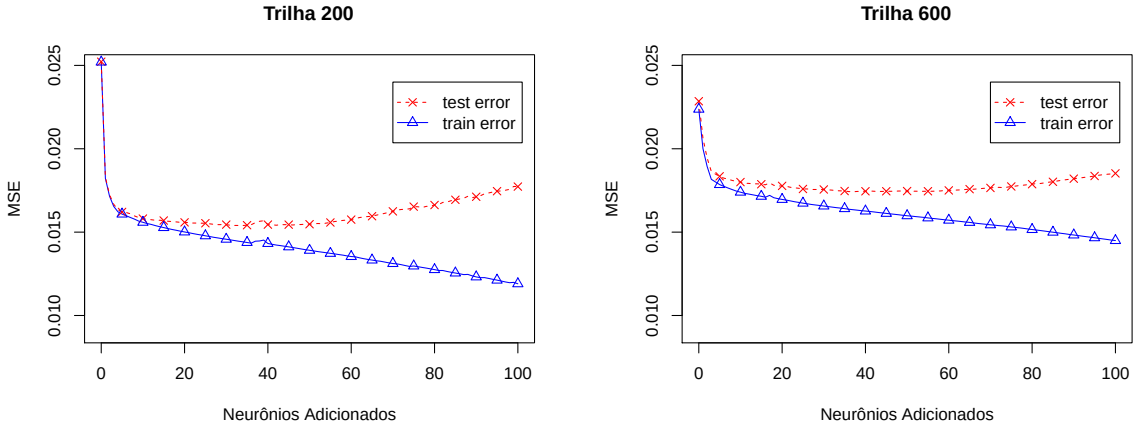
A rede neural treinada para o jogo Trilha com exemplos limitados a 200 segundos de simulação é denominada **Trilha:200** e com exemplos limitados a 600 segundos é denominada **Trilha:600**.

A seguir, na Figura 5.9 e na Figura 5.10, é mostrada a evolução do MSE no conjunto de treino e no conjunto de teste a cada adição de neurônio na rede CCNN, para cada configuração de treino, limitado até a adição do centésimo neurônio na rede.



(a) Evolução do erro na rede Othello:200. (b) Evolução do erro na rede Othello:600.

Figura 5.9: Evolução do erro das redes neurais para o jogo Othello.



(a) Evolução do erro na rede Trilha:200. (b) Evolução do erro na rede Trilha:600.

Figura 5.10: Evolução do erro das redes neurais para o jogo Trilha.

Na Tabela 5.4 estão listadas as redes neurais treinadas com o menor erro no conjunto de teste e a quantidade de neurônios adicionados.

Conforme esperado, o erro no treino sempre diminuiu, mas a adição de novos neurônios na rede, e consequentemente mais camadas escondidas, causou um *overfitting*

Rede	Neurônios Adicionados	MSE
Othello:200	51	0,015167
Othello:600	47	0,012592
Trilha:200	34	0,015397
Trilha:600	37	0,017419

Tabela 5.4: Redes treinadas com o menor erro no conjunto de teste.

nos dados do conjunto de treino, aumentando o erro no conjunto de testes a partir de um determinado número de neurônios adicionados.

Enquanto no jogo Othello o menor erro diminui da rede **Othello:200** para a rede **Othello:600**, no jogo Trilha ocorreu o contrário. Essa diferença pode ser influência das utilidades estimadas para os exemplos, da quantidade de exemplos gerados e da codificação binária de cada jogo. As redes **Othello:200** e **Trilha:200** possuem erros bem próximos (MSE de 0,015167 para o Othello e 0,015397 para o Trilha), assim como o número de exemplos usados no treino (149876 para o Othello e 167360 para o Trilha), mas o número de neurônios adicionados à rede é maior para o jogo Othello (51 para o Othello e 34 para o Trilha).

As redes **Othello:600** e **Trilha:600** possuem uma maior diferença dos erros (MSE de 0,012592 para o Othello e 0,017419 para o Trilha), o que pode estar relacionado com a quantidade de exemplos usados no treino, que é maior para o jogo Trilha (144219 para o Othello e 258260 para o Trilha). Os números de neurônios adicionados continuam próximos aos das redes anteriores (47 para o Othello e 37 para o Trilha).

No jogo Othello, não importando as ações dos jogadores, o jogo sempre terá no máximo 60 turnos, quando todo o tabuleiro é preenchido. Já o jogo Trilha, quanto mais fortes são os jogadores, mais ações defensivas são realizadas levando a um maior número de turnos até a finalização do jogo, o que pode explicar o maior número de exemplos gerados a partir das jogadas de agentes fortes (com maior tempo de simulação). Com a maior quantidade de exemplos de treino, melhor fica caracterizada a função que a rede neural tenta aproximar, o que para o jogo Trilha se mostrou ser uma função mais difícil de ser aprendida. O jogo Trilha apresenta quatro tipos diferentes de ações, incluindo movimentação de peças no tabuleiro, além de fases diferentes de jogo que mudam as regras das ações possíveis, o que pode explicar a maior dificuldade de aprender a função de valor. Apesar disso, não se pode afirmar que aprender Trilha é mais difícil que Othello, visto as diferenças na modelagem de *features* e na quantidade de exemplos usados no treino da rede.

5.5 Experimentos com agentes NeuralMinimax

Para avaliar a força das quatro funções de valor geradas foram executadas partidas entre dois agentes Minimax com poda Alfa-Beta, sendo que um jogador utiliza como função de avaliação a função de valor gerada pelo processo UCT-CCNN e o outro jogador utiliza uma função de avaliação desenvolvida especificamente para o jogo. No contexto desse trabalho, os agentes Minimax específicos utilizados foram desenvolvidos por alunos participantes de uma competição de agentes inteligentes aplicada como trabalho da disciplina de Inteligência Artificial ofertada pelo Departamento de Ciência da Computação da UFMG. Para o agente Othello, o trabalho escolhido foi de um aluno vencedor da competição no segundo semestre de 2014 com o agente Minimax denominado *Bothello*. Para o agente Trilha, o trabalho escolhido foi de um aluno, autor desta própria dissertação, que ficou em segundo lugar na competição no segundo semestre de 2015 com o agente Minimax denominado *J.A.R.V.I.S*, que perdeu do primeiro colocado por um critério de desempate com diferença de uma peça a mais quando perdeu em uma das duas partidas.

Para cada configuração do experimento foram executadas 100 partidas entre o agente NeuralMinimax e um dos agentes específicos. Ambos os oponentes *Bothello* e *J.A.R.V.I.S* possuem altura da árvore Minimax limitada até três níveis. Para avaliar diferentes estados de jogo e testar a generalização da função de valor, ficou decidido para o agente NeuralMinimax executar partidas utilizando de 3 a 7 níveis para o jogo Othello e 3 a 6 níveis para o jogo Trilha. Espera-se que quanto maior a altura na árvore de estados dos estados sendo avaliados, melhores serão as estimativas da rede neural, pois as redes foram treinadas a partir de exemplos de estado-utilidade gerados por um agente MCTSPlayer. No MCTS, quando um estado sendo avaliado está em uma altura maior na árvore de estados, a média do número de visitas nos nós serão maiores pois cada simulação é finalizada em um menor tempo, visto que as simulações percorrem um menor número de estados até atingir um estado de fim de jogo.

Além da variação na altura da árvore, também foi considerado na configuração dos experimentos se o agente NeuralMinimax inicia ou não a partida. Isso é importante porque em alguns jogos o jogador que inicia a partida pode começar com uma certa vantagem. Essa estratégia será mantida para o jogo Trilha, mesmo já sendo demostrado que em jogadas perfeitas o jogo sempre termina em empate [Gasser, 1996], pois a avaliação de estados distintos pode levar a diferentes resultados.

A seguir nas Tabelas 5.5 e 5.6 estão listadas as configurações de partidas realizadas para o jogo Othello, entre os agentes NeuralMinimax e *Bothello*, com os dados do agente

NeuralMinimax de quantidade de vitórias, derrotas, empates e o intervalo de confiança de 95% para a taxa de vitória.

Inicia a Partida	**Altura**	**Vitórias**	**Empates**	**Derrotas**	**IC 95% Taxa de vitória** (%)
Não	3	100	0	0	(96.4 ; 100)
Não	4	0	0	100	(0 ; 3.6)
Não	5	0	0	100	(0 ; 3.6)
Não	6	0	0	100	(0 ; 3.6)
Não	7	0	0	100	(0 ; 3.6)
Sim	3	0	0	100	(0 ; 3.6)
Sim	4	0	0	100	(0 ; 3.6)
Sim	5	0	0	100	(0 ; 3.6)
Sim	6	100	0	0	(96.4 ; 100)
Sim	7	0	0	100	(0 ; 3.6)

Tabela 5.5: Dados de partidas do agente NeuralMinimax Othello:200 contra *Bothello*.

Contra o agente Bothello, o agente NeuralMinimax **Othello:200** ganhou 100% das vezes em duas das dez configurações de partidas. A variação da altura máxima para o agente NeuralMinimax não ocasionou em melhorias nas taxas de vitórias como era esperado. O benefício de visualizar com antecedência um estado final na árvore de jogo, nos momentos finais da partida, não fez diferença na taxa de vitória. Isso ocorreu porque a árvore do jogo Othello tem em média uma altura de 60 níveis, e a vantagem de avaliar até 4 níveis a mais não foi suficiente para o agente NeuralMinimax, pois as ações executadas anteriormente na partida que determinaram o vencedor.

A variação no limite de altura possibilitou a avaliação da função de valor obtida em diferentes conjuntos de estados. Como das dez configurações o agente NeuralMinimax **Othello:200** ganhou todas as partidas em apenas duas configurações, podemos concluir que a função de valor gerada não é precisa, apesar do agente NeuralMinimax **Othello:200** ter ganhado do oponente *Bothello* com o mesmo limite de altura. Nas partidas com o agente NeuralMinimax **Othello:200** não houve nenhum empate e o comportamento foi absolutamente determinístico, com jogadores ganhando ou perdendo todas as partidas de uma configuração.

Já com o agente NeuralMinimax **Othello:600**, das dez configurações de partidas, ganhou 100% das vezes em três configurações, 57% em uma e 30% em outra. Com o aumento no tempo de simulação para a geração de exemplos da rede neural, a qualidade da função de valor aprendida aumentou, levando a um aumento no número de vitórias. Esse aumento de qualidade apresenta uma certa tendência para estados com maior profundidade na árvore de busca. O agente NeuralMinimax passou a perder na altura de três níveis pois a função de valor gerada ainda não é precisa, mesmo com a melhora

Inicia a Partida	**Altura**	**Vitórias**	**Empates**	**Derrotas**	**IC 95% Vitórias** (%)
Não	3	0	0	100	(0 ; 3.6)
Não	4	0	0	100	(0 ; 3.6)
Não	5	57	0	43	(46.7 ; 66.9)
Não	6	100	0	0	(96.4 ; 100)
Não	7	100	0	0	(96.4 ; 100)
Sim	3	0	0	100	(0 ; 3.6)
Sim	4	0	0	100	(0 ; 3.6)
Sim	5	0	0	100	(0 ; 3.6)
Sim	6	100	0	0	(96.4 ; 100)
Sim	7	30	0	70	(21.2 ; 40)

Tabela 5.6: Dados de partidas do agente NeuralMinimax Othello:600 contra *Bothello*.

da taxa de vitória para mais configurações de experimentos. Apesar disso, os resultados mostram que é possível melhorar a precisão da função de valor com o aumento do tempo de simulação para a geração de exemplos da rede neural.

A seguir nas Tabelas 5.7 e 5.8 estão listadas as configurações de partidas realizadas para o jogo Trilha, entre os agentes NeuralMinimax e *J.A.R.V.I.S*, com os dados do agente NeuralMinimax de quantidade de vitórias, derrotas, empates e o intervalo de confiança de 95% para a taxa de vitória.

Inicia a Partida	**Altura**	**Vitórias**	**Empates**	**Derrotas**	**IC 95% Vitórias** (%)
Não	3	45	0	55	(35 ; 55.3)
Não	4	0	17	83	(0 ; 3.6)
Não	5	0	15	85	(0 ; 3.6)
Não	6	6	1	93	(2.2 ; 12.6)
Sim	3	0	0	100	(0 ; 3.6)
Sim	4	5	11	84	(1.6 ; 11.3)
Sim	5	5	4	91	(1.6 ; 11.3)
Sim	6	0	0	100	(0 ; 3.6)

Tabela 5.7: Dados de partidas do agente NeuralMinimax Trilha:200 contra *J.A.R.V.I.S*.

Para as partidas do jogo Trilha também podemos observar a tendência de melhores taxas de vitória com o aumento no tempo de simulação para a geração de exemplos da rede neural. Apesar disso, a quantidade de vitórias foi bem menor em comparação ao jogo Othello. O agente NeuralMinimax **Trilha:200** ganhou 45% das vezes em seu melhor resultado, mas ganhou 100% das vezes na mesma configuração de partidas com a rede **Trilha:600**, com limite de altura Minimax de 3 e não iniciando a partida. No experimento **Trilha:600** de configuração com altura 5 e iniciando a partida, o agente NeuralMinimax não perdeu nenhuma partida, mas nessa mesma configuração houve

Inicia a Partida	**Altura**	**Vitórias**	**Empates**	**Derrotas**	**IC 95% Vitórias** (%)
Não	3	100	0	0	(96.4 ; 100)
Não	4	0	4	96	(0 ; 3.6)
Não	5	65	35	0	(54.8 ; 74.3)
Não	6	0	3	97	(0 ; 3.6)
Sim	3	0	0	100	(0 ; 3.6)
Sim	4	0	4	96	(0 ; 3.6)
Sim	5	0	1	99	(0 ; 3.6)
Sim	6	1	9	90	(0 ; 5.4)

Tabela 5.8: Dados de partidas do agente NeuralMinimax Trilha:600 contra *J.A.R.V.I.S.*

35 empates, levando a uma taxa de vitória de 65% para o agente NeuralMinimax. Novamente se pode notar que o jogo Trilha aparenta ser mais difícil de aprender que o jogo Othello, conforme se observou no treinamento das redes neurais, o que refletiu nos experimentos com a utilização das funções de valor obtidas. Outra característica é a maior tendência a empates no jogo Trilha, sendo que não houve nenhum empate para o jogo Othello.

Vale ressaltar que as taxas de vitória são em sua maioria 100% ou 0% ou próximo desses valores devido ao comportamento determinístico do algoritmo Minimax. Aleatoriedades somente ocorrem quando a função de avaliação retorna o mesmo valor para mais de um estado, então dentre esses estados um é escolhido aleatoriamente de maneira uniforme. Raramente a função de valor representada pela rede neural retorna o mesmo valor para estados diferentes e a frequência desses valores repetidos é maior para estados próximos do fim de jogo.

Com esse resultado podemos dizer que as funções de valor obtidas não são precisas, no sentido de preverem o resultado verdadeiro de jogo em uma politica perfeita, mas de maneira nenhuma são ruins, pois os agentes NeuralMinimax foram capazes de ganhar em determinadas situações, mesmo sendo agentes gerados sem nenhum conhecimento específico de domínio. Observando com mais detalhes os valores de utilidade atribuídos pelas redes neurais aos estados ao longo das partidas, percebe-se que muitos estados possuem utilidades bem próximas. Pode-se concluir que a função de valor obtida não foca no melhor estado cuja ação deve ser executada, mas sim "aponta" para os estados cujas ações associadas serão provavelmente boas. Cabe mencionar que o mesmo ocorre com a rede neural treinada a partir de jogadas reais de profissionais do jogo Go no Algoritmo AlphaGo, explicado na Seção 3.1.

No artigo que descreve o algoritmo AlphaGo [Silver et al., 2016], foi dito que agentes baseados em redes neurais treinadas a partir de jogadas profissionais de GO chegaram a ganhar apenas 11% das vezes contra agentes específicos, que utilizam de

conhecimento específico de domínio. A diferença é que enquanto nesta dissertação foram utilizados cerca de 150 mil exemplos gerados por simulações Monte Carlo, no AlphaGo foram utilizados cerca de 30 milhões de exemplos de partidas jogadas entre profissionais. Além disso, a rede neural utilizada no AlphaGo é uma Rede Neural Convolucional Profunda, com grande capacidade de detecção de *features*, enquanto neste trabalho foram utilizadas redes neurais de correlação em cascata, as quais dependem de uma boa engenharia de *features* realizada manualmente. Observando por esse lado, os resultados obtidos foram bem satisfatórios, dadas as possíveis limitações. Vale mencionar que neste trabalho a rede neural foi utilizada como uma função de avaliação do algoritmo Minimax com Poda Alfa-Beta, o que pode ter ajudado nos resultados obtidos.

Nos experimentos realizados, pode-se questionar que os agentes específicos escolhidos não são reconhecidos como agentes fortes e que não podem ser utilizados para generalizar a força dos agentes gerados pelo método UCT-CCNN. Apesar dessa afirmação estar correta, a ideia desses experimentos era mostrar a capacidade das funções de valores geradas frente a funções de avaliação implementadas por humanos com conhecimento específico de domínio. Na maioria dos trabalhos da literatura, os agentes genéricos são comparados contra agentes MCTS-UCT. Uma comparação desse tipo será discutida na próxima seção.

5.6 Experimentos com agentes NeuralMCTS

Com inspiração no algoritmo AlphaGo, explicado na Seção 3.1, decidiu-se aplicar na política da árvore do algoritmo MCTS a função de valor obtida, a fim de aproveitar a característica da rede de indicar os estados filhos provavelmente melhores e assim redirecionar o crescimento da árvore na direção desses estados. Essa estratégia foi implementada no agente do tipo NeuralMCTS, explicado na Seção 4.4.

No algoritmo UCT padrão, estados presentes na árvore MCTS são selecionados utilizando uma política que leva em consideração os dados estatísticos obtidos até o momento. Por causa disso, o que pode ocorrer é que o tempo de simulação MCTS pode ser insuficiente para coletar a quantidade de dados necessária para indicar a melhor ação a ser realizada. Na tentativa de contornar esse problema, no agente NeuralMCTS a política da árvore incorpora a rede neural treinada para avaliar os estados de jogo e assim aumentar o bônus de exploração para os estados provavelmente melhores. Essa alteração na política da árvore irá enviesar o crescimento da árvore MCTS na direção de um conjunto de estados provavelmente melhores, o que poderá ajudar no problema

da limitação do tempo total de simulação. Vale mencionar que a política padrão, na fase de simulação, continua sendo aleatória e uniformemente distribuída para o agente NeuralMCTS.

Essa estratégia foi implementada no agente do tipo NeuralMCTS e experimentos foram realizados contra agentes MCTS-UCT, com o número de simulações MCTS limitado para todos os agentes em 5000 simulações. Para os agentes NeuralMCTS foram utilizadas as redes neurais **Othello:600** e **Trilha:600**. Foram executados quatro configurações de 200 partidas e para cada configuração são listados na Tabela 5.9 a quantidade de vitórias, derrotas, empates e o intervalo de confiança de 95% da taxa de vitória do agente principal NeuralMCTS.

Jogo	**Inicia a Partida**	**Vitórias**	**Empates**	**Derrotas**	**IC 95% Vitória** (%)
Othello	Não	156	6	38	78 (71.6 ; 83.5)
Othello	Sim	154	3	43	77 (70.5 ; 82.6)
Trilha	Não	60	61	79	30 (23.7 ; 36.9)
Trilha	Sim	31	50	119	15.5 (10.8 ; 21.3)

Tabela 5.9: Partidas NeuralMCTS com as redes Othello:600 e Trilha:600.

A ideia é que as funções de valor representadas pelas redes **Othello:600** e **Trilha:600** indicam um espectro de estados interessantes a serem investigados, e o agente NeuralMCTS utiliza essas funções para enviesar o crescimento da árvore na direção desses estados. O agente NeuralMCTS ganhou em média 78% das vezes em que jogou contra o MCTS-UCT para o jogo Othello, sendo significativamente melhor do que o algoritmo padrão UCT. Já para o jogo Trilha o agente NeuralMCTS ganhou apenas 30% das vezes em que não iniciou a partida contra o MCTS-UCT e 15.5% das vezes em que iniciou a partida contra o MCTS-UCT.

Diante desses resultados podemos concluir que a função de valor representada pela rede **Othello:600** possui uma acurácia satisfatória a ponto de elevar a taxa de vitórias ao enviesar o crescimento da árvore em direção a estados sem nenhuma avaliação estatística, apenas com o uso da rede neural, o que mostra a qualidade da função de valor obtida. Os exemplos utilizados no treino da rede **Trilha:600** aparentemente não possuem a precisão necessária para se obter bons resultados, sendo que uma solução poderia ser aumentar o tempo de simulação para a geração de exemplos da rede neural ou ajustar a codificação binária dos estados do jogo Trilha a fim de melhorar a engenharia de *features*. Essa é apenas uma constatação inicial, visto que a estratégia

de focar o crescimento da árvore na direção de estados interessantes como consequência da alteração na política da árvore pode não funcionar para qualquer tipo de jogo.

Melhorar a política da árvore não é garantia de sucesso para o algoritmo MCTS. A política padrão desempenha o papel mais importante no MCTS pois a mesma é utilizada para controlar as simulações que são executadas a partir do estado expandido. A política da árvore apenas indica qual estado será expandido. Depois disso, é de responsabilidade da política padrão avaliar o estado na fase de simulação, conforme explicado na Seção 2.1. A forma mais comum da política padrão seleciona ações aleatoriamente até que um estado final seja alcançado. Melhorias na política padrão podem ajudar a melhorar ainda mais os resultados, e essa pode ser uma das razões pela qual o agente NeuralMCTS não obteve bons resultados no jogo Trilha.

A estratégia implementada no agente NeuralMCTS, explicado na Seção 4.4, é inspirada na estratégia adotada pelo algoritmo AlphaGo. Tanto no jogo Go como no Othello, as possíveis ações são de colocação de peças apenas, mas as regras de colocação, captura e tamanho de tabuleiro são diferentes. Já no jogo Trilha, além da ação de colocação de peças, existem ações de movimentação e remoção e diferentes fases de jogo. Identificar propriedades de jogos que caracterizam melhor performance para melhorias específicas no MCTS é uma tarefa difícil. Em algumas situações a seleção de nós da árvore, seja na política da árvore ou na política padrão, usando uma distribuição uniforme pode ser melhor do que aplicar conhecimento específico de domínio para enviesar a seleção [James et al., 2017].

Capítulo 6

Conclusão

Nesta dissertação introduzimos o processo UCT-CCNN como uma estratégia de *general game playing* para se obter agentes inteligentes para jogos genéricos de tabuleiro com dois jogadores, de soma zero, de informação perfeita, determinísticos, discretos e sequenciais, capazes de ganhar de agentes que se utilizam de conhecimento específico de domínio.

Para alcançar esse objetivo o método UCT-CCNN se utiliza do algoritmo *Monte Carlo Tree Search* (MCTS) com a política da árvore *Upper Confidence Bounds for Tree* (UCT) para gerar uma base de dados de exemplos de estado-utilidade. A partir desses exemplos uma rede neural *Cascade Correlation Neural Network* (CCNN) é treinada gerando assim uma função de valor para avaliar qualquer estado de jogo. A função de valor gerada pode ser facilmente integrada a qualquer algoritmo, como o Minimax com Poda Alfa-Beta e o próprio MCTS.

A base de dados de exemplos de estado-utilidade é gerada a partir de múltiplas partidas executadas entre agentes do tipo MCTS. Os exemplos são filtrados para selecionar apenas aqueles gerados por jogadores vencedores, e dentre os exemplos repetidos são considerados aqueles cujo estado foi visitado o maior número de vezes pelas simulações MCTS.

A constante de exploração UCT dos agentes MCTS utilizados na geração de exemplos foi obtida a partir de um método de otimização estocástica denominado *Cross Entropy Method* (CEM). Essa otimização é necessária visto que cada jogo exige um valor de constante de exploração diferente e não existe uma maneira óbvia de se definir esse valor.

As redes CCNN são utilizadas porque representam uma categoria de redes neurais construtivas capazes de se adaptar ao problema que são submetidas. Isso favorece o aspecto GGP do método UCT-CCNN, evitando-se assim o uso de parâmetros para

especificar a arquitetura da rede, como a quantidade de camadas escondidas e o número de neurônios em cada uma delas.

Conforme mostrado no Capítulo 5, agentes que utilizam funções de valor geradas pelo método UCT-CCNN foram capazes de ganhar de agentes específicos em algumas situações. A precisão da função de valor obtida está atrelada à precisão das utilidades estimadas para os exemplos usados no treino da rede neural. Quanto maior o tempo de simulação MCTS melhor será a precisão das utilidades estimadas. Além disso foi mostrado que a função de valor pode ser utilizada em uma política da árvore para guiar a busca MCTS, e com isso foi capaz de melhorar a performance em comparação à estratégia padrão UCT com o mesmo número de simulações para o jogo Othello. Já para o jogo Trilha a performance do agente piorou, indicando que essa estratégia não é indicada para qualquer jogo, ou que o método precisa de melhorias como parametrizar a influência da rede neural na política da árvore.

Uma importante observação é sobre o controle no tempo de simulação MCTS para a geração de exemplos a serem utilizados no treino da rede neural. Esse parâmetro garante ao processo UCT-CCNN uma flexibilidade no controle da força do agente a ser gerado. Nos experimentos realizados o aumento no tempo de simulação de 200 para 600 segundos aumentou consideravelmente a força do agente gerado para o jogo Othello. Para o jogo Trilha a melhora também foi observada, mas ela foi mais amena. De uma maneira geral, o método UCT-CCNN teve uma melhor performance para o jogo Othello do que para o jogo Trilha, indicando que tal método pode se sair melhor em jogos com determinadas características.

Outra observação é que a função de valor gerada para os jogos pode ser usada como uma política da árvore MCTS, por indicar um espectro de estados provavelmente melhores. Essa estratégia melhorou significativamente o agente MCTS em relação ao algoritmo UCT padrão para o jogo Othello, mas em contra partida piorou em relação ao UCT para o jogo Trilha, o que mostra que o uso de uma função de valor na política da árvore não é recomendado para qualquer jogo. Esse comportamento de melhorias no MCTS já é observado em outros trabalhos, onde certas melhorias funcionam bem em alguns jogos e em outros não.

Vale notar que a função de valor utilizada nos agentes gerados se beneficia de um conjunto de *features* completo do jogo, o que normalmente não é possível em melhorias convencionais no algoritmo MCTS, como RAVE e FAST, que são descritas na Seção 3.2. Outra característica do processo UCT-CCNN é que o agente gerado se beneficia da função de valor logo nas primeiras decisões da partida, o que só é possível graças ao processo de geração *off-line*. Para os agentes de GGP geralmente é necessário acumular simulações para estimar valores de utilidade para as *features*, muitas vezes

bastante simplificadas. O que o UCT-CCNN perde em relação a esses agentes de GGP é a necessidade da execução de um processo *off-line* para geração da função de valor do agente.

Além disso o UCT-CCNN possui mais uma característica importante de gerar uma função de valor genérica, representada por uma rede neural, podendo ser usada em conjunto com qualquer outro algoritmo. Nessa dissertação usamos a função de valor gerada com os algoritmos Minimax e MCTS.

A principal característica do UCT-CCNN é sem dúvida o foco em *general game playing*, definindo um método que não é restrito a um jogo em específico, com o uso de ferramentas poderosas como MCTS-UCT e redes neurais construtivas, que é o caso das redes CCNN. O processo UCT-CCNN é um importante passo na direção para geração de agentes inteligentes fortes, com um processo completamente automatizado, possibilitando a criadores de jogos a geração de agentes inteligentes, com dificuldade variável, sem a necessidade de implementação manual de código de inteligência artificial, o que muitas vezes é um grande desafio.

6.1 Trabalhos Futuros

Conforme mostrado no Capítulo 5, as funções de valores obtidas não fornecem uma avaliação perfeita próxima da política ótima mas garantem um bom espectro de estados provavelmente melhores. Isso influenciou o uso das funções de valor na política da árvore do algoritmo MCTS. Infelizmente a política padrão do MCTS é a política principal e é responsável pela maior parte da força do agente MCTS. Uma função de valor boa para a política padrão deve ser idealmente próxima da política ótima. No algoritmo AlphaGo esse problema foi solucionado com o uso de aprendizado por reforço para melhorar a acurácia da rede neural obtida. Uma solução parecida poderia ser adotada para melhorar a acurácia da rede CCNN obtida pelo processo UCT-CCNN.

Aprendizado por reforço com redes CCNN foi introduzido por Nissen [2007] e pode ser uma estratégia adotada para gerar uma função de valor boa como política padrão, que junto com a estratégia adotada no agente NeuralMCTS na política da árvore pode resultar em agentes muito mais fortes.

Outra possível melhoria no método UCT-CCNN está na filtragem dos exemplos MCTS, onde poderiam ser descartados exemplos com pequeno número de simulações, a fim de melhorar a qualidade estatística dos exemplos utilizados para o treino da rede neural. Provavelmente essa estratégia iria reduzir o número de exemplos gerados, exigindo uma maior quantidade de partidas efetuadas para a geração de exemplos. Uma

outra estratégia poderia ser forçar um mínimo de simulações para todos os estados, mas isso poderia afetar o comportamento do algoritmo MCTS.

Outra ideia seria usar Redes de Aprendizado Profundo para gerar as funções de valores. O principal problema é que essas redes necessitam de bastante cuidado na especificação da arquitetura a ser utilizada. Isso poderia dificultar a característica *general game playing* do método UCT-CCNN, resultando em parâmetros adicionais difíceis de se definir pelo usuário.

No contexto de jogos suportados, o universo de jogos poderia ser expandido ao se adaptar o algoritmo MCTS utilizado na geração de exemplos para jogos com mais de dois jogadores, estocásticos e parcialmente observáveis. Para jogos com mais de dois jogadores e estocásticos o método que combina UCB e MAC (*Maintaining Arc Consistency*), introduzido por Koriche et al. [2016], poderia ser uma alterativa promissora ao clássico MCTS-UCT.

Uma melhoria direta mais interessante seria receber como entrada a descrição GDL dos jogos [Love et al., 2008]. Para isso deve ser criado um método de geração de implementações das *interfaces* de descrição de jogo, gerando tais classes a partir da descrição GDL dos jogos. Obviamente no estado atual deste trabalho os jogos GDL recebidos como entrada deveriam respeitar as restrições de jogos genéricos de tabuleiro com dois jogadores, de soma zero, de informação perfeita, determinísticos, discretos e sequenciais.

Referências Bibliográficas

Auer, P.; Cesa-Bianchi, N. & Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235--256.

Balázs, G. (2009). Cascade-correlation neural networks: A survey. *Department of Computing Science, University of Alberta, Edmonton, Canada*, pp. 1--6.

Bellemare, M. G.; Naddaf, Y.; Veness, J. & Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253--279.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; Colton, S. et al. (2012). A survey of Monte-Carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1--43.

Chaslot, G.; Winands, M.; Uiterwijk, J.; Van Den Herik, H. & Bouzy, B. (2007). Progressive strategies for Monte-Carlo tree search. Em *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)*, pp. 655--661.

Chaslot, G.; Winands, M. H. M.; Szita, I. & van den Herik, H. J. (2008). Cross-entropy for Monte-Carlo tree search. *ICGA Journal*, 31(3):145--156.

de Mello, T. H. & Rubinstein, R. Y. (2002). Estimation of rare event probabilities using cross-entropy. Em *Proceedings of the Winter Simulation Conference*, volume 1, pp. 310–319 vol.1.

Ebner, M.; Levine, J.; Lucas, S. M.; Schaul, T.; Thompson, T. & Togelius, J. (2013). Towards a video game description language. Em Lucas, S. M.; Mateas, M.; Preuss, M.; Spronck, P. & Togelius, J., editores, *Artificial and Computational Intelligence in Games*, volume 6 of *Dagstuhl Follow-Ups*, pp. 85--100. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.

Fahlman, S. E. (1989). Faster-learning variations on back-propagation: An empirical study. Em Touretzky, D. S.; Hinton, G. E. & Sejnowski, T. J., editores, *Proceedings of the 1988 Connectionist Models Summer School*, pp. 38--51. San Francisco, CA: Morgan Kaufmann.

Fahlman, S. E. & Lebiere, C. (1990). Advances in neural information processing systems 2. capítulo The Cascade-correlation Learning Architecture, pp. 524--532. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Finnsson, H. (2012). *Simulation-Based General Game Playing*. Tese de doutorado, Reykjavik University.

Gasser, R. (1996). Solving nine men's morris. *Computational Intelligence*, 12(1):24--41.

Gelly, S. & Silver, D. (2007). Combining online and offline knowledge in UCT. Em *Proceedings of the 24th international conference on Machine learning*, pp. 273--280. ACM.

Gelly, S. & Wang, Y. (2006). Exploration exploitation in Go: UCT for Monte-Carlo Go. Em *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*, Canada.

Genesereth, M.; Love, N. & Pell, B. (2005). General game playing: Overview of the AAAI competition. *AI magazine*, 26(2):62.

Goertzel, B. & Pennachin, C. (2007). *Artificial general intelligence*, volume 2. Springer.

Hausknecht, M.; Lehman, J.; Miikkulainen, R. & Stone, P. (2014). A neuroevolution approach to general atari game playing. *Computational Intelligence and AI in Games, IEEE Transactions on*, 6(4):355--366.

Igel, C. & Hüsken, M. (2000). Improving the Rprop learning algorithm. Em *Proceedings of the second international ICSC symposium on neural computation (NC 2000)*, volume 2000, pp. 115--121. Citeseer.

James, S.; Konidaris, G. & Rosman, B. (2017). An analysis of Monte-Carlo tree search. *AAAI Conference on Artificial Intelligence*.

Knuth, D. E. & Moore, R. W. (1975). An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293--326.

Kocsis, L. & Szepesvári, C. (2006). Bandit based Monte-Carlo planning. Em *European conference on machine learning*, pp. 282--293. Springer.

Kocsis, L.; Szepesvári, C. & Willemson, J. (2006). Improved Monte-Carlo search. *Univ. Tartu, Estonia, Tech. Rep*, 1.

Koriche, F.; Lagrue, S.; Piette, É. & Tabary, S. (2016). General game playing with stochastic CSP. *Constraints*, 21(1):95--114.

Kullback, S. (1959). Statistics and information theory. *J. Wiley and Sons, New York*.

Liang, Y.; Machado, M. C.; Talvitie, E. & Bowling, M. (2016). State of the art control of atari games using shallow reinforcement learning. Em *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, pp. 485--493. International Foundation for Autonomous Agents and Multiagent Systems.

Love, N.; Hinrichs, T.; Haley, D.; Schkufza, E. & Genesereth, M. (2008). General game playing: Game description language specification.

Nissen, S. (2003). Implementation of a fast artificial neural network library (FANN). *Report, Department of Computer Science University of Copenhagen (DIKU)*, 31:29.

Nissen, S. (2007). Large scale reinforcement learning using Q-Sarsa(λ) and cascading neural networks. Dissertação de mestrado, Department of Computer Science, University of Copenhagen.

Nissen, S. & Nemerson, E. (2000). Fast artificial neural network library. *Available at leenissen. dk/fann/html/files/fann-h. html*.

Perez-Liebana, D.; Samothrakis, S.; Togelius, J.; Schaul, T. & Lucas, S. M. (2016). General video game ai: Competition, challenges and opportunities. Em *Thirtieth AAAI Conference on Artificial Intelligence*.

Riedmiller, M. & Braun, H. (1993). A direct adaptive method for faster backpropagation learning: The Rprop algorithm. Em *Neural Networks, 1993., IEEE International Conference on*, pp. 586--591. IEEE.

Rubinstein, R. Y. & Kroese, D. P. (2013). *The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation and machine learning*. Springer Science & Business Media.

Schaul, T. (2013). A video game description language for model-based or interactive learning. Em *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pp. 1--8. IEEE.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M. et al. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484--489.

van Hasselt, H.; Guez, A. & Silver, D. (2016). Deep Reinforcement Learning with Double Q-Learning. Em *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pp. 2094--2100.

Yannakakis, G. & Togelius, J. (2015). A panorama of artificial and computational intelligence in games. *Computational Intelligence and AI in Games, IEEE Transactions on*, 7(4):317–335.

Apêndice A

Treinando redes CCNN com a biblioteca FANN

Neste apêndice mostramos a implementação de um código C++ para o treinamento de uma rede do tipo CCNN. No Código A.1 estão listados os comandos básicos necessários para o treinamento da rede. Esses comandos representam uma porção do código utilizado no treinamento das redes CCNN no método UCT-CCNN, introduzido nesta dissertação.

```
#include "floatfann.h"
#include "fann_cpp.h"

FANN::neural_net net;
FANN::training_data train_data;

train_data.read_train_from_file(train_file);
net.create_shortcut(2, train_data.num_input_train_data(),
        train_data.num_output_train_data());

net.set_training_algorithm(
        FANN::training_algorithm_enum::TRAIN_RPROP);
net.set_activation_function_hidden(
        FANN::activation_function_enum::SIGMOID_SYMMETRIC);
net.set_activation_function_output(
        FANN::activation_function_enum::SIGMOID_SYMMETRIC);
net.set_train_error_function(
        FANN::error_function_enum::ERRORFUNC_LINEAR);

net.cascadetrain_on_data(
        train_data, max_neurons,
        neurons_between_reports,
        desired_error
);
```

Código A.1: Treinando rede CCNN com a biblioteca FANN

No Código A.1, a variável **max_neurons** é referente ao número máximo de neurônios a serem adicionados à rede. Esse número pode ser definido como um parâmetro do objeto *GeneralAgentTrainingConfig*, mas por padrão é definido como o número total de neurônios na camada de entrada. A variável **neurons_between_reports** representa a quantidade de neurônios a ser adicionada até que a função *callback* do treino seja chamada. Esse valor é definido como 1 para que a cada adição de neurônio a função *callback* seja chamada e se obtenha o erro no treino e no teste da rede naquela iteração. A variável **desired_error** representa o erro desejado para o qual o treinamento será interrompido. O valor do erro desejado é por padrão setado com um valor baixo (0.005). O que determinará a rede a ser utilizada é o erro de teste obtido na função de *callback*, o qual começa a divergir antes mesmo que o erro mínimo fosse alcançado nos testes executados, os quais serão apresentados no capítulo 5. **SIGMOID_SYMMETRIC** se refere à função de ativação tangente hiperbólica. Essa função foi utilizada para garantir o intervalo de valores [−1,1], apesar de ser recomendado utilizar uma função de ativação linear na camada de saída de uma rede Cascade 2 [Nissen, 2007]. **TRAIN_RPROP** se refere ao algoritmo de treino iRPROP. **ERRORFUNC_LINEAR** se refere a uma função de erro padrão linear, indicada pela própria biblioteca para o treinamento de redes CCNN.