UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
INSTITUTO TÉRCIO PACITTI DE APLICAÇÕES E PESQUISAS COMPUTACIONAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

BRUNO BOTTINO FERREIRA

APRENDIZAGEM COM LÓGICA NEBULOSA USANDO GPU COMPUTING APLICADA A JOGOS

Rio de Janeiro
2013

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
INSTITUTO TÉRCIO PACITTI DE APLICAÇÕES E PESQUISAS COMPUTACIONAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

BRUNO BOTTINO FERREIRA

APRENDIZAGEM COM LÓGICA NEBULOSA USANDO GPU COMPUTING APLICADA A JOGOS

Dissertação de Mestrado submetida ao Corpo Docente do Departamento de Ciência da Computação do Instituto de Matemática, e Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários para obtenção do título de Mestre em Informática.

Orientador: Adriano Joaquim de Oliveira Cruz

Rio de Janeiro
2013

F383 Ferreira, Bruno Bottino

Aprendizagem com lógica nebulosa usando GPU computing aplicada a jogos / Bruno Bottino Ferreira. – 2013.
103 f.: il.

Dissertação (Mestrado em Informática) – Universidade Federal do Rio de Janeiro, Instituto de Matemática, Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Programa de Pós-Graduação em Informática, Rio de Janeiro, 2013.

Orientador: Adriano Joaquim de Oliveira Cruz.

.

1. Lógica Nebulosa. 2. GPU Computing. 3. Jogos Eletrônicos. – Teses. I. Cruz, Adriano Joaquim de Oliveira (Orient.). II. Universidade Federal do Rio de Janeiro, Instituto de Matemática, Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Programa de Pós-Graduação em Informática. III. Título

CDD

BRUNO BOTTINO FERREIRA

Aprendizagem com lógica nebulosa usando GPU computing aplicada a jogos

Dissertação de Mestrado submetida ao Corpo Docente do Departamento de Ciência da Computação do Instituto de Matemática, e Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários para obtenção do título de Mestre em Informática.

Aprovado em: Rio de Janeiro, _____ de _____ de _____.

_____

Ph.D. Adriano Joaquim de Oliveira Cruz (Orientador)

_____

Josefino Cabral Melo Lima, Docteur

_____

Silvana Rossetto, D.Sc.

_____

Esteban Walter Gonzales Clua, D.Sc.

Rio de Janeiro
2013

AGRADECIMENTOS

Agradeço primeiramente ao meu orientador, Adriano, pela paciência e liberdade sempre dispensadas a seus alunos em seus estudos.

À minha família, meus pais e irmãos, e minha namorada, Débora, por sempre acreditarem em mim e darem todo o apoio para que corresse atrás dos meus sonhos.

Aos amigos que sempre estiveram próximos e continuaram próximos mesmo quando distantes. Vocês sabem quem são.

Aos colegas da Aquiris Game Studio pelo apoio na reta final, e a todos que ajudam a tornar mais divertida a louca vida de desenvolvedor de jogos no Brasil.

Finalmente, agradecimentos especiais a Thiago Gomes e Lucas Ribeiro pela ajuda na execução de testes do método desenvolvido em diferentes máquinas do Laboratório de Métodos Numéricos do PPGI.

RESUMO

Ferreira, Bruno Bottino. **Aprendizagem com lógica nebulosa usando GPU computing aplicada a jogos**. 2013. 103 f. Dissertação (Mestrado em Informática) - PPGI, Instituto de Matemática, Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2013.

Este trabalho foca na busca por maneiras novas e mais eficientes de implementar agentes dinâmicos em jogos eletrônicos através de Inteligência Artificial e Computacional, com ênfase em Lógica Nebulosa. A utilização de sistemas nebulosos do tipo TSK é estudada no domínio dos jogos eletrônicos, juntamente com métodos para acelerar o treinamento de tais sistemas com o uso de técnicas modernas de GPU Computing. O trabalho descreve uma nova técnica para treinamento de sistemas nebulosos TSK em uma arquitetura massivamente paralela e sua implementação na plataforma CUDA, da nVidia. Isto é seguido pela definição de modelos de agentes dinâmicos para dois jogos diferentes, implementados com aprendizado de regras em tempo real num sistema TSK baseado em entradas de um jogador humano e os resultados de suas aplicações.

Palavras-chave: Lógica Nebulosa, GPU Computing, Jogos Eletrônicos.

ABSTRACT

Ferreira, Bruno Bottino. **Aprendizagem com lógica nebulosa usando GPU computing aplicada a jogos**. 2013. 103 f. Dissertação (Mestrado em Informática) - PPGI, Instituto de Matemática, Instituto Tércio Pacitti, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2013.

This work focuses on the search for novel and faster methods of implementing dynamic agents in video games based on Artificial and Computational Intelligence, with an emphasis in Fuzzy Logic. The use of Fuzzy TSK systems is studied in the domain of video games, as well as methods for speeding up training of such systems using modern techniques of GPU Computing. The work describes a new techniqe for training of TSK Fuzzy Systems in a massively parallel architeture and its implementation in the CUDA platform. This is followed by the definition of models for dynamic agents in two different games, implemented with real-time learning of rules in a TSK system based on inputs from a human player and the results of such applications.

Keywords: Fuzzy Logic, GPU Computing, CUDA, Video Games, Adaptive Systems.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

IA	Inteligência Artificial
RNA	Rede Neural Artificial
CPU	Central Processing Unit
GPU	Graphics Processing Unit
CUDA	Compute Unified Device Architecture
BLS	Batch Least Squares
RLS	Recursive Least Squares
TSK	Takagi-Sugeno-Kang
FIS	Fuzzy Inference System
ANFIS	Adaptive Neuro Fuzzy Inference System
SONFIN	Self-Constructing Neural Fuzzy Inference Network

SUMÁRIO

1 INTRODUÇÃO

1.1 Inteligência Artificial

A Inteligência Artificial (IA) é um assunto que sempre fascinou o homem. Desde a criação das primeiras máquinas automáticas, o imaginário humano é permeado pela ideia de máquinas que "pensam" e "agem" sozinhas, e a mídia escrita e audiovisual explorou exaustivamente este tema em inúmeros trabalhos criativos.

Já o campo da Inteligência Artificial, enquanto assunto acadêmico e técnico, localiza-se deveras distante da ideia romântica, muito propagada pelos livros e filmes, de máquinas auto-conscientes que tomam decisões por si mesmas, criando o caos e em muitos casos dominando o mundo.

RUSSELL; NORVIG (2003) sugere que existem variadas formas de definir, ou separar a IA, considerando duas abordagens: um eixo que separa *pensamento e raciocínio* de *comportamento*, e outro que separa a busca pelo desempenho *humano* em oposição à busca por uma solução *ideal* ou *racional* para um problema.

Todos os conceitos de IA advindos desta classificação são válidos, e geram campos de pesquisa diferentes, pois atingem a objetivos diferentes. Dentro da busca por soluções ótimas, por exemplo, podemos citar controladores robóticos (*comportamento*) e sistemas de inferência lógica como o Prolog (*pensamento e raciocínio*). Por outro lado, considerando-se a busca por comportamentos humanos, destacam-se pesquisas que tentam entender a fundo o funcionamento do pensamento humano (*pensamento e raciocínio*) e a criação de agentes autônomos com características hu-

manoides para uso, por exemplo, em filmes, para simulação de grandes batalhas ou multidões (*comportamento*).

1.1.1 Inteligência Artificial e Inteligência Computacional

Mais recentemente do que a origem dos estudos em IA, criou-se na academia a noção de *Inteligência Computacional* (IC) como uma sub-área da IA. Apesar de sua definição estrita (e ainda se esta deve ser considerada um tipo de Inteligência Artificial ou uma área de estudo completamente nova) ainda estar sob discussão, o espírito dos estudos em Inteligência Computacional tem seus fundamentos na busca por comportamentos baseados na natureza para resolver problemas complexos que, por não possuírem modelos matemáticos bem formados, não se adequam muito bem às abordagens de análise lógica exaustiva comummente utilizadas na IA tradicional.

Modelos de IC costumam ser inspirados em comportamentos biológicos de certos tipos de células, sociedades animais, raciocínio humano impreciso ou mesmo de mecanismos biológicos emergentes entre as espécies e o ambiente. Podemos citar como exemplos, respectivamente, as Redes Neurais Artificiais, modelos baseados em colônias de formigas e abelhas, Lógica Nebulosa e Algoritmos Genéticos.

Por trabalharem com problemas que não possuem modelagem matemática bem definida e, por consequência, algoritmos diretos para sua resolução, boa parte das técnicas de Inteligência Computacional baseia-se em aprendizado de uma grande quantidade de parâmetros a partir de um extenso conjunto de amostras de entradas e saídas esperadas. Desta forma, utiliza-se um modelo de algoritmo fixo baseado em algum comportamento natural, onde as particularidades do problema em questão são aproximadas pelos parâmetros deste algoritmo, e que por sua vez são deduzidos a partir de alguma métrica de erro após repetidas exposições das amostras de entrada

ao modelo definido.

1.2 Inteligência Artificial em Jogos Eletrônicos

A área de Jogos Eletrônicos, devido à sua enorme abrangência e ao grande número de gêneros de jogos existentes (Estratégia, Tiro em Primeira Pessoa, Tabuleiro, Puzzle etc) explora praticamente todas as técnicas de IA de alguma forma. Porém, pela sua aplicação em um domínio específico, a definição de *IA para jogos* pode ser um pouco mais restrita do que a definição de IA em geral.

SCHWAB (2004) define "Game AI" como "the code in a game that makes the computer-controlled elements appear to make smart decisions when the game has multiple choices for a given situation, resulting in behaviors that are relevant, effective, and useful". Ou seja, a IA em um jogo não precisa gerar resultados *ótimos* (no sentido matematicamente estrito da palavra), mas sim resultados que *pareçam inteligentes* e sejam relevantes e efetivos para a experiência do jogador.

Esta definição é de extrema importância, pois foca no objetivo final dos jogos eletrônicos: a experiência do jogador. Este foco garante a escolha e aplicação das técnicas mais adequadas e, em especial, possibilita a utilização de "atalhos" ou algoritmos mais simples que podem simplificar os problemas a serem resolvidos e diminuir o tempo de processamento necessário (o que é especialmente importante, conforme mencionado na seção 1.2.1).

Dentre exemplos da aplicação de técnicas de IA em jogos, podemos citar desde o *pathfinding* (busca por caminhos para, por exemplo, um personagem ir do ponto A ao ponto B em um ambiente) com o algoritmo A* até estudos recentes que buscam aplicação de técnicas de Inteligência Artificial ou Computacional para tornar perso-

nagens de jogos mais humanos: técnicas evolutivas e adaptativas OLIVEIRA CRUZ; DEMASI (2002, 2003, 2004); DEMASI (2003), Inteligência Computacional aplicada ao problema de *path following* AGUIAR GRIECO (2007), modelagem e simulação de emoções com Lógica Nebulosa ALVIM (2008); ALVIM; OLIVEIRA CRUZ (2008); LOPES (2011).

1.2.1 Questões de desempenho

Muitas técnicas de IA, principalmente as baseadas em aprendizado com Inteligência Computacional, requerem um alto poder de processamento. Como jogos costumam ser simulações pesadas, envolvendo, além do processamento de IA, gráficos complexos, gerenciamento do estado do mundo, física, colisões, controles etc, o tempo de processamento dedicado à IA acaba sofrendo e sendo negligenciado.

Atualmente, o modelo clássico de hardware onde os jogos eletrônicos são executados é uma máquina com uma ou mais CPUs (unidades de execução de código genérico) e uma ou mais GPUs (*Graphics Processing Units*, que são unidades de processamento gráfico). Mesmo com a variedade de plataformas atual–PCs, notebooks, consoles, *videogames* portáteis, celulares e *tablets*–todos seguem esta mesma arquitetura; mesmo os PCs e Notebooks mais simples (e a maioria dos portáteis) possuem uma GPU embutida no circuito integrado da CPU. Além disso, estão presentes conjuntos de memória RAM que podem ser acessados pela CPU e pela GPU. Dependendo da plataforma, esta memória pode ser compartilhada por ambas ou não.

A natureza complexa das simulações encontradas nos jogos eletrônicos modernos, em especial devido à complexidade gráfica (que é muito bem ilustrada na Figura 1.1) faz com que os diferentes módulos da simulação compitam entre si pelos

Figura 1.1: *Screenshot* do jogo *Bioshock Infinite*, da Irrational Games

recursos da máquina: CPU, GPU e memória. Cabe ao desenvolvedor do jogo dividir estes recursos entre cada um dos módulos da melhor maneira possível, considerando a máquina alvo e as características do jogo em questão.

Tradicionalmente, a GPU era responsável exclusivamente pela representação gráfica do jogo na tela, recebendo comandos da CPU e gerando o resultado final, com pouca programabilidade, de forma que a maioria das outras tarefas do jogo (incluindo a IA) eram necessariamente executadas na CPU. Porém, mais recentemente as GPUs têm se tornado cada vez mais programáveis, como será explorado a fundo no capítulo 4. Desta forma, a GPU passou a ser um recurso candidato a ser utilizado para outros tipos de cálculos, sendo o processamento de colisões e interações físicas o primeiro a ser explorado, a exemplo da biblioteca física PhysX, da nVidia[1]. Este novo campo de utilização da GPU é conhecido atualmente como *GPU Computing*.

[1]http://www.geforce.com/hardware/technology/physx

As GPUs são, na verdade, formadas por centenas de unidades aritméticas simples e muito rápidas, o que as torna ideais para processamento massivamente paralelo com independência de dados; isto, é claro, casa muito bem com o processamento gráfico de milhões de vértices e pixels independentes para o qual elas foram criadas. Mas outros tipos de processamento encontrados em jogos podem se beneficiar deste potencial massivamente paralelo; além da física, como mencionado anteriormente, a GPU torna-se um campo promissor para o estudo de técnicas mais robustas de processamento sonoro e, o que nos interessa em especial, Inteligência Artificial e Computacional.

1.3 Motivação

De acordo com o que foi apresentado na seção 1.2, o maior foco desta pesquisa foi a busca por soluções alternativas para IA em jogos, que explorem os avanços mais recentes em *hardware* para possibilitar a utilização de técnicas de IA que tradicionalmente foram negligenciadas devido ao seu alto custo computacional.

Além disso, a busca pela otimização de técnicas de aprendizado computacional também possui inúmeros usos em áreas que trabalham com tratamento de grandes volumes de dados, onde o custo (de tempo e energia) com o processamento por cada unidade de dado é fator determinante no sucesso da aplicação. Apesar deste não ser o foco do presente trabalho, podemos citar como exemplos de aplicações: meteorologia, mapeamento geológico e busca de recursos naturais, processamento de grande quantidade de imagens ou vídeo, reconhecimento biométrico, além de muitos problemas em bioinformática, a exemplo de WANDERLEY et al. (2014).

1.4 Objetivos

Este trabalho foca na otimização de técnicas de aprendizado computacional com Lógica Nebulosa e sua aplicação em tempo real, com foco na utilização em jogos eletrônicos. Mais especificamente, no treinamento de Sistemas de Inferência Nebulosa do tipo TSK (Takagi-Sugeno-Kang) utilizando *GPU Computing* na plataforma CUDA, da nVidia[2] e a utilização de sistemas TSK para geração e atualização em tempo real de um oponente virtual em um jogo.

O primeiro objetivo do trabalho (explorado no capítulo 5) é desenvolver um método completo para treinamento de sistemas nebulosos TSK utilizando o paralelismo proporcionado pelas técnicas de GPU Computing. De especial interesse na modelagem deste método é a exploração do paralelismo do problema de forma a maximizar os ganhos em cenários mais prováveis de ocorrerem em jogos.

O segundo e final objetivo buscado (e descrito no capítulo 6) consiste na criação de uma prova de conceito da aplicação do método estudado em jogos eletrônicos modernos, através da criação de um jogo 3D simples que utilize algum tipo de IA baseada em aprendizado de máquina com sistemas nebulosos TSK.

Desta forma, o trabalho busca gerar fortes contribuições em duas frentes: a exploração das modernas tecnologias de processamento massivamente paralelo em GPUs para treinamento de sistemas de Lógica Nebulosa, e provar que a tecnologia dos processadores modernos chegou a um patamar que permite a utilização em tempo real de sistemas de aprendizado com Inteligência Computacional e grandes quantidades de dados.

[2] http://www.nvidia.com/cuda

1.5 Estrutura do texto

O capítulo 2 consiste em um estudo e revisão da literatura recente acerca dos temas abordados ao longo do texto, além de alguns trabalhos similares e relacionados ao método desenvolvido.

O capítulo 3 faz uma breve exposição da Lógica Nebulosa, em especial sobre sistemas de inferência nebulosa do tipo TSK, e explora algumas técnicas que podem ser utilizadas para treinamento de seus parâmetros a partir de um conjunto de amostras conhecidas.

O capítulo 4 parte de um breve histórico sobre a área de GPU Computing, além de explorar maiores detalhes sobre a arquitetura CUDA, suas particularidades e melhores técnicas de programação.

O capítulo 5 descreve com detalhes o método desenvolvido para treinamento de sistemas nebulosos TSK utilizando a arquitetura CUDA, incluindo testes de desempenho e comparação com implementações de referência em CPU.

O capítulo 6 fala sobre os experimentos realizados com o método de treinamento estudado aplicado em jogos em tempo real e seus resultados.

O capítulo 7 apresentará as conclusões obtidas ao final do desenvolvimento do trabalho e sugestões de trabalhos futuros.

2 REVISÃO DA LITERATURA

2.1 Fundamentos teóricos

Todo o trabalho desenvolvido nesta pesquisa baseou-se em fundamentos gerais e bem conceituados de Inteligência Artificial e Computacional, incluindo sua aplicação em jogos eletrônicos.

Considera-se RUSSELL; NORVIG (2003) como o pilar mais importante dos estudos modernos em IA, devido a sua extensão, didática e reconhecimento acadêmico evidenciado por sua adoção quase unânime em cursos de graduação, inclusive o curso de Ciência da Computação da UFRJ. Apesar de não se aprofundar em temas de Inteligência Computacional, suas definições básicas e classificações de áreas da IA são ótimas referências para delimitar os estudos e facilitar sua compreensão.

Enquanto isso, SCHWAB (2004) é centrado especificamente em IA para jogos, delimitando com cuidado o que se considera uma IA "boa" nesta situação. Esta definição é essencialmente diferente da definição acadêmica usualmente utilizada (onde em geral busca-se a otimização numérica dos resultados), já que uma boa IA em um jogo é aquela que otimiza parâmetros qualitativos como a experiência, a satisfação e o nível de desafio do jogador. Além disso, este livro também apresenta uma extensa discussão de técnicas utilizadas pela indústria em jogos publicados, separadas por gênero, com uma quantidade enorme de exemplos de código.

Voltando-se para a Inteligência Computacional, em especial a Lógica Nebulosa, considera-se a origem acadêmica desta em ZADEH (1965), onde são inicial-

mente definidos o conceito de conjuntos nebulosos e relações entre eles, em oposição a conjuntos binários da lógica clássica. Logo seguem-se trabalhos que constroem em cima destas definições para a criação de sistemas de inferência baseados nos conjuntos nebulosos, sendo os mais famosos o Mamdani MAMDANI (1977) e o Takagi-Sugeno-Kang TAKAGY; SUGENO (1985); SUGENO; KANG (1988).

Posteriormente, desenvolveram-se algumas estruturas de sistemas de inferência baseados em Lógica Nebulosa inspirados na arquitetura das Redes Neurais Artificiais PáDUA BRAGA; CARVALHO; LUDEMIR (2007). O ANFIS JANG (1993) e, mais recentemente, o SONFIN JUANG; LIN (1998) são exemplos deste tipo de estrutura. Enquanto, essencialmente, são sistemas de inferência do tipo Takagi-Sugeno-Kang, seus métodos de treinamento baseiam-se em modelar a estrutura do sistema como uma rede de nós por onde um número de valores de entrada passam e são transformados para gerar valores de saída, e aplicar técnicas de treinamento inspiradas em uma analogia com as Redes Neurais Artificiais.

Finalmente, existe também uma literatura farta sobre a utilização de Lógica Nebulosa e sistemas de inferência nebulosos para a engenharia e o controle ROSS (2010); PASSINO; YURKOVICH (1998). Estes trabalhos contém várias técnicas de treinamento que podem ser aproveitadas para uma infinidade de aplicações, incluindo a utilização da técnica BLS, que será utilizada como parte do método desenvolvido no capítulo 5.

É importante ressaltar que a teoria sobre Lógica Nebulosa, sistemas de inferência nebulosos e o treinamento destes será explorada mais a fundo no capítulo 3.

2.2 Trabalhos relacionados

Esta seção faz um breve apanhado sobre trabalhos relacionados ao treinamento de sistemas baseados em Inteligência Computacional e como eles se relacionam com os objetivos deste trabalho, introduzindo os fundamentos que levaram ao posterior desenvolvimento do método explicado no capítulo 5. Também são exploradas outras aplicações de Inteligência Computacional em jogos.

2.2.1 Paralelismo e aprendizado em Inteligência Computacional

Inicialmente, podemos citar alguns trabalhos relacionados na área da implementação de métodos de aprendizado com grande quantidade de dados com algoritmos paralelos, tanto em CPUs quanto em GPUs.

Sendo a abordagem mais utilizada baseada em aprendizado com amostras, podemos encontrar vários exemplos deste tipo que trabalham com Redes Neurais Artificiais. Por exemplo, em NAGESWARAN et al. (2009); UETZ; BEHNKE (2009) são apresentadas implementações de diferentes tipos de RNAs usando CUDA. Já em JANG; PARK; JUNG (2008) podemos encontrar uma abordagem híbrida para RNAs que combina CUDA e OpenMP para conseguir o melhor desempenho possível explorando simultaneamente o paralelismo oferecido pela CPU e pela GPU.

Considerando métodos no domínio da Lógica Nebulosa, ANDERSON; COUPLAND (2008) adaptaram o modelo clássico e baseado em linguística Mamdani para ser executado em GPUs com CUDA, com resultados promissores para grandes volumes de dados. Em um escopo maior, fora de sistemas de inferência, uma abordagem para processamento de imagens baseada em Lógica Nebulosa e implementada com CUDA pode ser encontrada em LUKE et al. (2009).

Finalmente, uma implementação completa de um método paralelo para treinamento de sistemas TSK a partir de amostras utilizando CUDA foi feito por JUANG; CHEN; CHENG (2011). Neste artigo, uma estrutura chamada GPU-FNN é proposta, baseada na estrutura e nos métodos de aprendizado do SONFIN JUANG; LIN (1998). Os resultados obtidos são muito encorajadores para problemas com entradas de alta dimensionalidade, já que o método de paralelização proposto explora o processamento concorrente no domínio das dimensões da entrada e das regras, mas não entre as amostras em si, dado que na GPU-FNN as amostras são processadas serialmente. Sendo assim, ela não é uma escolha muito boa para problemas com uma grande quantidade de amostras e/ou um número pequeno de dimensões de entrada (como alguns que podem ser encontrados em jogos eletrônicos, como por exemplo a previsão do comportamento do jogador baseado em dados históricos).

2.2.2 Inteligência Computacional e adaptatividade em jogos

Fazendo um breve apanhado de pesquisas recentes na aplicação de Inteligência Computacional em jogos, não se pode deixar de citar as contribuições de OLIVEIRA CRUZ; DEMASI (2002, 2003, 2004); DEMASI (2003) através de várias técnicas evolutivas e adaptativas em tempo real para uso em jogos eletrônicos. Estas vão desde previsão de comportamentos baseada em predição sequencial até aprendizado de regras nebulosas em tempo real para a criação de um oponente personalizado enquanto se joga. Em especial, podemos citar o jogo de naves espaciais desenvolvido em OLIVEIRA CRUZ; DEMASI (2002) como um ótimo arcabouço para o teste de métodos para aprendizado de regras em tempo real.

No âmbito geral da adaptatividade de dificuldade em jogos, podemos citar o trabalho de ARAUJO; FEIJó (2012), que foi baseado em uma pesquisa a respeito da motivação dos jogadores e o que eles buscam nos jogos; a isso somou-se o desenvolvi-

mento de um sistema de adaptação dinâmica de dificuldade baseado no desempenho do jogador e sua efetividade foi avaliada em relação à manutenção do interesse do jogador ao longo do jogo.

Em aplicações não diretamente relacionadas ao controle de oponentes, podemos citar a abordagem de AGUIAR GRIECO (2007) ao problema de *path following* utilizando técnicas de Inteligência Computacional para a geração de caminhos que pareçam mais naturais. Em ALVIM; OLIVEIRA CRUZ (2008); ALVIM (2008) temos a criação de uma máquina de estados nebulosa onde os estados representam diferentes emoções, onde os graus de cada emoção podem ser aplicados à geração de *feedback* visual para o jogador ou à mudança de estratégia do oponente. Finalmente, LOPES (2011) estuda a aplicação de um modelo de emoções com memória e que sofrem influência das características do ambiente para auxiliar na tomada de decisão de um agente no cenário de um jogo.

Podemos dizer que este trabalho é, principalmente, uma extensão das ideias que começaram a ser desenvolvidas por DEMASI (2003) para o estudo de técnicas de Inteligência Computacional para controle de agentes em tempo real em jogos eletrônicos. Esta extensão se faz tanto no domínio da modelagem dos agentes (como explorado no capítulo 6) quanto nas tecnologias utilizadas em sua implantação (discutido no capítulo 5).

3 LÓGICA NEBULOSA E SISTEMAS DE INFERÊNCIA NEBULOSOS

3.1 Introdução

A Lógica Nebulosa ZADEH (1965), ou Lógica Fuzzy, está entre as técnicas mais utilizadas para o tratamento de dados e inferência de informações ultimamente, juntamente com outros métodos da Inteligência Computacional como Redes Neurais Artificiais (RNAs) e Algoritmos Genéticos. Alguns dos usos comuns da Lógica Nebulosa são sistemas de controle, problemas de classificação, regressão de dados etc. Em particular, a utilização de algumas técnicas da Lógica Nebulosa em jogos também foi estudada recentemente, conforme visto na seção 2.2.2 (OLIVEIRA CRUZ; DEMASI (2002); DEMASI (2003); OLIVEIRA CRUZ; DEMASI (2003); ALVIM; OLIVEIRA CRUZ (2008); LOPES (2011)).

3.2 Variáveis e conjuntos

Na lógica proposicional clássica, podemos considerar que uma variável é pertencente a apenas um conjunto de cada vez. Por exemplo, podemos dizer que *Bruno é alto*, o que implica uma certeza de 100% de que Bruno pertence ao conjunto dos altos e de que não pertence ao conjunto dos baixos, ou dos de estatura mediana. Supondo que uma estatura mediana seja aquela compreendida entre 160cm e 180cm, um gráfico que representa os conjuntos (baixo, mediano, alto) em relação ao valor da altura poderia ter a aparência descrita na Figura 3.1. Pode-se perceber que há uma mudança brusca na classificação dos valores próximos às fronteiras entre os

conjuntos.

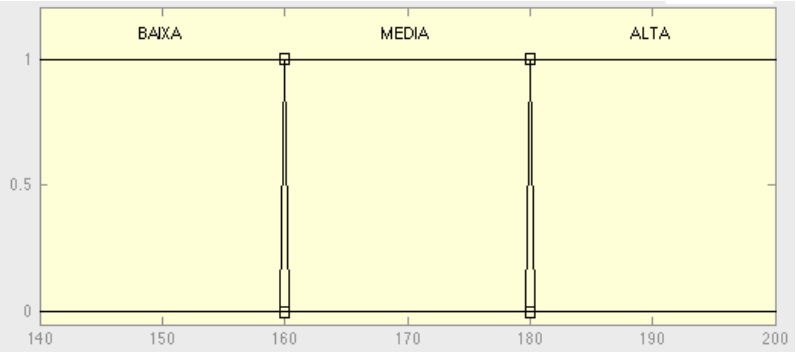

Figura 3.1: Um possível gráfico de conjuntos para a variável lógica clássica *Altura*

A essência da Lógica Nebulosa está em estender este raciocínio, criando regiões de incerteza onde o valor de uma variável pode fazer com que ela pertença simultaneamente a mais de um conjunto, com valores de certeza entre 0 e 1. Por exemplo, se considerarmos uma região de tolerância de 10cm em cada uma das separações descritas acima, uma pessoa com altura entre 175cm e 185cm seria ao mesmo tempo de estatura mediana e alta, dependendo de quão perto esteja de cada um destes valores. Considerando o gráfico mostrado na Figura 3.2, uma pessoa com altura igual a 180cm seria de estatura mediana com 50% de certeza e alta com 50% de certeza. Ou seja, pertenceria ao conjunto dos medianos com grau 0.5 e ao conjunto dos altos também com grau 0.5. Ao mesmo tempo, uma altura de 185cm seria suficiente para declarar com 100% de certeza que o indivíduo em questão é alto.

As funções que determinam a relação entre o valor de uma variável e o grau de pertinência em um dado conjunto são chamadas *funções de pertinência* e em geral possuem a forma de um triângulo, trapézio ou (em situações em que é desejável que sejam contínuas) uma curva gaussiana, dentre outras possibilidades.x

Partindo-se apenas desta definição abre-se um enorme leque de possibilidades e, ao mesmo tempo, lacunas para a definição das operações clássicas de conjuntos

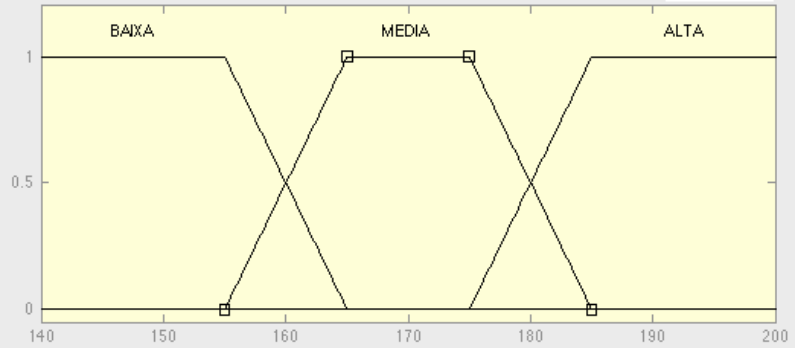

Figura 3.2: Gráfico de conjuntos para a variável *Altura*, com conjuntos nebulosos

neste novo domínio. Em especial, é importante discutir brevemente como podem ser implementadas as operações de união (equivalente ao operador OU) e interseção (equivalente ao operador E).

Voltando ao exemplo anterior: se temos, na lógica clássica, que *Bruno é alto* e *Bruno é magro*, podemos afirmar com 100% de certeza (ou grau 1) que Bruno é *alto E magro*. Ao mesmo tempo, também temos total certeza que Bruno é *alto OU gordo*, já que é alto, apesar de não pertencer ao conjunto dos gordos.

Considerando uma situação em Lógica Nebulosa em que, por exemplo, Bruno seja alto com grau 0.5, magro com grau 0.8 e gordo com grau 0.2, é necessário definir uma maneira de chegar a um valor numérico para representar o grau de certeza com que Bruno é *alto E magro* e com que grau ele é *alto OU gordo*. Mais do que isso, seria desejável que estas operações correspondessem exatamente às definições da lógica clássica para casos com graus iguais a 0 e 1 apenas.

Uma discussão mais profunda sobre as propriedades que estas operações devem ter pode ser encontrada na literatura especializada ROSS (2010); PASSINO; YURKOVICH (1998), mas podemos citar rapidamente algumas das mais utilizadas dentre as operações viáveis.

Considerando a operação $x = a\ E\ b$, podemos utilizar as seguintes fórmulas:

Mínimo $x = min(a, b)$.

Produto $x = a * b$.

Já para a operação $x = a\ OU\ b$, listamos algumas das operações mais utilizadas:

Máximo $x = max(a, b)$.

Soma probabilística $x = a + b - a * b$.

Finalmente, voltando ao exemplo, aplicando os operadores *mínimo* e *máximo*, a afirmação *Bruno é alto E magro* teria grau igual a $min(0.5, 0.8) = 0.5$, enquanto *Bruno é alto OU gordo* teria grau igual a $max(0.5, 0.2) = 0.5$. Já com as operações de *produto* e *soma probabilística*, os valores seriam iguais a $(0.5 * 0.8) = 0.4$ e $(0.5 + 0.2 - 0.5 * 0.2) = 0.6$, respectivamente.

3.3 Sistemas de inferência nebulosos

Uma das estruturas que utilizam as representações e princípios da Lógica Nebulosa para processamento de dados e tomada de decisões são os sistemas de inferência nebulosos (FIS - Fuzzy Inference System).

Um sistema de inferência nebuloso funciona como uma "caixa preta" com um certo número de entradas e saídas, quase sempre representadas por números reais. O

processo de inferência realizado pelo sistema consiste, então, em calcular as saídas baseado nos valores apresentados como entradas para o mesmo. Visto de fora, o processo de inferência se assemelha bastante com outros métodos como as Redes Neurais Artificiais, apesar de seu funcionamento ser bastante diferente.

Em um FIS, cada uma dessas entradas (e, dependendo do tipo de sistema, também as saídas) é representada como uma variável nebulosa do tipo descrito na seção 3.2 e composta por um certo número de conjuntos ou funções de pertinência. Estas funções de pertinência são então definidas para o domínio da variável em questão dependendo do tipo de dado que se espera receber como entrada (ou gerar como saída) no sistema.

Além das variáveis de entrada e saída, um FIS também é definido por um conjunto de *regras*. Estas regras são, em essência, o que define o processo responsável por obter as saídas do sistema a partir das entradas apresentadas. Existem alguns tipos diferentes de FIS, dependendo na maneira como as regras do sistema se apresentam, são avaliadas e seus consequentes são combinados para gerar o resultado final. Dois destes principais tipos são descritos nas seções a seguir.

3.3.1 Mamdani

O modelo Mamdani MAMDANI (1977) de sistema de inferência nebuloso é baseado em raciocínio lingüístico e regras definidas por um especialista humano. Neste tipo de sistema, tanto as variáveis de entrada como as de saída são representadas como variáveis nebulosas e divididas em conjuntos em geral definidos manualmente a partir de conhecimento especializado do problema a ser tratado. Sendo assim, os consequentes das regras de inferência em um FIS Mamdani são conjuntos nebulosos definidos para alguma variável de saída. Uma regra Mamdani tem, então,

a forma:

se *A* é *A1* E *B* é *B1* E ... E *N* é *N1* **então** *X* é *X1*,

onde A, B, ..., N são variáveis de entrada, A1, B1, ..., N1 são certos conjuntos destas variáveis, X é uma variável de saída e X1 é um conjunto desta variável.

Um exemplo de regra real baseado nos conceitos exemplificados na seção 3.2 poderia ser:

se *Altura* é *Alto* E *Peso* é *Baixo* **então** *Tipo físico* é *Atlético*.

O processo de inferência de um sistema Mamdani começa pela avaliação das regras. A cada regra é dada uma *força* equivalente à interseção da força de cada um dos conjuntos dos antecedentes da regra. Ou seja, cada antecedente é avaliado separadamente, e o grau de pertinência de cada um dos conjuntos nas variáveis correspondentes é combinado através da operação de interseção, ou E lógico, definida para o sistema (de acordo com a seção 3.2).

A força de cada regra é então aplicada ao conjunto da variável de saída do consequente para gerar uma nova função de pertinência ponderada por esta força; os operadores utilizados para isto são, em geral, mínimo (que "corta" a função) ou produto (que torna a função "achatada").

Após obter todas as funções de pertinência de uma variável de saída, ponderadas pelas forças das regras, estas são combinadas de alguma maneira (em geral com o operador de máximo) para gerar uma única função no domínio da variável.

Esta função é então transformada em um valor único de saída através de um processo chamado *desnebulização* (ou *defuzzificação*).

Este trabalho não utiliza sistemas do tipo Mamdani, mas explicações mais detalhadas sobre seu funcionamento podem ser encontradas em ROSS (2010); PASSINO; YURKOVICH (1998); DEMASI (2003); LOPES (2011).

3.3.2 Takagi-Sugeno-Kang

Em comparação com o Mamdani, o modelo TSK (Takagi-Sugeno-Kang) de FIS é baseado em um modelo matemático mais flexível, onde o consequente de uma regra para cada variável de saída é uma função polinomial de ordem N (para algum N) dos valores de entrada. Sendo assim, um sistema TSK de ordem 0 tem como consequente de todas as regras apenas um número real, e pode ser considerado análogo a um sistema Mamdani onde os conjuntos das variáveis de saída são definidos como pontos singulares.

Devido à natureza paramétrica e mais matematicamente flexível do modelo TSK, ele é em geral uma boa escolha para aprendizado automático de sistemas inteligentes baseado em exemplos. Os parâmetros a serem aprendidos para definir um sistema TSK, então, corresponderiam à definição dos conjuntos das variáveis de entrada (ou funções de pertinência), os coeficientes dos polinômios das saídas e as combinações de conjuntos nebulosos que formam o antecedente de cada regra do sistema. Funções gaussianas costumam ser usadas para aprendizado automático, devido a suas propriedades de continuidade e diferenciabilidade, as quais são necessárias para alguns métodos de aprendizado como o método do gradiente (ROSS (2010)), explicado na seção 3.4.2.

Sendo assim, podemos dizer que, formalmente, um FIS do tipo TSK é unicamente definido por:

- Variáveis de entrada e saída (e algum número de conjuntos nebulosos para cada), onde cada conjunto nebuloso é uma função que mapeia o espaço da variável de entrada para o domínio [0, 1] e representa o *grau de pertinência* neste conjunto de algum valor real que a variável possa assumir;
- Regras de inferência, onde:
  - Antecedentes são o E de tuplas da forma *variável É conjunto*, onde:
    - *variável* é uma variável de entrada do sistema considerado, e
    - *conjunto* é um dos conjuntos nebulosos definidos para esta variável.
  - Consequentes são da forma *variável É* $P(x_1, x_2, ..., x_n)$, onde:
    - *variável* é uma variável de saída do sistema considerado,
    - $x = x_1, x_2, ..., x_n$ são os valores reais das variáveis de entrada, e
    - $P(x)$ representa um polinômio de ordem n (para qualquer n maior ou igual a 0) destes valores.

Um exemplo de regra no modelo TSK seria como a seguir:

se *Experiencia* é *Alto* E *Desempenho* é *Alto* **então** *Salario* é
$5000 + Experiencia + Desempenho^2$.

O cálculo da saída de um sistema TSK difere bastante de um sistema Mamdani (consulte a seção 3.3.1), e convém defini-lo com um rigor matemático maior para facilitar a avaliação e definição dos métodos de aprendizado, que são baseados

em otimização e minimização de erros. Podemos então dizer que a saída de um sistema deste tipo é feita da seguinte forma:

- Cada regra é avaliada e a ela é dado um *grau de ativação* baseado na combinação de graus de pertinência retornados por cada par (variável, conjunto) definido nos antecedentes da regra. Esta combinação é feita da mesma maneira que num sistema Mamdani, mas em geral usa-se o operador de produto para simplificar a equação final que representa o sistema e garantir que ela continue diferenciável em todos os pontos. Considerando então o operador de produto, o grau de ativação da regra k seria definido como

$$\mu_k(x) = \prod_{j=1}^{V} M_j^k(x_j), \tag{3.1}$$

  onde V é o número de variáveis de entrada e M_j^k é o grau de pertinência para a variável x_j retornado pelo conjunto nebuloso definido para esta variável no antecedente da regra k. Considerando-se funções de pertinência gaussianas para as variáveis de entrada,

$$M_j^k(x_j) = exp\left(-0.5 * \left(\frac{x_j - c_j^k}{w_j^k}\right)^2\right), \tag{3.2}$$

  onde c_j^k e w_j^k são os centros e larguras da função gaussiana de pertinência definida para a variável x_j e o conjunto correspondente a esta variável no antecedente da regra k. Sendo assim, a expressão completa para o grau de ativação da regra k será

$$\mu_k(x) = \prod_{j=1}^{M} exp\left(-0.5 * \left(\frac{x_j - c_j^k}{w_j^k}\right)^2\right). \tag{3.3}$$

- A saída final de cada variável é calculada como uma média ponderada da avaliação dos polinômios de saída de cada regra, onde o peso dado a cada

valor de saída é o grau de ativação da regra correspondente. Desta forma, se $P_k(x)$ representa a saída da regra k, a saída final para uma variável do sistema nebuloso é

$$f(x) = \frac{\sum_{k=1}^{K} \mu_k(x) * P_k(x)}{\sum_{k=1}^{K} \mu_k(x)}. \tag{3.4}$$

Podemos então concluir que, no fim das contas, a saída $f(x)$ de um sistema nebuloso TSK como o descrito acima para um vetor de entrada x depende unicamente dos valores de c_j^k, w_j^k e dos coeficientes dos polinômios $P_k(x)$. A seção 3.4 discutirá métodos para encontrar estes valores através de treinamento supervisionado baseado apenas em um conjunto de entradas e saídas conhecidas.

3.4 Treinamento de sistemas nebulosos TSK

Vários métodos existem para treinamento e ajuste de parâmetros de sistemas nebulosos TSK ROSS (2010); PASSINO; YURKOVICH (1998), alguns dos quais são métodos baseados em redes JANG (1993); JUANG; LIN (1998) que tomam emprestadas idéias e inspiração das arquiteturas e métodos de treinamento das Redes Neurais Artificiais.

Dois métodos em especial que são aplicáveis a alguns ou todos os parâmetros que devem ser aprendidos são discutidos nas seções 3.4.1 e 3.4.2. É importante considerar que a métrica tipicamente utilizada como objetivo de otimização (e que é utilizada nos métodos que serão explorados a seguir) é o Erro Médio Quadrático (MSE - Mean Squared Error), que é calculado da seguinte forma:

$$\frac{\sum_{i=1}^{N} \sum_{l=1}^{L} (y_l^d(i) - y_l(i))^2}{N}, \tag{3.5}$$

onde N é o número de amostras utilizadas no treinamento, L é o número de variáveis de saída, $y_l^d(i)$ é o valor desejado de saída para a *i-ésima* amostra e a *l-ésima* variável de saída, e $y_l(i)$ é a saída correspondente dada pelo sistema treinado.

3.4.1 O método Batch Least Squares para ajuste das saídas

Uma das maneiras clássicas de treinamento para os parâmetros dos consequentes é transformar o problema em um sistema linear de equações sobredeterminado e resolve-lo com a abordagem dos Mínimos Quadrados. Isto garante valores ótimos para os parâmetros dos consequentes quando a métrica a ser minimizada é o Erro Médio Quadrático RAO; TOUTENBURG (1999), que é por definição equivalente à minimização da soma das distâncias entre as amostras do sistema e o plano definido como solução, encontrado pelo método dos Mínimos Quadrados.

Como este método serve para realizar a adaptação apenas dos coeficientes dos polinômios de saída, podemos realizar a análise inicialmente considerando apenas uma variável de saída. A explicação a seguir é baseada em ROSS (2010). Começamos transformando o problema em um de otimização linear. Primeiro, observamos que a equação (3.4) pode ser reescrita como

$$f(x) = \sum_{k=1}^{K} \left(\frac{\mu_k(x)}{\sum_{k'=1}^{K} \mu_{k'}(x)} * P_k(x) \right) \tag{3.6}$$

Agora, consideremos por um momento que todos os polinômios de saída são de ordem 0. Esta simplificação momentânea não retira a validade do raciocínio para polinômios de grau mais elevado (como será observado ao fim da explicação), mas facilita o desenvolvimento. Sendo assim, cada um dos polinômios $P_k(x)$ corresponde a apenas um valor constante. Isto nos leva a

$$f(x) = \sum_{k=1}^{K} \left(\frac{\mu_k(x)}{\sum_{k'=1}^{K} \mu_{k'}(x)} * p_k \right), \tag{3.7}$$

onde p_k é a saída constante da regra k. Agora, observamos que são os valores de p_k que queremos encontrar, minimizando o Erro Médio Quadrático entre $f(x)$ e o valor de saída da amostra de treinamento y_d. Se considerarmos ϕ como sendo um vetor contendo os valores constantes (em relação a p_k) de $\frac{\mu_k(x)}{\sum_{k'=1}^{K} \mu_{k'}(x)}$ e θ como sendo o vetor dos valores p_k, então a equação (3.7) se torna

$$f(x) = \phi \cdot \theta. \tag{3.8}$$

Isto é considerando apenas uma amostra. Então, considerando Φ como a matriz formada pela concatenação dos vetores ϕ de todas as amostras, Y como o vetor de saídas das amostras de treinamento para todas as amostras, e que o nosso objetivo é otimizar para o melhor ajuste de $f(x)$ para Y, encontramos o sistema linear sobredeterminado (considerando $N > K$) final a ser otimizado:

$$\Phi\theta = Y \tag{3.9}$$

De acordo com RAO; TOUTENBURG (1999); ROSS (2010), este sistema pode ser resolvido (em relação aos mínimos quadrados) para θ resolvendo-se o sistema linear (determinado) dado por

$$(\Phi^T\Phi)\theta = (\Phi^T Y). \tag{3.10}$$

É importante notar que este raciocínio pode ser estendido para polinômios de saída com graus maiores que 0. Neste caso, θ (e, por conseguinte, ϕ) seriam aumentados para compreender os coeficientes adicionais. Os valores extras em ϕ seriam então iguais aos originais (para um sistema de ordem 0), multiplicados por cada um dos produtos das variáveis de entrada de acordo com o formato dos polinômios (i.e. x_i, x_i^2, $x_i x_j$ etc).

Da mesma forma, podemos considerar que Y (e logo, θ) possui L colunas, e interpretar isto como um problema de Mínimos Quadrados para mais de um vetor à direita, de modo a resolver o problema para várias variáveis de saída ao mesmo tempo. Dependendo da implementação, isto pode economizar tempo de processamento devido a ser necessário pré-processar Φ (por exemplo, por algum tipo de fatoração) apenas uma vez para todos os vetores de saída.

Um outro método de treinamento derivado do BLS é o RLS (Recursive Least Squares). Este consiste em executar a otimização dos mínimos quadrados uma amostra por vez, utilizando a matriz anterior como base para o resultado do próximo passo de uma maneira recursiva (daí o seu nome). De acordo com ROSS (2010), as equações para treinamento de uma amostra são derivadas da equação (3.10) e dadas por

$$P(i) = \frac{1}{\lambda}(I - P(i-1)\phi_i(\lambda I + \phi_i^T P(i-1)\phi_i)^{-1}\phi_i^T)P(i-1) \tag{3.11}$$

$$\theta_i = \theta_{i-1} + P(i)\phi_i(Y_i - \phi_i^T\theta_{i-1}), \tag{3.12}$$

onde i é a iteração atual, λ é um fator de aprendizado e P é uma matriz auxiliar que é atualizada a cada passo e inicializada da forma $P = \alpha I$, com $\alpha > 0$

e preferencialmente grande ROSS (2010). Além disso, o vetor θ também deve ser inicializado com uma estimativa inicial; por isso pode vir a ser uma boa ideia aplicar o método BLS para algumas amostras e obter esta estimativa antes de passar-se à utilização do RLS.

Apesar do método RLS ser especialmente interessante para cenários interativos onde a coleta das amostras é feita em tempo real, sua utilização para os objetivos propostos neste trabalho se mostra pouco convidativa devido à impossibilidade de explorar paralelismo entre as amostras de treinamento (já que, neste método, é apresentada uma de cada vez). Mesmo considerando que as operações realizadas nas equações (3.11) e (3.12) na verdade são bastante simples (pois as matrizes envolvidas são vetores ou diagonais), o tempo de execução de uma versão paralelizada do método ainda estaria restrito pelo número de amostras.

3.4.2 O método do gradiente para ajuste de parâmetros

Enquanto o método BLS (descrito na seção 3.4.1) encontra a solução ótima para θ em apenas uma iteração, ele pode ser um pouco lento (devido à necessidade de resolver um sistema linear que pode se tornar grande), além de não poder ser utilizado para ajustar os parâmetros gaussianos dos antecedentes das regras.

O método do gradiente, por outro lado, pode ser utilizado para ajustar todos os parâmetros de um sistema TSKJANG (1993); ROSS (2010); JUANG; CHEN; CHENG (2011). Ele é um método recorrente em análise numérica para encontrar mínimos (ou máximos) de funções de dimensionalidade alta, e consiste em, começando de um ponto válido, caminhar no espaço das soluções na direção e sentido opostos do vetor gradiente da função no ponto atual por uma certa distância. Se a função a ser minimizada é bem comportada (no sentido de ter poucos platôs e

mínimos ou máximos locais), o método do gradiente em geral consegue encontrar uma boa solução.

As maiores desvantagens do método são:

- a sensibilidade aos mínimos e máximos locais, platôs, tamanho do passo e ponto inicial;
- a necessidade de que a função a ser otimizada seja totalmente contínua e diferenciável no espaço considerado;
- a quantidade de iterações necessárias para atingir a convergência, inversamente proporcional ao tamanho do passo;
- a falta da garantia de encontrar um mínimo ou máximo global;
- a necessidade de definir o número de iterações e o tamanho do passo empiricamente, dependendo do problema.

Para aplicar o método do gradiente ao problema do treinamento de um sistema TSK, queremos ajustar os valores de c_j^k, w_j^k e p_k para minimizar o Erro Médio Quadrático como definido na equação (3.5). Considerando um passo de tamanho igual a η, e de acordo com JUANG; CHEN; CHENG (2011), uma iteração do método do gradiente para cada um destes parâmetros, considerando o Erro Médio Quadrático para uma única amostra, corresponde às seguintes equações:

$$c_j^k(t+1) = c_j^k - \eta * \sum_{l=1}^{L}[(y_l - y_l^d)(p_k^l - y_l)] * \phi_k * \frac{(x_j - c_j^k)}{(w_j^k)^2} \tag{3.13}$$

$$w_j^k(t+1) = w_j^k - \eta * \sum_{l=1}^{L}[(y_l - y_l^d)(p_k^l - y_l)] * \phi_k * \frac{(x_j - c_j^k)^2}{(w_j^k)^2} \tag{3.14}$$

$$p_k^l(t+1) = p_k^l - \eta * (y_l - y_l^d) * \phi_k, \tag{3.15}$$

onde L é o número de variáveis de saída. Podemos observar que pode-se eliminar a soma se estivermos considerando cada variável de saída como um sistema isolado com regras e conjuntos de antecedentes distintos, já que neste caso $L = 1$.

Se desejarmos então aplicar um passo de ajuste considerando o Erro Médio Quadrático de todas as amostras ao mesmo tempo, é suficiente (pois o gradiente é um operador linear) calcular a média dos gradientes para cada uma das amostras (ou seja, a parte depois de η nas equações acima) e utilizar esta média nas equações de ajuste apenas uma vez. Esta estratégia é especialmente útil para ambientes paralelos, como ficará claro no capítulo 5.

4 GPU COMPUTING E CUDA

4.1 Introdução

Unidades de processamento gráfico (Graphics Processing Unit - GPU) são processadores intrinsicamente paralelos. Suas arquiteturas, originalmente criadas e evoluídas com o intuito de transformar, rasterizar, texturizar e iluminar polígonos e pixels giram em torno da repetição da mesma (e muitas vezes simples) operação – qualquer uma das listadas anteriormente – para uma grande quantidade de dados diferentes. Por exemplo, uma aplicação pode desejar rotacionar no espaço tridimensional um modelo composto de milhares de triângulos; para este fim, essa aplicação enviaria (dentre outras coisas) um comando para a GPU a fim de transformar todos esses triângulos utilizando a mesma matriz de rotação e obter um modelo rotacionado uniformemente.

Entretanto, as primeiras GPUs possuíam hardware bastante específico e pouco programável, o que tornava a utilização destas para cálculos arbitrários uma tarefa complexa, com o esforço extra frequentemente neutralizando os ganhos em paralelização. Mesmo assim, na medida em que o poder de computação e programabilidade das GPUs aumentou, alguns grupos conseguiram desenvolver métodos para usa-las em computação paralela de propósito geral. Isto era feito "disfarçando" estes cálculos como operações para as quais a GPU foi desenvolvida para realizar, ou seja, transformação e desenho de primitivas geométricas. Nesta época, alguns tipos de operações primitivas em programação paralela, notavelmente a operação de *scatter* (veja seção 4.3), eram difíceis de serem realizadas em GPUs devido a não haver um correspondente direto no *pipeline* de renderização OWENS et al. (2007). Esta nova área de pesquisa era chamada, à época, de GPGPU (General Purpose Computing

on Graphics Processing Unit).

Este cenário mudou drasticamente com o lançamento da arquitetura CUDA (Compute Unified Device Architecture) da nVidia, uma das maiores fabricantes de hardware gráfico no mundo. CUDA[1] é uma extensão da linguagem C/C++, sendo que através de uma API e algumas palavras-chaves novas permitem a desenvolvedores criar código paralelo de propósito geral que é executado na GPU utilizando uma linguagem e ambiente de desenvolvimento familiares e poderosos, sem a necessidade trabalhosa e ineficiente de representar algoritmos como uma sequência de operações gráficas. Apesar de ainda existirem limitações e peculiaridades devido à arquitetura de baixo nível de uma GPU ser bastante diferente do conhecido mundo das CPUs, a linguagem CUDA abriu o caminho para a pesquisa e o uso ubíquos da programação de propósito geral em GPU, recentemente consolidada com o nome de *GPU Computing*.

Outros arcabouços similares também foram criados com o mesmo objetivo, como OpenCL[2] e DirectCompute[3]; este capítulo focará na descrição da plataforma CUDA por ter sido a utilizada no trabalho, mas grande parte dos conceitos e técnicas é comum a todas as plataformas de GPU Computing.

[1] http://www.nvidia.com/cuda
[2] http://www.khronos.org/opencl/
[3] http://en.wikipedia.org/wiki/DirectCompute

4.2 A plataforma CUDA

4.2.1 Arquitetura Geral

Como mencionado brevemente na seção 4.1, CUDA é uma extensão de C++ criada pela nVidia que permite o desenvolvimento de código de propósito geral que é executado na GPU utilizando uma linguagem e ambiente familiares. No paradigma CUDA, código de CPU e GPU coexistem no mesmo projeto, e funções que rodam em GPU (chamadas *kernels*) são marcadas com palavras-chave especiais na sua declaração. É importante ressaltar que código CUDA compilado, até a época desta pesquisa, só pode ser executado em alguns hardwares da nVidia, apesar de alguns esforços para emular a API CUDA em outras plataformas estarem em execução[4].

No estudo da plataforma CUDA, são de especial interesse a organização de processadores e memória em seu modelo de programação, por diferirem bastante dos paradigmas comuns em programação para CPUs. Como pode ser visto na Figura 4.1, processadores individuais (SPs - Stream Processors) são agrupados dentro de multiprocessadores (SMs - Stream Multiprocessors). A grosso modo, isso significa que cada SP dentro de um dado SM está executando a mesma instrução a cada momento (mas em diferentes conjuntos de dados). Este conhecimento é importante para a consideração de desvios de código divergentes dentro do mesmo SM, que podem prejudicar a performance do código devido a alguns SPs estarem ociosos enquanto se executa um bloco de código em um caminho que não foi tomado por eles. Podemos notar também que a quantidade de SMs disponíveis em uma GPU, assim como o número de SPs em cada SM, é variável dependendo do modelo do hardware.

[4]http://code.google.com/p/gpuocelot/

Na API CUDA, unidades de execução em SPs são abstraídas como *threads* e as unidades de execução em SMs são representados como *blocos*. Além disso, a API utilizada neste trabalho (CUDA 4.0) requer que o desenvolvedor especifique manualmente o tamanho (quantidade de threads) de cada bloco. O tamanho ótimo do bloco de threads é, de maneira geral, um exercício em tentativa e erro porque depende de fatores como a natureza do problema sendo tratado e a organização de seus dados, a maneira como os kernels são programados e mesmo considerações sobre o hardware em que o código será executado devido a diferenças em memória e organização dos núcleos de processamento. Por outro lado, alguns problemas possuem inerentemente um modelo de dados em dois níveis, e se encaixam bem nesta organização hierárquica (como será visto no capítulo 5).

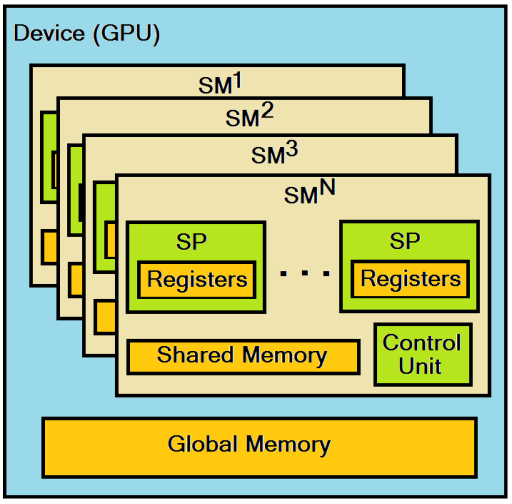

Figura 4.1: Arquitetura geral CUDA simplificada.

4.2.2 Tipos de memória

A organização de memória no modelo CUDA também é um pouco complicada. Excluindo alguns tipos de *cache* que são invisíveis ao programador (mas devem ser considerados para obter um desempenho realmente ótimo, assim como

em programação para CPU), podemos listar os seguintes tipos de memória que estão disponíveis para uso direto:

- *Memória global* é o tipo de memória mais comum, maior e mais lento da GPU, em geral medida desde 256MB até alguns GB. Suporta acesso aleatório de leitura e escrita a qualquer momento, sem restrições ou otimizações.
- *Memória de textura* é fisicamente a mesma que a memória global, mas é acessada com um hardware dedicado que pode otimizar certos padrões de acesso e realizar interpolação quase gratuita entre posições adjacentes. Como seu nome sugere, ela foi originalmente desenvolvida para otimizar o acesso a texturas que são mapeadas em polígonos.
- *Memória constante* também está localizada no espaço global de memória mas possui acesso otimizado para dados que sejam constantes (durante o tempo de vida da execução de um kernel). Este tipo de memória é originalmente utilizado para armazenar parâmetros globais de renderização.
- *Memória compartilhada* é pequena (da ordem de dezenas ou centenas de KB) e de muito rápido acesso, localizada no chip, dentro de cada SM. Desta forma, como o nome sugere, ela é compartilhada entre todos os SPs em um SM e pode ser lida e escrita por todas as threads de um mesmo bloco (mas não entre blocos diferentes). A memória compartilhada pode ser acessada somente de dentro dos kernels e não está visível para a CPU. Isto significa que qualquer dado armazenado nesta memória deve ser gerado ou copiado por código de GPU antes de ser utilizado, o que o torna ideal para armazenamento de dados temporários ou como *cache* para dados comuns a um bloco de threads.
- *Registradores e memória local* formam a região de memória local a uma única thread, onde todas as variáveis locais e a pilha de chamadas de uma thread estão localizadas.

4.2.3 Organização da API

A API da plataforma CUDA pode ser dividida em algumas grandes áreas que agrupam tipos específicos de tarefas, a saber:

- *Chamadas de gerenciamento*: englobam as funções de gerenciamento de *streams*, sincronização ou bloqueio de fluxos de código, tratamento de erros e gerenciamento de eventos.
- *Chamadas de acesso a memória*: são acessadas por código de CPU e são responsáveis pela alocação, gerenciamento e acesso à memória de GPU a partir do código de CPU. Em especial, estão contidas neste conjunto as funções responsáveis pela alocação e liberação de espaços de memória em GPU e cópia de dados entre memória de CPU e GPU ou entre diferentes espaços de GPU.
- *Chamadas de controle de execução*: incluem chamadas a código de GPU escrito pelo desenvolvedor do programa e tarefas de gerenciamento relativos a elas. Chamadas de kernel em CUDA podem utilizar uma sintaxe especial que permite a passagem de parâmetros relativos à execução do referido bloco de código, como quantidade de blocos e threads utilizados pelo kernel em questão.
- *Chamadas de interoperabilidade gráfica*: podem ser utilizadas para comunicação entre código CUDA e APIs gráficas como OpenGL e DirectX.

4.2.4 Ciclo de vida de um programa CUDA

Podemos dizer que um programa CUDA (construído para resolver apenas um problema em particular), em geral, pode ter seu ciclo de vida resumido da seguinte forma:

- *Inicialização do programa*;
- *Alocação de estruturas de dados em CPU e GPU*: Alocação de estruturas em CPU para conter os dados iniciais do programa, em geral através de chamadas a **malloc** e em GPU para receberem esses mesmos dados, com chamadas a **cudaMalloc**;
- *Preparação dos dados de entrada do problema em CPU*: Geração dos dados de entrada do problema. Este processo em geral é feito em CPU por ser bastante específico e usualmente depender de dados externos que não estão disponíveis em GPU, como arquivos;
- *Cópia dos dados de entrada para GPU*: É necessário realizar a cópia dos dados iniciais definidos anteriormente em CPU para a memória de GPU de forma que estes possam ser processados. Isto é feito através de chamadas a **cudaMemcpy**;
- *Chamada de um ou mais kernels de GPU para processamento dos dados*;
- *Cópia dos dados processados de volta para CPU*: Após a chamada dos kernels, uma ou mais chamadas a **cudaMemcpy** devem ser feitas para recuperar os dados processados;
- *Apresentação ou posterior utilização dos resultados.*

4.2.5 Padrões de acesso a memória e sincronização

Assim como o hardware de execução de código das GPUs, a memória deste tipo de processador também é otimizada para acesso simultâneo a várias unidades de dados, e seguindo padrões consistentes, o que é chamado de *coalescência*. Desta forma, recomenda-se idealmente que todas as threads em execução acessem a memória ao mesmo tempo, e preferencialmente threads com IDs consecutivos acessem

endereços consecutivos de memória. Desta forma, vários acessos feitos por threads diferentes podem ser combinados pelo hardware em apenas um acesso. Apesar de versões mais recentes dos hardwares da nVidia e do *runtime* do CUDA serem mais permissivos em relação a esses padrões (a partir do CUDA 3.x), este tipo de padrão de programação ainda é considerado uma boa prática.

Dentro do paradigma CUDA, código de CPU não pode acessar diretamente o código ou espaço de memória de GPU e vice-versa. Toda transferência de dados deve ser feita via chamadas de função do tipo *cudaMemcpy(...)*. Estas chamadas podem copiar um conjunto de dados de memória de CPU para GPU (o que é chamado de cópia *Host to Device*), GPU para CPU (*Device to Host*) ou GPU para GPU (*Device to Device*). Além disso, toda a alocação de memória de GPU (*device*), exceto memória local e compartilhada (que são declaradas dentro do kernel), deve ser feita a partir de código CPU através de chamadas a *cudaMalloc*.

Chamadas de kernels e chamadas a *cudaMemcpyAsync* são não bloqueantes em CPU a princípio, mas bloquearão na próxima chamada a um kernel ou *cudaMemcpy(...)* até que seu processamento em GPU esteja terminado, a fim de evitar problemas de sincronização. Entretanto, pode-se utilizar uma funcionalidade da API chamada *CUDA streams* para paralelizar mais de uma sequência de comandos de GPU, aproveitando ao máximo o potencial disponível no hardware.

4.3 Técnicas de Programação Paralela e seu uso em GPU Computing

Esta seção busca listar algumas técnicas de alto nível utilizadas em programação paralela e distribuída, de especial interesse neste trabalho, à luz de sua utilização no ambiente de GPU Computing. Um breve resumo sobre operações mencionadas

neste trabalho é feito a seguir, em especial sobre seu uso com a tecnologia CUDA, levando em consideração suas particularidades, restrições e melhores práticas de otimização.

- *gather* e *scatter*: estas operações básicas consistem, respectivamente, em fazer um conjunto de leituras e escritas em posições aleatórias dentro de um espaço de memória. Apesar de serem triviais em CPUs monoprocessadas (que são razoavelmente otimizadas para acessos aleatórios), essas operações podem ser mais custosas em ambientes paralelos, principalmente as GPUs, que originalmente foram desenvolvidas para realizar operações de leitura e escrita em conjuntos de dados com posições específicas (por exemplo, o desenho de um polígono em *pixels* adjacentes na tela ou a leitura de uma textura inteira para mapeamento). Antes do advento de tecnologias que permitem programação de propósito geral em GPUs (como CUDA), a realização destas operações costumava ser bastante difícil, principalmente a operação de *scatter* (OWENS et al. (2007)). Plataformas mais modernas permitem uma maior facilidade ao programador nesse sentido, mas ainda é fortemente recomendável seguir alguns princípios e recomendações para obter-se o melhor desempenho possível do hardware, conforme foi visto na seção 4.2.5.

- *reduce* (ou *redução*): este procedimento consiste em, dada uma lista de elementos, realizar uma operação seguidamente em todos os elementos da lista, de forma a obter apenas um elemento no final. Exemplos clássicos desta operação são somatórios, produtórios, operações de máximo e mínimo e operações booleanas (e, ou, ou exclusivo). Operações deste tipo são muito recorrentes em programação de maneira geral, sendo o caso da maior parte dos usos de *loops*, mas sua implementação em programação paralela merece atenção especial: se a operação a ser realizada tiver as propriedades adequadas (principalmente associatividade), seu tempo de execução em um ambiente paralelo pode ser tão

baixo quando $O\big(log(N)\big)$ (em oposição a $O\big(N\big)$ em um ambiente sequencial), onde N é o tamanho da lista de elementos.

5 TREINAMENTO DE SISTEMAS TSK EM GPU

5.1 Introdução

Este capítulo descreve um método desenvolvido para o treinamento de sistemas de inferência nebulosos do tipo TSK com GPU Computing, utilizando a plataforma CUDA. O desenvolvimento do método descrito a seguir é norteado pelo desejo de utilizar este tipo de sistema como mais uma alternativa para IA em jogos, notavelmente para a geração de oponentes virtuais em tempo real. Este direcionamento geral foi responsável por várias decisões na modelagem da paralelização do treinamento, de modo a explorar o melhor desempenho possível em situações mais prováveis de acontecerem neste tipo de utilização; este pensamento será revisitado na seção 5.3.5, durante a avaliação dos resultados obtidos.

5.2 Resumo do Método

O método proposto neste trabalho para treinamento de um sistema TSK foi inspirado em JUANG; CHEN; CHENG (2011) e possui as seguintes características:

- Definição das funções de pertinência iniciais para cada conjunto dividindo-se o espaço de entrada de cada variável uniformemente com um certo número de conjuntos de largura uniforme; este número deve ser determinado empiricamente dependendo do problema. Outros métodos mais refinados poderiam ser utilizados para a definição da configuração inicial do sistema (como os baseados em *clusterização*), desde que possam trabalhar em todas as amostras

simultaneamente e estejam baseados apenas nos dados de treinamento e não em parte dos resultados. Isto foi feito para garantir que fosse possível explorar paralelismo entre as amostras de treinamento, e torna o método descrito em JUANG; CHEN; CHENG (2011) inadequado, pois ele obriga o processamento serial das amostras;

- Geração de regras iniciais através da combinação dos conjuntos definidos anteriormente. Todas as combinações possíveis entre os conjuntos definidos segundo o método descrito anteriormente são utilizadas como antecedentes de regras, gerando uma regra para cada combinação. Isto garante que o espaço de entrada do problema seja totalmente coberto por regras; caso contrário, a saída do sistema para uma entrada em uma região descoberta poderia ser indefinida;

- Todas as regras possuem todas as variáveis nos antecedentes e o conjunto relativo a cada variável em uma regra é definido como único para aquela regra. Mesmo que várias regras sejam inicializadas com conjuntos iguais para certas variáveis no começo do treinamento, a evolução dos parâmetros das funções de pertinência é única para cada regra e poderá fazer com que estes conjuntos divirjam ao longo do treinamento. Esta escolha foi feita para simplificar a modelagem do problema e maximizar a exploração do paralelismo;

- Adoção de polinômios de saída de ordem zero (constantes) para as regras, por motivos de simplificação do processo de treinamento, mas que poderiam ser facilmente estendidos para ordens maiores de acordo com a seção 3.4.1;

- Utilização do método BLS (descrito na seção 3.4.1) para ajuste dos consequentes das regras, devido a sua simplicidade e paralelismo inerente, por ser um método matricial;

- Aplicação do método do gradiente (descrito na seção 3.4.2) para ajuste dos conjuntos das variáveis de entrada, por ser um método relativamente simples

dentre os viáveis para esta tarefa, e adequado para ajustes finos.

Pode-se notar que a descrição geral do método é, para efeitos do processamento de treinamento realizado, basicamente a mesma que o ANFIS original, como descrito em JANG (1993). As particularidades deste método consistem na definição de conjuntos específicos para cada regra e no fato das amostras serem todas apresentadas simultaneamente para efeito do cálculo do gradiente, de modo que o gradiente global (média do gradiente para cada amostra) seja utilizado para o ajuste dos antecedentes, em vez de alternar-se uma operação de mínimos quadrados global com um ajuste de gradiente de uma amostra.

Sejam os valores, conforme definidos na seção 3.4,

- $n\_iterations$ um número arbitrário de iterações desejado para a convergência do sistema;
- x_j^i o valor da i-ésima amostra para a j-ésima variável;
- phi^i o vetor de graus de ativação em cada regra atribuídos para a i-ésima amostra;
- c_j^k e w_j^k o centro e a largura, respectivamente, de cada função gaussiana definida como conjunto de entrada para a j-ésima variável na k-ésima regra;
- Y o vetor de saídas esperadas para uma dada variável de saída;
- θ o vetor de coeficientes dos polinômios de saída de cada regra,

então o funcionamento do algoritmo desenvolvido seria como a seguir:

- Definir as funções de pertinência iniciais espalhando-as igualmente pelo domínio de cada variável de entrada;

- repetir $n\_iterations$ vezes:

  - Calcular a matriz Φ, concatenando os valores de ϕ^i para cada amostra baseado em x_j^i, c_j^k and w_j^k;
  - Encontrar θ solucionando $\Phi\theta = Y$ com o método dos mínimos quadrados para todas as variáveis de saída;
  - Adaptar c_j^k e w_j^k com o método do gradiente fazendo a média dos gradientes de erro para cada amostra.

5.2.1 Implementação em CPU

Uma implementação de referência em CPU foi construída para fins de verificação de corretude e comparação de performance com o método em GPU. Esta implementação utilizou apenas código sequencial, sem explorar nenhum tipo de paralelismo em nível de CPU com bibliotecas como OpenMP ou MPI. O algoritmo utilizado foi baseado na mesma modelagem do método em GPU, sem as estruturas de paralelismo mas baseado nas mesmas equações de adaptação.

O cálculo de Φ e o ajuste do método do gradiente foram implementados diretamente, e para solucionar o problema dos Mínimos Quadrados utilizou-se uma versão da biblioteca LAPACK. Como os testes foram realizados em um ambiente Windows, a implementação LAPACK utilizada para compilação foram binários pré-compilados para Windows fornecidos gratuitamente pelo Innovative Computing Laboratory da University of Tennessee[1].

[1] http://icl.cs.utk.edu/lapack-for-windows/

5.2.2 Implementação em GPU

Todos os algoritmos, exceto a definição inicial dos parâmetros das gaussianas (que possui tempo desprezível) foram implementados paralelamente em CUDA.

Assim como na implementação de CPU, o código para calcular Φ e o ajuste dos antecedentes com o método do gradiente foram escritos como uma implementação direta dos algoritmos, sem a utilização de nenhuma biblioteca. Ao mesmo tempo, a biblioteca CULA [2] foi usada para resolver o problema dos Mínimos Quadrados em GPU. CULA é uma biblioteca comercial que contém código de Álgebra Linear otimizado para CUDA, sendo a maior parte uma implementação das interfaces de programação para Álgebra Linear BLAS e LAPACK. Ela possui uma versão gratuita com um número limitado de funções, uma das quais resolve o problema de mínimos quadrados linear necessário para o trabalho em questão.

O código escrito diretamente foi separado em 3 kernels diferentes, cujos funcionamentos serão detalhados nas seções seguintes. A técnica de *redução*, clássica na área de programação paralela, é utilizada várias vezes para encontrar a soma de todos os elementos em uma lista de valores. Estas reduções são todas feitas dentro de um único bloco de threads, utilizando memória compartilhada para armazenamento, o que torna as operações fáceis de serem implementadas e muito eficientes. A implementação da operação de redução é basicamente a mesma que em JUANG; CHEN; CHENG (2011).

Outro detalhe de implementação interessante é que o número de entradas e saídas do sistema foram passados para os kernels como parâmetros de *templates*. Esta é uma funcionalidade da linguagem C++, suportada pelo compilador CUDA, que marca certos parâmetros de uma função como valores constantes em tempo de

[2] http://www.culatools.com/

compilação. Estes valores são, então, substituídos dentro do código da função nos lugares ocupados pelos nomes dos parâmetros correspondentes. A maior vantagem desta funcionalidade é permitir a propagação de constantes no código durante a etapa de compilação, permitindo otimizações mais agressivas, mantendo o código simples e genérico. No caso deste trabalho, isso permite que o compilador CUDA expanda pequenos *loops* dentro dos kernels de modo a otimizar um pouco mais o tempo de execução do código através da eliminação de instruções de desvio, que são bastante custosas de serem executadas em GPUs.

As seções seguintes detalham cada etapa da implementação em GPU. A definição de cada kernel é acompanhada pela explicação da partição de blocos e threads empregada–ou seja, o que cada bloco e cada thread de CUDA representam e como se mapeiam para unidade do problema a ser resolvido. Segue-se uma descrição de alto nível detalhando a sequência de passos realizada por cada kernel. É importante ressaltar que a sequência de passos descrita em cada seção é executada *sequencialmente* dentro do kernel. Os passos foram separados no texto apenas para facilitar o entendimento das sub-etapas do fluxo do algoritmo para cada parte do processamento.

5.2.2.1 Inicialização

Antes de iniciar o *loop* principal do algoritmo e executar o primeiro kernel, os dados iniciais (amostras de entrada e saída e parâmetros iniciais das gaussianas) são copiados da memória *host* (CPU) para *device* (GPU). Então o loop principal se inicia e o processo descrito nas seções 5.2.2.2 até 5.2.2.5 é executado $n\_iterations$ vezes.

5.2.2.2 Calculando Phi

A partição utilizada para este kernel foi de um bloco para cada amostra e uma thread para cada regra. Optou-se por apenas uma regra por thread para manter o código inicial mais simples, porém poder-se-ia implementar este kernel também como processando um certo número constante de threads por regra, aumentando assim o número máximo de regras suportadas.

Devido ao código de redução utilizado precisar trabalhar com uma quantidade de valores que seja potência de dois (por motivos de otimização), o número de regras do sistema é arredondado para cima até a potência de dois mais próxima e o valor de ativação das regras "extras"é definido como 0, para não influenciar no resultado final.

O funcionamento deste kernel então é como a seguir:

- *cada thread*: Calcular o grau de ativação (μ_k) para a regra e amostra atuais (de acordo com a equação (3.3)) e armazena-lo em memória compartilhada. O produtório contido nesta equação é implementado sequencialmente dentro de cada thread;
- *todas as threads em cada bloco*: Calcular a soma de todos os graus de ativação para este bloco através de redução em memória compartilhada;
- *cada thread*: Normalizar o grau de ativação da regra atual (para a amostra atual) dividindo-o pela soma de todos os graus e armazená-lo em Φ.

Após a execução deste kernel temos a matriz Φ completa e pronta em memória global de GPU. Como um detalhe de implementação, uma segunda cópia da matriz é

gerada em memória de GPU depois da chamada do kernel (através de uma chamada a *cudaMemcpy*) porque a função que realiza a operação de mínimos quadrados utiliza a matriz passada como argumento como espaço de trabalho e modifica seus valores. Esta cópia é realizada totalmente em GPU, já que tanto a origem quanto o destino da cópia estão em localizadas em memória da GPU.

5.2.2.3 Encontrando θ com Mínimos Quadrados

Isto consiste em uma simples chamada à biblioteca CULA, feita em código de CPU, passando a matriz Φ calculada para encontrar o vetor θ correspondente. Antes da chamada à biblioteca, uma cópia de Y é feita em GPU, porque o resultado do cálculo é escrito no mesmo espaço de memória passado como o lado direito do sistema linear.

A biblioteca CULA não expõe sua estrutura interna de kernels, de forma que esta chamada é feita como uma chamada de função normal do C++, com a particularidade de que alguns parâmetros são ponteiros para endereços de memória de GPU.

Além disso, a biblioteca permite que sejam passadas várias colunas do lado direito de uma só vez para realizar a estimativa de mínimos quadrados para vários sistemas ao mesmo tempo (desde que possuam o mesmo lado esquerdo na equação matricial). Esta funcionalidade foi utilizada para otimizar o resultado em sistemas com mais de uma variável de saída.

5.2.2.4 Calculando os gradientes de erro

O cálculo e a aplicação dos gradientes de erro (para ajuste das curvas gaussianas nos antecedentes) foram divididos em dois kernels. O primeiro calcula os gradientes de erro para cada amostra. Já o segundo é responsável por realizar a média de todos eles e aplica-los aos parâmetros. Esta divisão foi feita porque a partição de blocos e threads definida para cada parte do cálculo é diferente.

A partição para o primeiro kernel é a mesma que a utilizada para calcular Φ: um bloco por amostra e uma thread por regra. A mesma idéia de valores "extras"por regra foi utilizada para arredondar a operação de redução presente. Este kernel é basicamente responsável por calcular as equações (3.13) e (3.14) e funciona da seguinte maneira:

- *cada thread*: Calcular um elemento da equação (3.8) para todas as saídas e armazená-lo em memória compartilhada;
- *todas as threads em cada bloco*: Calcular a soma de todos os valores encontrados anteriormente para cada saída, utilizando redução em memória compartilhada. O resultado desta operação é a saída do sistema nebuloso para cada amostra e variável de saída;
- *uma thread por bloco*: Encontrar o erro para a amostra atual subtraindo a saída esperada (em Y) da saída calculada;
- *cada thread*: Calcular os gradientes de erro de c_j^k e w_j^k para a amostra e regra correspondentes, e para cada variável de entrada. Armazená-los em memória global.

Esta chamada de kernel é imediatamente seguida pela próxima.

5.2.2.5 Aplicando os gradientes de erro

A maior parte do trabalho no último kernel é calcular a média de cada gradiente de erro através de todas as amostras. Isto é feito com uma operação de redução dentro de cada bloco. Sendo assim, a partição para este kernel é de um bloco por regra, e uma thread para uma certa quantidade de amostras.

Como não podemos iniciar uma thread para cada amostra dentro de um bloco devido ao limite de threads por bloco (além disso, seria uma decisão de eficiência ruim porque provavelmente espalharia um mesmo bloco por vários SMs), o número de amostras é dividido por um número fixo de grupos (32 nesta implementação) e uma thread é criada para cada grupo. Cada thread primeiro soma, sequencialmente, os gradientes de erro para todas as amostras em seu grupo. Após isto, uma redução é feita para obter a soma global correspondente à regra definida para o bloco considerado. A primeira thread do bloco então calcula as médias finais e as aplica para ajustar os valores de c_j^k e w_j^k. Um cuidado extra na implementação deve ser tomado para tratar o último grupo, já que ele pode ter um tamanho diferente dos demais.

O funcionamento deste kernel é, então, como a seguir:

- *cada thread*: Calcular a soma dos gradientes de erro para todas as amostras dentro do grupo correspondente e armazena-la em memória compartilhada;
- *todas as threads em cada bloco*: Encontrar a soma dos gradientes de todos os grupos utilizando redução em memória compartilhada. O resultado desta operação corresponde à soma dos gradientes para todas as amostras;
- *uma thread por bloco*: Calcular as médias finais dividindo as somas dos gradientes pelo número de amostras;

- *uma thread por bloco*: Aplicar as médias finais dos gradientes para ajustar os valores de c_j^k e w_j^k.

Após a execução deste kernel, uma iteração de ajuste do sistema nebuloso estará completa. Como agora os antecedentes mudaram devido ao ajuste do método do gradiente, na próxima iteração os valores em Φ estarão diferentes, e a solução para o problema dos mínimos quadrados também será, desejavelmente, mais próxima do valor ótimo global para o sistema completo. Este processo deverá continuar até que se esteja satisfeito com o resultado do treinamento.

Ao fim de cada iteração, o estado do sistema nebuloso e todas as matrizes temporárias estão disponíveis em memória de GPU e podem ser copiados para memória de CPU para verificação e/ou utilização. Uma possível aplicação neste sentido seria ler de volta o erro médio quadrático para a última iteração realizada e utiliza-lo para decidir a respeito de parar o treinamento ou mudar algum parâmetro do algoritmo (por exemplo, o tamanho do passo no método do gradiente).

5.2.2.6 Considerações Finais

A plataforma CUDA possui um limite de 65536 blocos para cada chamada de kernel, e 1024 threads para cada bloco (no momento da elaboração deste trabalho). Devido às partições de blocos e threads utilizadas neste método, a princípio existe um limite inicial de 1024 regras, 65536 amostras para treinamento em um dado momento. Porém, estes limites podem ser aumentados de certa forma realizando-se uma adaptação do algoritmo para que ele funcione em conjuntos de 65536 amostras de cada vez (e o mesmo para o número de regras) ou adaptando os kernels para processarem mais de uma regra sequencialmente em cada thread. Porém, isto não foi feito neste trabalho porque os casos testados funcionaram suficientemente bem

com 65536 ou menos amostras e menos de 1024 regras.

5.3 Testes

O método desenvolvido neste trabalho e descrito nas seções anteriores foi testado através do treinamento de um sistema nebuloso TSK de ordem zero com duas entradas e 35 regras. As versões de CPU e GPU do algoritmo foram executadas em 3 máquinas diferentes, rodando cada uma cinco casos de teste, com 16^2, 32^2, 64^2, 128^2 e 256^2 amostras.

Cada sessão de treinamento durou 100 iterações. Este valor foi escolhido após algumas rodadas de teste como um meio termo razoável entre rapidez do treinamento e bons resultados. Além disso, valores maiores não fizeram com que o valor do erro diminuísse muito mais e demonstraram sinais de *overlearning*, ou seja, quando o sistema se especializa demais nas amostras apresentadas e perde sua capacidade de generalização. Este número de iterações foi mantido em todos os testes, a fim de validar a comparação de desempenho entre eles.

5.3.1 Dados de Treinamento

A origem dos dados de treinamento é um sistema de inferência Mamdani utilizado para estacionar um carro simulado, dirigindo com a marcha a ré em velocidade constante, como descrito em NGUYEN; WIDROW (1990); KOSKO (1992). O carro a ser estacionado existe em um espaço de 100x100 unidades de tamanho e a posição de destino (a "vaga"onde o carro deve estacionar) encontra-se no topo e no meio deste espaço (ou seja, em $x = 50, y = 100$). As entradas deste sistema são a posição x atual e o ângulo do carro (a posição y não é utilizada como entrada pelo

sistema). A única variável de saída é um ângulo que representa quanto as rodas da frente do veículo devem ser giradas em relação à sua posição neutra (para frente). Este sistema é comumente utilizado para estudo e aprendizado da lógica nebulosa, com uma simulação que coloca o carro em alguma posição do espaço inicialmente e executa o sistema descrito a cada passo de tempo para determinar o ângulo atual que as rodas devem ser giradas.

Como este sistema possui 5 conjuntos para a variável de posição e 7 conjuntos para o ângulo, foi definido que as mesmas entradas no sistema TSK treinado também começariam com 5 e 7 funções de pertinência gaussianas, respectivamente para as variáveis correspondentes. Desta forma, esperar-se-ia que o método do gradiente as ajustasse para o melhor formato de forma a representar o sistema original, ou seja, para um formato parecido com as variáveis originais.

Amostras deste sistema foram então obtidas dividindo-se o espaço das amostras de entrada em 16x16, 32x32, 64x64, 128x128 e 256x256 pontos e avaliando o sistema em cada um destes pontos. Esta operação resultou em 5 conjuntos de dados, contendo 16^2, 32^2, 64^2, 128^2 e 256^2 amostras.

5.3.2 Avaliação de corretude

Como o objetivo dos testes referidos foi de avaliar o desempenho da implementação em GPU do método criado (e não o método em si, que já é largamente estudado na literatura JANG (1993); ROSS (2010)), nenhuma validação das saídas do sistema foi realizada através de separação do conjunto de amostras entre conjuntos de treinamento e teste.

Entretanto, alguns testes foram realizados para verificar se a implementação

do método estava correta, ou seja, realizando corretamente os cálculos descritos na seção 3.4 e resultando nos mesmos valores que a implementação de referência em CPU.

Primeiramente, foi-se verificado que os resultados obtidos pela implementação em GPU foram realmente os mesmos que aqueles obtidos pela versão CPU através da execução dos mesmos cenários de treinamento com as duas implementações e comparação do valores finais obtidos.

Além disso, foi também feita uma análise subjetiva da evolução do erro médio quadrático do treinamento a cada passo. O comportamento esperado, neste caso, é que o erro comece com um valor mediano na primeira iteração (já que a estimativa de mínimos quadrados com conjuntos gaussianos padrões é o suficiente para uma aproximação grosseira) e caia para um valor menor com o passar das iterações de treinamento (devido ao ajuste fino dos parâmetros das funções gaussianas que representam os conjuntos de entrada).

Durante o teste seguinte, o tamanho do passo no método do gradiente para cada conjunto de amostras foi ajustado para conseguir o menor erro médio quadrático possível em 100 iterações do processo. Então o método desenvolvido foi executado (em GPU) para cada um dos conjuntos de amostras utilizando o tamanho de passo encontrado. A tabela 5.1 mostra o valor do erro na primeira e na última iteração e o tamanho do passo para cada conjunto de amostras.

5.3.3 Especificações de hardware e software

O código dos testes foi compilado em Windows 7 com o compilador do Microsoft Visual Studio 2010 e o CUDA Toolkit versão 4.2. O compilador foi configurado

N. de amostras	Erro Inicial	Erro Final	Tam. Passo
16^2	5.5322	0.8894	50
32^2	7.4268	1.6556	40
64^2	7.4422	1.5335	10
128^2	7.3233	1.4878	25
256^2	7.3569	1.5175	40

Tabela 5.1: Erro inicial, erro final e tamanho do passo de aprendizado para cada conjunto de amostras

para gerar código 32-bits.

Os testes foram executados em 3 máquinas diferentes (a serem referidas como Máquina 1, Máquina 2 e Máquina 3). Todas as máquinas rodavam Windows 7 Professional 64-bit. A tabela 5.2 contém as especificações de CPU, GPU e RAM de cada máquina.

Máquina	CPU	RAM	GPU
Máquina 1	i7 2600 @3.40GHz	8GB	GeForce GTX 550 Ti
Máquina 2	i7 960 @3.20GHz	16GB	Tesla C2070
Máquina 3	i7 960 @3.20GHz	16GB	GeForce GTX 590

Tabela 5.2: Configurações das máquinas de teste

5.3.4 Metodologia de teste e tempos resultantes

Para obter resultados dos testes, o tempo gasto em cada operação do algoritmo (calculando Φ, mínimos quadrados e método do gradiente) em cada iteração foi gravado. O tempo das operações de GPU foi medido utilizando-se uma funcionalidade da API CUDA chamada *CUDA Events* que permite a marcação de vários *eventos* em lugares do código e posterior cálculo do intervalo de tempo entre quaisquer dois eventos. Já o tempo das operações de CPU foi medido com a chamada da API do Windows *QueryPerformanceCounter*. Os valores de tempo para cada iteração foram então somados para encontrar-se os tempos totais listados. É importante

lembrar também que a implementação de CPU utilizada para os testes é puramente sequencial.

O tempo gasto para a cópia inicial de amostras e parâmetros gaussianos, assim como copiar de volta os parâmetros aprendidos pelo método, não foi contado por ser desprezível comparado a 100 iterações do método de treinamento. Podemos ressaltar que a cópia inicial de amostras é necessária porque as amostras são coletadas em código de CPU; no caso dos testes estas foram carregadas do disco e, em um caso geral em tempo real, seriam coletadas durante o *gameplay*. Os parâmetros iniciais gas gaussianas poderiam ser calculados paralelamente com um kernel de GPU, mas esta possibilidade não foi explorada por ser considerado um esforço desnecessário dado o tempo desprezível da operação de geração e cópia destes valores frente a dezenas de iterações do método de treinamento.

O tempo gasto para duplicar o vetor Y de valores de saída em cada iteração também foi muito pequeno, e não está listado em separado nas tabelas, apesar de ter sido contabilizado nos cálculos de tempo total e *speedup* para garantir a precisão nas comparações. Da mesma forma, o tempo gasto para a duplicação da matriz Φ está contabilizado juntamente com o cálculo de seus valores.

Cada teste foi realizado apenas uma vez em cada máquina, com 100 iterações em cada. Foi considerado desnecessário executar o processo completo várias vezes em cada máquina devido à uma realização do processo completo na verdade consistir de 100 repetições do mesmo código, o que corresponderia à realização de uma média sobre 100 execuções do algoritmo executado por uma única iteração.

As tabelas e figuras a seguir contêm os resultados dos testes realizados. A tabela 5.3 mostra os tempos totais de CPU e GPU e speedups para todas as 3 máquinas de teste. Já a tabela 5.4 mostra o detalhamento de tempos totais separados

por cada uma das três etapas principais do algoritmo.

Máquina	Tempo CPU	Tempo GPU	Speedup
Máquina 1	66.3738	13.5498	4.8985x
Máquina 2	83.3153	13.9587	5.9687x
Máquina 3	82.9066	13.3409	6.2145x

Tabela 5.3: Tempo total de execução (em segundos) para CPU e GPU e speedup, para 256^2 amostras

Máquina		Calc. Phi	Mín. Quadrados	Gradiente
Máquina 1	CPU	15.9303	42.1538	8.1316
	GPU	1.1651	8.8011	3.5791
Speedup		13.6729x	4.7896x	2.2719x
Máquina 2	CPU	19.3489	55.3174	8.4074
	GPU	0.4865	11.9729	1.4928
Speedup		39.7716x	4.6202x	5.6319x
Máquina 3	CPU	19.2381	55.0543	8.3742
	GPU	0.4101	11.7816	1.1435
Speedup		46.9107x	4.6729x	7.3233x

Tabela 5.4: Detalhamento de tempo de execução (em segundos) para cada passo em todas as três máquinas de teste, para 256^2 amostras

Modelo	Clock Principal	Clock de Memória	Número de Núcleos
GeForce GTX 550 Ti	900 MHz	4104 MHz	192
Tesla C2070	575 MHz	1500 MHz	448
GeForce GTX 590	607 MHz	3414 MHz	1024

Tabela 5.5: Características principais das GPUs utilizadas nos testes

A Figura 5.1 mostra a evolução dos tempos e speedups alcançados conforme se aumentou o número de amostras, na máquina 3. É importante observar que os dois eixos da figura da esquerda, e o eixo horizontal da figura da direita, são logarítmicos.

Finalmente, a Figura 5.2 mostra uma comparação dos tempos em CPU e GPU, também na máquina 3, separados por cada operação realizada pelo algoritmo.

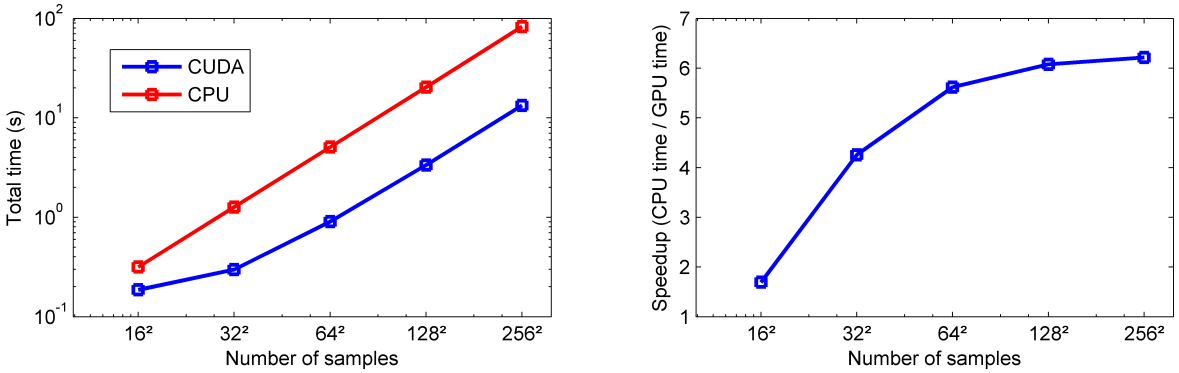

Figura 5.1: Tempo total das implementações em CPU e CUDA, e speedup geral alcançado na máquina 3

5.3.5 Interpretação dos resultados

A tabela 5.3 claramente mostra que o speedup geral menor nas máquinas 2 e 3 é devido às CPUs mais lentas (quando comparado à máquina 1), e não as GPUs, dado que os tempos de GPU são todos aproximadamente equivalentes. Entretanto, na Tabela 5.4 podemos observar que os speedups são bem diferentes entre as máquinas dependendo de qual parte da computação esteja sendo considerada. O cálculo de Phi e adaptação do gradiente tiverem um desempenho visivelmente melhor na Tesla C2070 e na GeForce GTX 590, enquanto o oposto aconteceu com a operação de mínimos quadrados.

Isto pode ser possivelmente explicado pela GeForce GTX 550 Ti possuir menos núcleos de computação mas taxas de *clock* mais altas (veja Tabela 5.5), o que causaria um impacto diferente em códigos que realizem tipos diferentes de operações. Em particular, computações com uso pesado de acessos a memória global (como é provavelmente o caso da operação de mínimos quadrados como implementada dentro da biblioteca CULA, dado que é necessário executar várias operações envolvendo os elementos de uma matriz grande) devem beneficiar-se de uma taxa mais alta de clock para acesso à memória global.

Também é interessante notar que a estratégia proposta conseguiu atingir um speedup mensurável mesmo com um sistema nebuloso pequeno (apenas 2 entradas e 35 regras), explorando paralelismo dentro do conjunto de amostras. Isto é em contraste ao modelo da GPU-FNN descrito em JUANG; CHEN; CHENG (2011), onde um sistema com apenas 2 entradas obteve desempenho *pior* na GPU do que na CPU devido ao paralelismo nesse modelo ser explorado primariamente nas dimensões de entrada, e não nas amostras. A GPU-FNN, por outro lado, obteve speedups muito maiores que os alcançados neste trabalho em casos com entradas de alta dimensionalidade; esses casos não foram especificamente testados com o algoritmo desenvolvido para esta pesquisa porque o algoritmo proposto trata as dimensões das entradas serialmente, focando no paralelismo entre as amostras. Desta forma, um problema de alta dimensionalidade seria pouco apropriado e é evidente que os ganhos seriam bem menores.

Finalmente, os valores brutos de speedup podem ser um pouco enganadores a princípio porque nenhum paralelismo de CPU foi explorado nestes testes. Porém, em um cenário típico de jogos eletrônicos modernos, tanto tempo de CPU quanto de GPU são escassos dependendo do que está sendo processado na simulação do jogo em um determinado momento. Sendo assim, mesmo que uma implementação paralela em CPU pudesse reduzir o valor bruto do ganho de velocidade e colocar CPU e GPU na mesma categoria de tempos, ainda é bastante útil que seja viável alocar ambos os recursos (CPU e GPU) eficientemente para um dado problema e escolher entre eles dependendo das necessidades específicas da aplicação e do contexto em que ela se encontra.

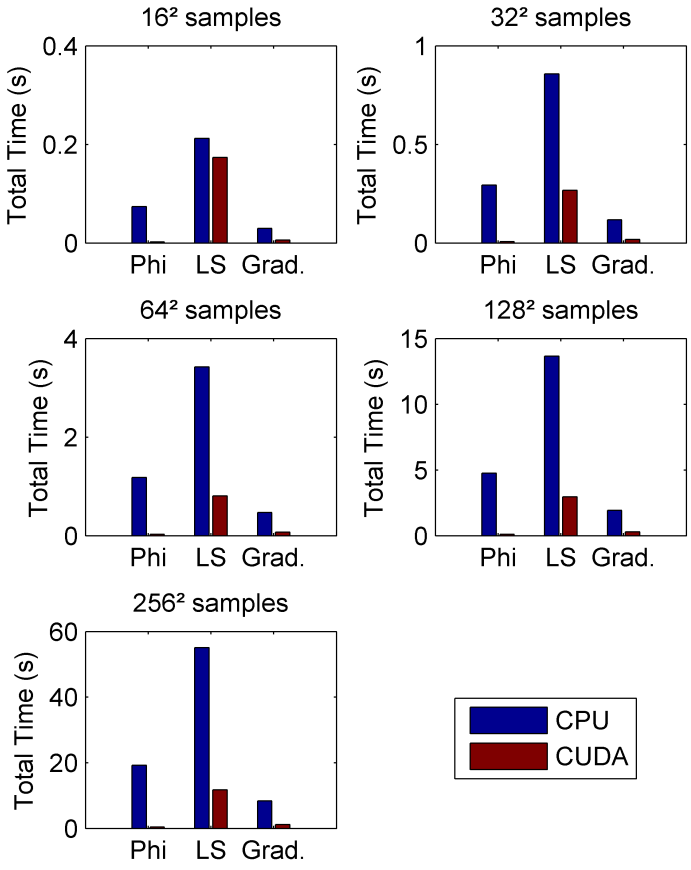

Figura 5.2: Comparação dos tempos de CPU e GPU na máquina 3, para cada conjunto de amostras, separados por operação

6 AGENTES NEBULOSOS TSK DINÂMICOS EM JOGOS

6.1 Introdução

Este capítulo traz um resumo dos experimentos realizados para a aplicação dos métodos estudados em tempo real dentro de jogos eletrônicos reais, além dos resultados advindos destes experimentos. Foram desenvolvidos dois jogos simples para realização dos testes: a seção 6.3 define o primeiro modelo de jogo que foi criado, seus problemas e o motivo de seu eventual abandono, enquanto a seção 6.4 trata da modelagem e criação do segundo jogo, considerando as deficiências do primeiro. A seção 6.4.3 apresenta a evolução dos agentes desenvolvidos e treinados e seu desempenho no segundo jogo, juntamente com uma análise dos resultados.

6.2 Tecnologia

Os dois casos de jogos utilizados para experimentos neste trabalho foram desenvolvidos utilizando-se o motor de jogos (*Game Engine*) Unity3D com a linguagem C#. Esta plataforma é, na data da realização desta pesquisa, amplamente reconhecida e utilizada para o desenvolvimento rápido e fácil (porém potencialmente poderoso) de jogos interativos 2D e 3D, inclusive com suporte a variados tipos de plataformas–PCs com Windows e OS X, consoles de jogos e dispositivos móveis como aqueles baseados nas plataformas Android, iOS, Blackberry e Windows Phone. Mais informações sobre o motor de jogos Unity3D podem ser encontrados em seu *website*

oficial[1].

Os algoritmos de treinamento do sistema nebuloso TSK foram implementados em uma biblioteca dinâmica (DLL) e acessados pelos jogos através da funcionalidade de chamadas a DLLs externas da plataforma Unity3D. Para o desenvolvimento e testes da modelagem dos agentes para os jogos, foi utilizada apenas a implementação em CPU dos algoritmos para facilitar o desenvolvimento e execução dos mesmos. Um teste final utilizando a implementação em GPU foi realizado ao fim do desenvolvimento dos modelos de agentes, cujos resultados podem ser encontrados na seção 6.5.

6.3 Primeira experiência: um jogo de corrida simples

A primeira ideia de jogo desenvolvida para aplicação e testes com um método de aprendizado dinâmico baseado em sistemas nebulosos TSK foi de um jogo simples de corrida em circuito. Este jogo é definido por uma pista de corrida em circuito, onde um carro por vez corre pela pista e o objetivo da competição é terminá-la no menor tempo possível.

O objetivo inicial para este teste foi definido, então, como a criação de um agente de IA que controla um carro no jogo baseado em um sistema TSK com entradas colhidas do ambiente. O treinamento desse agente seria feito observando-se o comportamento de um jogador humano por algumas voltas, a fim de observar seu comportamento, coletar amostras e utiliza-las para aplicação no método de aprendizado.

[1] http://www.unity3d.com

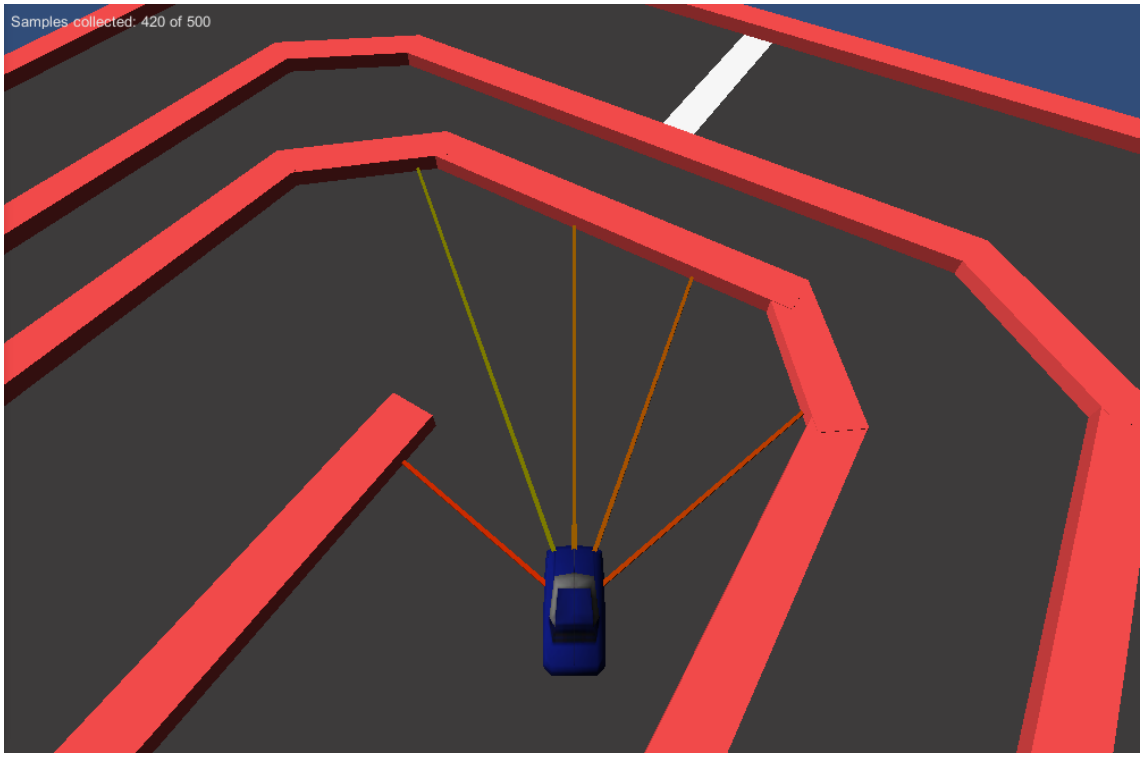

Figura 6.1: Imagem do protótipo desenvolvido para o jogo de corrida

6.3.1 Modelagem do problema

A modelagem do sistema desenvolvida para o controle de um carro em um circuito de corrida neste trabalho foi como descrita a seguir. Esta modelagem foi desenvolvida como uma primeira tentativa para avaliação da viabilidade da aplicação do sistema neste tipo de jogo. Outras modelagens não foram testadas devido ao jogo como um todo ter sido descartado, conforme explicado na seção 6.3.3.

- *Variáveis de Entrada*: Todas as entradas do sistema foram representadas por "sensores"de distância colocados na frente do carro, simetricamente, cada um a um ângulo diferente da direção frontal do veículo. Esta abordagem é semelhante à utilizada para controle de um robô inteligente em CONCEIçãO MOTA

(2010). Neste caso foram realizados testes com 5, 7 e 9 sensores à frente do carro. A leitura dos sensores virtuais dentro do ambiente do jogo é realizada através de um *Raycast* com um raio partindo do meio da frente do carro e direcionado no ângulo de cada sensor; a resposta de distância é a distância até uma colisão com o cenário, conforme reportada pelo módulo de física do motor do jogo, até um valor máximo considerado relevante pelo sistema.

- *Conjuntos de Entrada*: Cada variável de entrada foi dividida em um certo numero de conjuntos, sendo que as variáveis relativas aos sensores com ângulos mais ao centro possuíam mais conjuntos do que as relativas aos ângulos laterais, devido a haver mais variação de valores para estes sensores enquanto o carro se locomove pelo cenário.

- *Variáveis de Saída:* As duas saídas do sistema correspondiam aos eixos X e Y de controle do jogo, equivalentes às entradas de setas do teclado ou eixos do joystick no caso de um jogador humano. Neste caso, o eixo X (variando entre -1 e +1) controla a direção do volante do carro para a esquerda ou direita, enquanto o eixo Y (também variando de -1 a +1) é responsável pela aceleração, frenagem e marcha a ré do carro controlado.

6.3.2 Metodologia

A metodologia utilizada para o treinamento do agente inteligente foi a seguinte: uma pista era definida e o jogador humano era colocado para correr nesta pista. A cada passo de um intervalo de tempo fixo, desde que o carro estivesse em movimento, uma amostra do comportamento do jogador era coletada, na forma de uma leitura de todos os sensores virtuais de distância do veículo (entradas) juntamente com os comandos do jogador no eixo X e Y do teclado ou joystick (saídas). Estas amostras eram armazenadas até uma certa quantidade, quando o sistema passava a estar "pronto para ser treinado". Após um treinamento ser realizado, o

controle do carro no jogo passava para o agente de IA e seu comportamento era observado.

A quantidade de sensores no carro, os ângulos entre eles, o valor considerado máximo para distância, a quantidade de amostras coletadas e o tempo entre elas, assim como os parâmetros do algoritmo de treinamento em si, foram variados ao longo dos testes realizados. Os resultados, porém, pouco variaram em função dessas variações, conforme pode ser visto na seção 6.3.3. Testes em ambientes mais complexos com objetos dinâmicos ou bifurcações também não foram estudados devido ao jogo ter sido descartado após os resultados dos testes iniciais.

6.3.3 Resultados

Vários testes foram realizados com a modelagem do jogo de corrida conforme descrito nas seções 6.4.1 e 6.3.2, variando todos os parâmetros disponíveis em busca da melhor solução possível para o problema em questão com o ambiente mais simples possível. Os resultados, porém, foram nada animadores.

De maneira geral, em praticamente todos os casos de treinamento e teste o comportamento do agente treinado pode ser descrito da seguinte forma: na reta inicial da pista de corrida, o carro controlado pela IA seguia o trajeto da pista, eventualmente realizando até pequenas correções laterais de rota. Porém, ao atingir a primeira curva, o comportamento se mostrava extremamente errático: o carro parava, dava marcha a ré, virava o volante totalmente para um dos lados, e quase nunca conseguia realizar mesmo a primeira curva do trajeto corretamente. Ao mesmo tempo, a saída do Erro Médio Quadrático do sistema estava muito baixa, indicando que o treinamento foi um sucesso.

Uma investigação sobre o funcionamento do sistema revelou que as saídas do controlador estavam não só erráticas e incorretas, mas sua magnitude também estava muito longe do domínio esperado para o problema. Enquanto o treinamento havia sido sempre realizado com a apresentação de saídas entre -1 e +1, as saídas do sistema treinado, quando próximas da curva, em sua maioria beiravam a casa das centenas de unidades, quando não dos milhares.

Diante desses resultados extremos, a primeira hipótese levantada foi, então, a de que deveria haver algum problema na implementação do algoritmo. Para estudar essa possibilidade, fez-se mais uma vez o mesmo procedimento de coleta de amostras a partir do jogador humano, treinamento e observação do agente de IA. Durante a observação do agente, as entradas e saídas do sistema treinado foram anotadas. Após esta observação, as amostras coletadas para o treinamento foram importadas também no software MATLAB, uma ferramenta amplamente utilizada para pesquisa matemática e científica, e que possui implementações de vários algoritmos de Inteligência Computacional. Estas amostras importadas foram então treinadas dentro do MATLAB utilizando um sistema ANFIS com as configurações padrão do pacote, o que corresponderia ao resultado mais próximo possível da técnica empregada no jogo.

A posterior apresentação de alguns casos de teste anotados durante a execução do agente no jogo revelaram que o sistema ANFIS treinado pelo MATLAB estava dando como saídas valores bastante parecidos com o sistema do jogo. Esta constatação evidenciou fortemente que o problema do agente não deveria estar na implementação do algoritmo, mas sim na modelagem do problema.

Para investigar mais a fundo esta nova hipótese, foi utilizada uma funcionalidade do pacote de Lógica Nebulosa do MATLAB que permite visualizar um gráfico 3D de parte de um sistema nebuloso: por ser um gráfico 3D, deve-se escolher duas

variáveis de entrada e uma de saída de cada vez para se observar uma superfície que representa parte do funcionamento do sistema considerado. A Figura 6.2 mostra alguns dos gráficos obtidos para o sistema treinado no MATLAB utilizando-se a modelagem desenvolvida para o jogo. Alguns pares de variáveis de entrada foram escolhidos arbitrariamente para ilustrar regiões do domínio com comportamento particularmente errático.

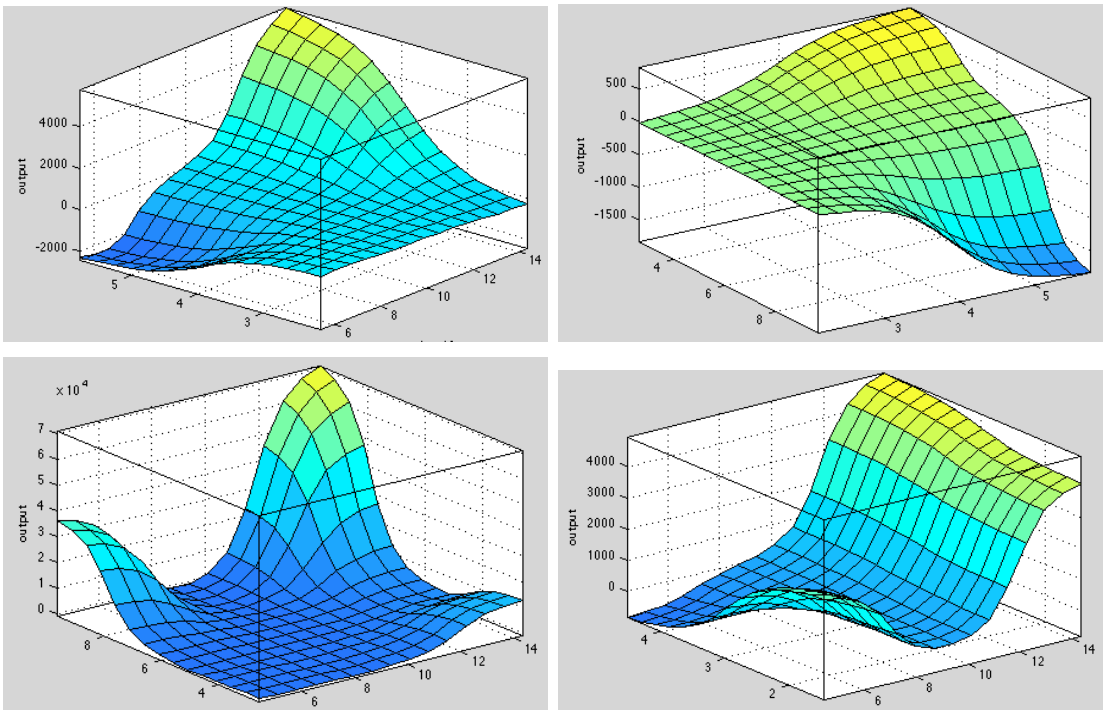

Figura 6.2: Gráficos de superfície da função treinada pelo ANFIS no MATLAB para o jogo de corrida. Podemos perceber como algumas regiões mantém a saída próxima de zero enquanto outras geram resultados muito longe do esperado para o contexto do sistema.

Como pode ser visto na figura, apenas uma pequena região (mais central) do espaço de entradas resultava em saídas válidas (próximas de 0). A maioria do espaço de entradas resultava em saídas distantes de 0, especialmente nas bordas. Ou seja, o sistema realmente estava atingindo um estado de treinamento em que a maior parte do espaço de entradas possíveis resultava em uma saída considerada

inválida pelo contexto em que este sistema seria utilizado.

Essa informação juntamente com o fato de que o erro do treinamento encontrava-se constantemente baixo (de duas a três ordens de grandeza menor que os valores de saída esperados) passou a indicar com bastante certeza que o problema era falta de cobertura do espaço de amostras, e que a metodologia aplicada não se mostrava adequada para realizar um treinamento satisfatório.

A perspectiva de reavaliar toda a modelagem do problema e desenvolver uma nova que contemplasse a geração de um conjunto de amostras que cobrisse satisfatoriamente todo o espaço de entradas possível, além da demora na realização dos testes e relativa complexidade do jogo em questão, levaram finalmente à decisão de se abandonar este caso de teste e à criação de um segundo jogo, que fosse mais simples e contemplasse desde o começo uma modelagem robusta e capaz de garantir que todo o espaço de entradas estivesse adequadamente coberto pelas amostras utilizadas no treinamento. Estes requisitos, como foi provado nos experimentos com o jogo de corrida, se fazem necessários para um bom desempenho no tipo de sistema adotado (ANFIS).

6.4 Destilando o problema: Inerpong

A busca por um modelo de jogo mais simples levou a uma volta aos modelos de jogos mais primordiais, dentre os quais o jogo PONG, lançado em 1972, é o mais clássico. Este jogo, originalmente, consistia em um ambiente contendo uma bola e duas "raquetes", uma em cada lado da tela (esquerdo e direito), sendo cada uma controlada por um jogador diferente. A bola movia-se pelo cenário e era rebatida pelos jogadores e pelas "paredes" formadas pelas laterais superior e inferior da tela, e um ponto era ganho por um jogador quando o outro falhasse em rebater a bola,

deixando-a escapar pela sua borda. A Figura 6.3 mostra como era a tela deste jogo.

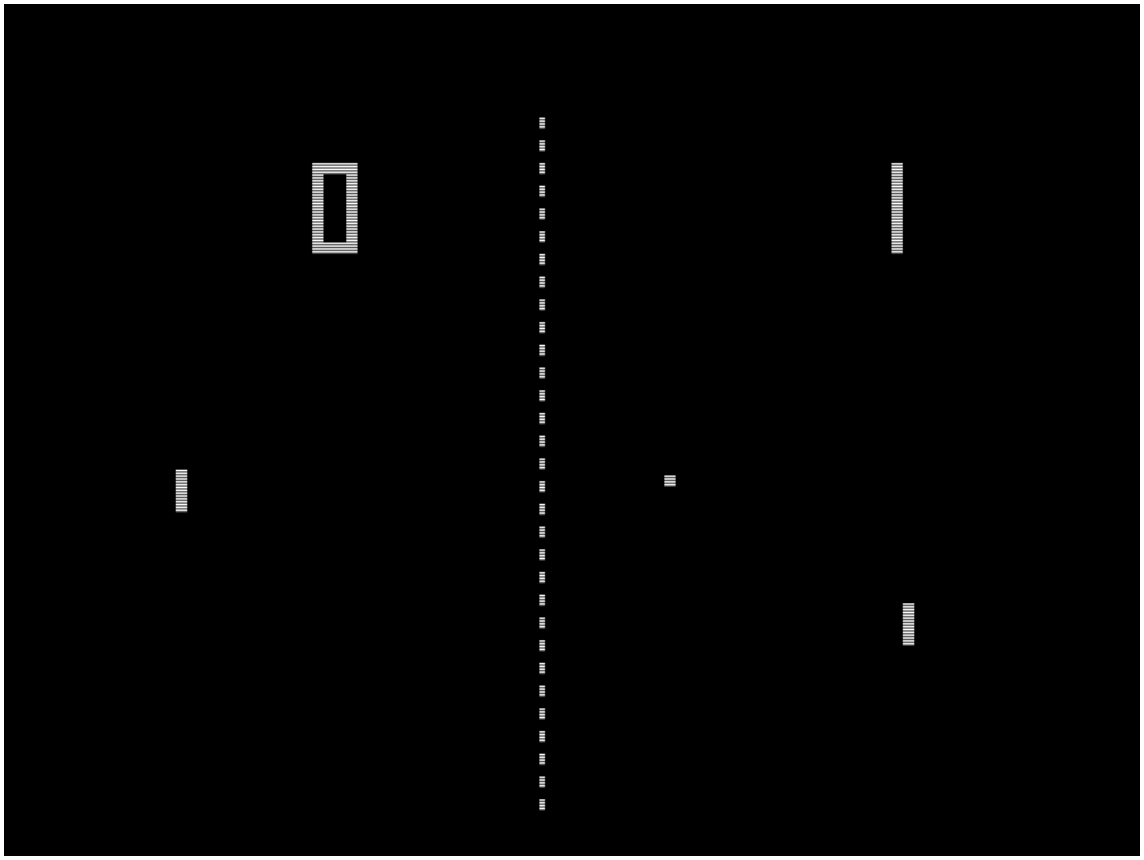

Figura 6.3: Imagem do jogo PONG original, lançado em 1972

A simplicidade das regras e, por conseguinte, da modelagem matemática do PONG, motivaram o desenvolvimento de um segundo protótipo de jogo baseado nestas mecânicas. O jogo então desenvolvido e nomeado *Inerpong* consiste em uma variação de PONG com os seguintes diferenciais:

- O movimento das raquetes dos jogadores é feito com inércia, ou seja, o jogador controla diretamente a aceleração lateral da raquete e não sua velocidade como no original;
- As raquetes dos jogadores estão localizadas nas laterais superior e inferior da

tela, e não na esquerda e na direita.

A motivação para estas modificações foi dificultar a modelagem matemática do problema, dado que no jogo original bastaria apenas controlar diretamente a raquete para mante-la na mesma altura da bola e há bastante tempo para realizar o movimento. Já neste caso, as diferenças tornam o jogo um pouco mais difícil, especialmente para um agente controlado por computador, porque adicionam um fator de estratégia: como a bola move-se com uma velocidade razoável e o movimento das raquetes é "pesado", para a realização de uma boa jogada quase sempre é necessário inferir onde a bola estará daqui a alguns momentos, considerando também o rebatimento nas bordas do cenário. A Figura 6.4 mostra a aparência do *Inerpong*.

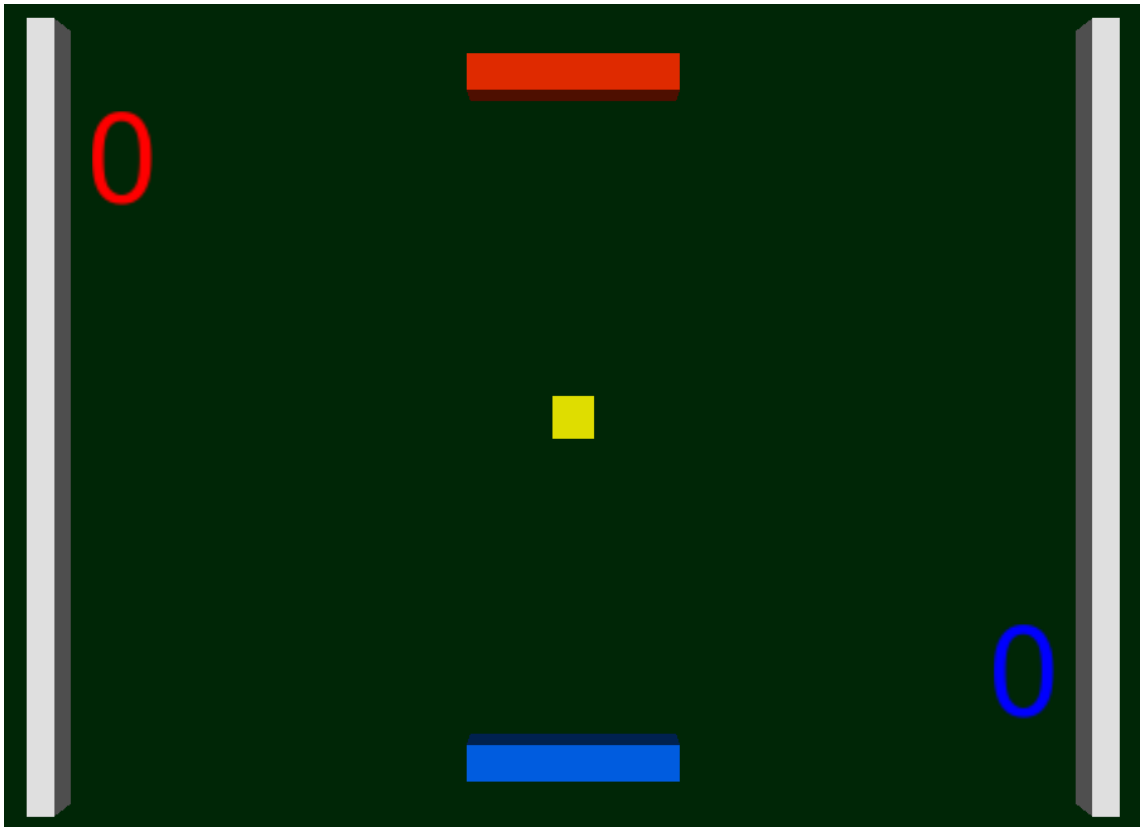

Figura 6.4: Imagem do segundo protótipo desenvolvido, uma variação de PONG

6.4.1 Modelagem do problema

A modelagem do sistema criado para a geração dinâmica de um oponente no Inerpong variou ao longo do testes (de acordo com o especificado na seção 6.4.3), mas podemos dizer que, de maneira geral, ela possui as seguintes características:

- *Variáveis de Entrada*: Algumas variáveis do ambiente do jogo foram disponibilizadas para utilização como entradas do sistema treinado (apesar de nem todas serem utilizadas simultaneamente, como pode ser visto na descrição da evolução dos agentes desenvolvidos na seção 6.4.3). São elas:
  - Posição absoluta da bola ($bola_x$ e $bola_y$), normalizada no domínio [0...1] em relação ao espaço do jogo, sendo que 0 na coordenada X representa o canto esquerdo da tela, e 0 na coordenada Y representa o canto mais próximo do jogador considerado;
  - Direção do movimento da bola, representada por um vetor 2D normalizado ($bola_x^v$ e $bola_y^v$);
  - Posição *objetivo* do jogador considerado (meu_x^{obj}), normalizada no domínio [0...1];
  - Posição absoluta do adversário ($outro_x$), normalizada no domínio [0...1];
  - Posição da bola em relação ao jogador ($bola_x^{rel} = bola_x$ - meu_x);
  - Posição do adversário em relação ao jogador ($outro_x^{rel} = outro_x$ - meu_x).
- *Conjuntos de Entrada*: A quantidade de conjuntos em que cada variável foi dividida inicialmente variou ao longo dos testes e será descrita com mais detalhes na seção 6.4.3, mas em todos os casos a divisão do domínio das variáveis em conjuntos foi feita de maneira uniforme.

- *Variáveis de Saída:* Dois tipos de saídas foram considerados ao longo dos testes: a *direção do movimento* do jogador ($dirMovimento$), que corresponde à entrada do jogador humano no teclado ou joystick; e a *posição alvo* para onde o jogador deve se movimentar ($posAlvo$).

6.4.2 Metodologia

Para ser possível comparar a evolução dos diferentes agentes criados, assim como realizar uma validação inicial do método de treinamento, foi desenvolvida uma IA ingênua (que chamaremos apenas de "agente burro") para servir de base de comparação. Seu funcionamento consiste apenas em verificar se a bola está à esquerda ou à direita, e tentar mover-se com força total para a direção correspondente (com uma pequena zona "morta"no centro). Como a função matemática que define este comportamento é bastante simples, o método desenvolvido deveria ser capaz de aproximá-la com facilidade. Além disso, pode-se observar o resultado de jogos entre o agente burro e outros com aprendizado baseado nele, utilizando diferentes parâmetros de treinamento, para que se descubra os melhores parâmetros possíveis.

A metodologia dos testes consistiu nas seguintes etapas:

- 1. Criação de um banco de amostras do comportamento do agente burro, dividindo-se o espaço das variáveis de entrada em uma grade uniforme;

- 2. Uma rodada de coleta de amostras do comportamento dos jogadores em uma partida do jogador humano contra o agente controlado por IA (inicialmente o agente burro, posteriormente os agentes treinados). As amostras são coletadas a um intervalo fixo de tempo, desde que a bola esteja se movendo, até atingir um número predefinido de amostras;

- 3. Modificação do banco de amostras utilizando-se as amostras coletadas no passo anterior juntamente com alguma estratégia de modificação;

- 4. Treinamento (ou re-treinamento) do sistema utilizando-se o estado atual do banco de amostras;

- 5. Substituição do agente não-humano pelo sistema treinado anteriormente;

- 6. Repetição dos passos 2-5 por 10 minutos, até que o agente treinado tenha evoluído algumas vezes e aprendido com o jogador humano;

- 7. Substituição do controlador humano pelo agente burro e reinicialização do placar;

- 8. Observação de uma partida do agente burro contra o último agente treinado até que algum deles tenha atingido 50 pontos.

Algumas exceções a esse processo se aplicam nos testes iniciais, conforme explicado na seção 6.4.3.

6.4.3 Evolução dos agentes desenvolvidos

Os testes realizados com o desenvolvimento de agentes inteligentes dinâmicos para o Inerpong seguiram uma lógica iterativa, onde o resultado de um teste levava a um refinamento do modelo e do método de treinamento para o próximo teste a ser realizado. O texto das seções 6.4.3.1 e 6.4.3.1 segue este processo, relatando as modificações e resultados de cada teste em ordem cronológica.

6.4.3.1 Agentes baseados no agente burro

O primeiro passo no estudo dos agentes para o Inerpong foi o desenvolvimento de um agente com um sistema nebuloso TSK que reproduzisse o comportamento do "agente burro"desenvolvido em lógica simples. A criação deste agente tinha como objetivos básico validar o funcionamento do método de aprendizado, já que a função matemática que rege o comportamento do agente burro (explicada na seção 6.4.2) é bastante simples e deveria ser aproximada com facilidade.

Para o desenvolvimento deste agente foram utilizadas, inicialmente, 4 variáveis de entrada: $bola_x$, $bola_x$, meu_x e $outro_x$. Como o comportamento do agente burro depende unicamente da diferença entre $bola_x$ e meu_x, estas variáveis deveriam ser mais que suficientes para modela-lo; além disso, as duas variáveis redundantes também verificariam a robustez do método frente a excesso de informação. Todas as variáveis utilizadas foram inicializadas com 3 conjuntos distribuídos uniformemente.

O treinamento do sistema TSK foi realizado em apenas uma iteração (um passo de Mínimos Quadrados e um passo do método do gradiente). Além disso, para este agente de referência não houve treinamento interativo ou dinâmico, ou seja, foi feito apenas um treinamento baseado no conjunto de amostras organizadas em grade obtidas a partir do agente burro original.

O primeiro agente treinado com esta estratégia, ao ser colocado para jogar contra o agente burro original, perdeu por um placar de 17x50. Um próximo teste trocou as variáveis absolutas $bola_x$ e $outro_x$ por suas equivalentes relativas $bola_x^{rel}$ e $outro_x^{rel}$ e também perdeu, desta vez por 14x50 (o que podemos considerar como um resultado quase equivalente).

Uma observação detalhada do comportamento do agente treinado revelou que

ele estava perdendo por ser mais *lento* que seu adversário: o sistema treinado foi capaz de aprender adequadamente o comportamento desejado no jogo em relação à *direção* do movimento, mas sua *intensidade* nunca chegava aos extremos -1 e +1.

Frente a esta constatação, o agente treinado foi revertido para utilizar entradas absolutas e modificado apenas para considerar como saída apenas a direção: ou seja, a resposta do sistema, antes de ser passada ao controle da raquete do jogo, era arredondada para -1, 0 ou +1. Após esta mudança, um teste com o novo agente contra o agente burro original resultou em um placar de 50x49, o que pode ser considerado um forte empate técnico e um forte indício de que a implementação do sistema de treinamento estava correta, além de ser também um indício de que a modelagem do jogo era propícia ao método utilizado.

Para realizar mais alguns testes com este agente, modificou-se novamente as variáveis de entrada para suas versões relativas, juntamente com o arredondamento da saída. Este teste resultou em um placar de 50x39, seguido de outro de 45x50 e outro de 50x49, que ainda podem ser considerados como um ótimo resultado.

Seguindo-se o mesmo modelo, um próximo teste modificou a quantidade de conjuntos de cada variável: a variável $ball_y$ passou a contar apenas com 2 conjuntos, enquanto a variável $ball_x$ passou a ter 5 conjuntos. Esta modificação foi realizada para verificar a hipótese de que era relevante considerar como mais granular o espaço horizontal da área de jogo do que o espaço vertical. O resultado deste treinamento, ainda com apenas uma iteração, foi um agente que ganhou do agente burro com um placar de 50x39.

Finalmente, a partir do último modelo criado foi feito um teste com 20 iterações de treinamento. Este agente acabou por perder com um placar de 38x50. Uma possível explicação para este comportamento é que o excesso de iterações causou um

overtraining no sistema, tornando-o especializado demais nas amostras utilizadas no treinamento e dificultando sua generalização para outros cenários.

6.4.3.2 Agentes baseados no comportamento do jogador

Após a validação do método de treinamento através da criação de um agente inteligente que copia as ações do agente burro simples, iniciou-se uma longa sequência de testes com aprendizado de comportamentos de um jogador humano. Os testes iniciaram-se com os melhores parâmetros deduzidos para o treinamento ao longo do desenvolvimento do agente de referência, ou seja: variáveis de entrada de posição relativas, consideração apenas da direção na saída do sistema, 10 iterações no treinamento, 5 conjuntos para a variável $bola_x$, 2 conjuntos para $bola_y$ e 3 conjuntos para meu_x e $outro_x$. É importante também ressaltar que o jogador humano considerado para os experimentos (o próprio desenvolvedor) possuía amplo conhecimento no funcionamento das regras do jogo.

O primeiro teste realizado com coleta de amostras do jogador humano consistiu no treinamento de um sistema baseado puramente em um conjunto de amostras coletados do jogador, ou seja, sem a utilização do agente burro para geração de um banco de amostras inicial com cobertura em todo o espaço de amostras. O resultado esperado para este caso, correspondendo ao que ocorreu com o primeiro protótipo (descrito na seção 6.3.3) era que o comportamento do agente fosse errático na maioria das situações por falta de cobertura no domínio completo das variáveis de entrada. Após a realização do treinamento, pôde-se observar que o resultado foi exatamente esse–apesar de reproduzir o comportamento do jogador em poucas situações específicas, na maior parte do tempo o sistema retornava saídas aparentemente aleatórias, em geral jogando a raquete do jogador com força total para algum dos lados da área de jogo, ou alternando direções de maneira completamente desconexa

com a situação atual do jogo.

Após este teste inicial, foi implementado o protocolo completo de treinamento e teste descrito na seção 6.4.2. Cada rodada de coleta de amostras era executada até que 100 amostras fossem obtidas, momento em que era refeito o treinamento do sistema. O método inicial implementado para a modificação do banco de amostras consistia em, para cada amostra coletada do jogador, simplesmente substituir a amostra mais próxima no banco (considerando distância euclidiana sobre as dimensões de entrada).

A realização de uma série de testes seguindo este modelo resultou em um agente que reproduzia satisfatoriamente o comportamento do agente burro, e aparentava ser levemente diferente em algumas situações, influenciado pela substituição de algumas amostras coletadas do jogador. Entretanto, estas situações não o levavam e ter o mesmo comportamento do jogador, e em alguns casos resultavam em um desempenho até pior do que o do agente burro original. Após algumas observações, percebeu-se que o tipo de situação que o agente treinado tinha dificuldades para aprender era justamente aquele onde o jogador deve antecipar-se ao rebatimento da bola em uma das paredes, inferindo sua futura mudança de direção. Como o modelo utilizado até então não contemplava a direção do movimento da bola, foi levantada a hipótese de que o método falhava em aprender o comportamento correto devido a falta de informação nas amostras de exemplo.

Sendo assim, foram adicionadas ao modelo de treinamento as variáveis $bola_x^v$ e $bola_y^v$, que representam a direção atual de movimento da bola. Para começar os testes com um modelo simples que detectasse apenas a direção aproximada do movimento (esquerda/direita e cima/baixo), estas duas variáveis foram configuradas com apenas 2 conjuntos cada. Neste momento, o total de regras definidas para o sistema (que é igual ao produtório da quantidade de conjuntos de todas as variáveis

de entrada) era igual a 360–valor ainda abaixo do limite de 512 regras estabelecido pelo método desenvolvido no capítulo 5.

Uma nova rodada de treinamento seguindo a metodologia completa de testes resultou em uma derrota de 42x50 contra o jogador burro. Observando-se o agente treinado, este parecia ter aprendido algumas manobras ofensivas tímidas mas não muitas defensivas (principalmente previsão de rebatimento da bola nas paredes). Neste momento, o total de amostras no banco (definido pela divisão da grade de amostragem do agente burro inicial) era 3375. Este número aumentou consideravelmente depois da adição das duas novas variáveis, e como cada rodada de treinamento coletava apenas 100 amostras, o tempo necessário para substituição de amostras suficientes no banco para refletir todos os comportamentos do jogador seria excessivamente grande.

Essa constatação levou ao desejo de implementar um novo método de modificação do banco de amostras que, para cada amostra coletada do jogador humano, fosse realizada uma modificação que influenciasse várias amostras simultaneamente no banco, dependendo da similaridade com a amostra coletada. Esta ideia veio diretamente do raciocínio de que o universo das variáveis do jogo é contínuo, e situações pouco diferentes levam a reações parecidas por parte do jogador. Além disso, este tipo de modificação afetaria a superfície da saída do sistema de uma maneira mais suave, evitando pontos de máximo ou mínimo grandes em casos em que a amostra substituída tivesse uma resposta original muito diferente da coletada.

Para a implementação deste "raio de influência", foi utilizada a distância euclidiana entre a nova amostra coletada e todas as amostras atualmente no banco de amostras, considerando-se as dimensões de entrada do sistema. Considerando que d seja esta distância para um certo par de amostras e R um raio máximo de influência, foi utilizada uma *função de suavização* definida pela equação 6.1.

$$F(d) = max\left(0,1 - \left(\frac{d}{R}\right)^{0.5}\right) \tag{6.1}$$

O objetivo desta função é definir, para cada amostra do banco, qual será a força que uma amostra coletada terá sobre ela. Seu valor de retorno deverá estar localizado no intervalo [0...1], onde 0 significa "nenhuma influência"e 1 significa "influência total". Este valor é aplicado diretamente na modificação da amostra no banco, fazendo com que ela seja substituída por uma interpolação linear entre o valor atual e o valor da nova amostra coletada, onde o peso da interpolação é definido por $F(d)$.

O próximo treinamento com o jogador humano levou em consideração esse novo método de modificação do banco de amostras, juntamente com uma nova consideração para acelerar o processo de treinamento: uma regra de simetria horizontal na coleta de amostras foi implementada, de modo que cada amostra coletada causa a aplicação do processo de substituição duas vezes (para ela própria e para uma situação análoga correspondente ao espelhamento horizontal da área de jogo). Após o treinamento, o resultado de uma partida contra o agente burro foi uma vitória por 50x47. Apesar de ser uma melhora frente ao último resultado, e uma observação revelar que comportamentos do jogador estavam sendo consistentemente reproduzidos, o agente treinado estava sendo prejudicado por copiar comportamentos "errados"do jogador que foram registrados durante a partida.

Dada essa observação, o próximo mecanismo implementado no treinamento foi um de *reforços positivos e negativos* sobre cada amostra coletada do jogador, da seguinte forma: Foi definido um valor de *força* para cada amostra coletada do jogador, inicialmente em 1. Caso, durante a coleta das amostras, o jogador rebata a bola com sucesso em um dado momento, a força das últimas amostras coletadas dentro de uma certa janela de tempo é multiplicada por um fator (maior do que 1)

proporcional à diferença de tempo entre o momento da coleta da amostra e o instante atual. Da mesma forma, caso o jogador perca uma bola (ponto do adversário), o mesmo procedimento é realizado com um outro fator (menor do que 1). O valor dessa força é então multiplicado pelo valor retornado pela função de suavização para formar o fator final da interpolação realizada em cada amostra do banco.

Nos testes, foram utilizados os valores de 0.5 segundos para a janela de tempo, 0.5 como fator mínimo de reforço negativo e 2.0 como fator máximo de reforço positivo. O resultado final após a aplicação deste método foi animador: uma vitória por 50x40, que, após ter sido deixado que o jogo continuasse, resultou em uma nova e consistente vitória por 100x81, ou seja, um desempenho 25% maior que o agente burro original. É importante ressaltar que esta foi a primeira ocasião em que o agente treinado teve um desempenho consistentemente melhor que o do agente burro, devido à utilização de comportamentos aprendidos com o jogador humano.

Mesmo assim, o agente treinado ainda parecia generalizar pouco os comportamentos em relação ao ângulo da bola. Ou seja, em algumas situações a IA se antecipava ao rebatimento da bola nas paredes, e em outras pouco diferentes não. Para tentar contornar este problema foi feita uma pequena mudança: a função de suavização $F(d)$ passou a ser calculada considerando-se a metade da distância para a variável $bola_y^v$. Além disso, para evitar confusões próximo da região em que a velocidade vertical da bola é igual a 0, adicionou-se uma checagem que multiplica por 5 a distância nesta mesma dimensão caso os sinais da amostra recém-coletada e da amostra sendo modificada sejam diferentes; isto garantiria que a resposta gerada pelo jogador para uma situação de "bola movendo-se para cima"não influencia uma de "bola movendo-se para baixo"e vice-versa.

Depois destas mudanças, o resultado de uma partida contra o agente burro foi uma derrota de 43x50. Pôde-se perceber que as mudanças realizadas estavam

complicando desnecessariamente o problema e *aumentando* mais ainda a confusão do treinamento. Sendo assim, estas últimas mudanças foram revertidas.

O próximo experimento consistiu na mudança da função de influência das amostras, de modo a aumentar a área de influência, para a descrita na equação 6.2. A motivação para este teste é que a função definida anteriormente cai muito rapidamente, e a malha da distribuição das amostras era esparsa o suficiente para que a maioria das amostras fosse modificada com um peso muito pequeno. Esta nova função, por ter uma queda mais lenta próxima do centro, acelera o processo de adaptação do agente treinado.

$$F(d) = max\left(0,1 - \left(\frac{d}{R}\right)^{1.5}\right) \tag{6.2}$$

O resultado de um teste após esta modificação foi uma vitória de 50x33. Deixando o jogo continuar por mais tempo, o resultado final foi uma vitória por 100x89. Apesar do resultado final não ser tão consistente, ou tão bom, quanto o anterior, esta modificação foi mantida pois foi possível observar que o agente treinado realmente estava copiando melhor os comportamentos do jogador. Além disso, uma certa inconsistência no comportamento pode ser interessante para que o jogador controlado por IA pareça menos "mecânico".

O próximo passo, mais elaborado, envolve uma mudança nas variáveis do sistema. Percebeu-se que considerar a saída do sistema como apenas a direção para onde deve-se ir estava limitando as capacidades de aprendizado do mesmo, além de não ser um valor tão representativo da estratégia do jogo como, digamos, a *posição objetivo* em que o jogador deve estar para rebater a bola com sucesso. Sendo assim, a mudança consistiu basicamente em remover a variável meu_x do sistema como uma entrada, e a mudança da variável de saída para uma nova variável que

representava a posição para onde o jogador deve se mover, diferentemente da original, que representava apenas a *direção* para onde o jogador deve se mover. O treinamento inicial e o interativo foram, então, modificados de acordo para gerar adequadamente esta nova variável e remover a antiga; no treinamento com o agente burro, a posição objetivo é sempre a mesma da posição horizontal da bola, e no interativo foi definida como sendo a posição atual do jogador acrescida de sua aceleração atual multiplicada por um fator igual a 0.05.

Outras mudanças também foram feitas para melhorar o desempenho do sistema agora com uma variável a menos: a variável $bola_y$ agora contava com 3 conjuntos iniciais, e a malha de amostras foi dividida de uma forma mais fina em relação às variáveis restantes para uma melhor cobertura do espaço, num total de 1575 amostras.

O resultado deste teste final foi que o aprendizado durante o jogo tornou-se perceptivelmente mais rápido, e o agente treinado mostrava-se agressivo, porém com um comportamento menos mecânico ao longo da partida, o que foi uma descoberta bastante interessante, dado que um dos objetivos da utilização deste tipo de treinamento *on-line* é a geração de oponentes mais "humanos". Além disso, o resultado da partida contra a IA burra foi de 50x29 inicialmente e, posteriormente, 100x69.

6.4.4 Resumo dos resultados

A tabela 6.1 mostra um resumo dos modelos de agentes desenvolvidos para o InerPong e seu desempenho. Cada modelagem foi testada contra o agente burro 3 vezes após uma única rodada de coleta de amostras com o jogador humano. A cada modelo é dada uma pontuação calculada como a razão entre a pontuação final no jogo do agente desenvolvido e a pontuação final do agente burro. Os diferentes

agentes são listados por suas características, na ordem em que foram descritos na seção 6.4.3.

Em relação ao tempo de processamento, podemos dizer que, mesmo sendo processado em CPU, o treinamento não causou uma perda de performance mensurável durante o jogo. Porém, isto pode ser explicado também pela simplicidade do jogo desenvolvido, que passou todo o tempo dos testes sendo executado a mais de 300 FPS (quadros por segundo).

Treino com agente burro	X	X	X	X	X	X	X	X
Apenas direção na saída		X	X	X	X	X	X	X
Treino com jogador humano			X	X	X	X	X	X
Incluir direção da bola			X	X	X	X	X	X
Raio de influência com $F(d)$				X	X	X	X	X
Reforço positivo e negativo					X	X	X	X
$F(d)$ diferente para $bola_y^v$						X		
$F(d)$ diminui mais devagar							X	X
Saída é posição objetivo								X
Pontuação	0.30	1.02	0.84	1.06	1.25	0.86	1.12	1.44

Tabela 6.1: Resumo dos agentes treinados para o Inerpong

6.5 Resultados do treinamento em GPU

Após o desenvolvimento de todos os modelos para o InerPong, foi realizado um teste rápido de comparação de desempenho entre a versão de CPU e GPU do método de treinamento desenvolvido. Os testes foram realizados na Máquina 1 (como definido na Tabela 5.2), dentro do editor da Unity3D, utilizando a mesma resolução de tela, e alternando apenas o código de treinamento utilizado entre o método puro de referência em CPU e o algoritmo implementado em GPU. Os tempos (em segundos) e *framerates* (em quadros por segundo) médios após 5 rodadas de teste podem ser verificados na Tabela 6.2.

Etapa	CPU		GPU	
	Tempo	Framerate	Tempo	Framerate
Coleta de amostras	5.3173	179.03	6.301	163.98
Treinamento	5.2691	339.527	13.8608	244.28

Tabela 6.2: Comparação de desempenho entre treinamento com CPU e GPU no Inerpong

Podemos observar pelos números da tabela que o desempenho com a utilização do código de GPU foi *pior* do que o correspondente em CPU. Uma provável explicação para este cenário é a simplicidade do jogo: os gráficos 3D eram bem simples e o processamento de CPU, além da IA, se limitava a processar a entrada do jogador, movimentar uma raquete e simular a bola. Em uma situação em que tanto CPU quanto GPU têm tempo de sobra, não se poderia esperar um resultado muito expressivo. Em especial, através do *profiler* do editor da Unity3D pôde-se verificar facilmente que o gargalo de processamento do jogo, em ambos os casos, era a GPU, dado o processamento de gameplay muito simples que estava sendo realizado. Sendo assim, mover mais processamento da CPU para a GPU claramente faria com que o desempenho geral do jogo piorasse.

Podemos dizer, então, que o resultado deste teste foi inconclusivo sobre as vantagens do método de treinamento em GPU em jogos modernos com processamento intensivo tanto em CPU quando em GPU. Testes mais extensos deveriam incluir aumento artificial de carga de processamento em ambos os processadores, mas não foram realizados para este trabalho por questões de tempo; sendo assim, formam uma excelente base para trabalhos futuros.

7 CONCLUSÕES

O objetivo principal deste trabalho foi a avaliação da aplicação de novas técnicas, importadas da Inteligência Computacional, para a definição de IA em jogos. Enquanto existe uma intensa pesquisa acadêmica com técnicas alternativas à IA "clássica" baseada em lógica tradicional, estes estudos são, em sua maioria, voltadas a áreas não correlatas aos jogos eletrônicos. Esta pesquisa buscou, então, ajudar a abrir o caminho para que algumas dessas técnicas sejam consideradas na criação de IA para jogos.

Baseado nos resultados encontrados nos capítulos 5 e 6, podemos perceber que a não exploração destes métodos baseia-se primariamente em desconhecimento, dado que uma justificativa recorrente para este desinteresse é que métodos de aprendizado com Inteligência Computacional costumam ser muito lentos e ineficientes para aplicações em tempo real, por requererem uma grande quantidade de dados e processamento para obter um resultado satisfatório.

Em oposição a esse sentimento, o capítulo 5 mostrou que é possível, através do moderno *hardware* das placas de vídeo programáveis, utilizar todo o poder de processamento disponível em computadores e consoles de jogos modernos para possibilitar o processamento de técnicas de aprendizado mais complexas sem a necessidade de ocupar uma grande parte de tempo de CPU com esta tarefa. Este resultado tem valor simbólico não apenas na pesquisa de técnicas adaptativas de IA para jogos, mas também para várias outras aplicações dinâmicas que frequentemente deixam de lado soluções como o ANFIS por seu suposto alto custo computacional.

Ao mesmo tempo, o capítulo 6 mostrou que um sistema simples baseado em

aprendizado com lógica nebulosa pode ser utilizado para a geração e adaptação dinâmica de um agente num jogo eletrônico com um desempenho bastante interessante. Este resultado abre o leque de possibilidades para mais estudos neste sentido por um outro lado: aquele da noção de que a definição de sistemas menores com técnicas mais elaboradas, eventualmente tendo seus resultados combinados entre si, também se mostra uma opção viável para a criação de oponentes virtuais. Esta mesma noção foi explorada em ALVIM (2008), através da utilização de Máquinas de Estado Nebulosas para controle de emoções de um oponente virtual.

Podemos lembrar, também, que o tipo de sistema desenvolvido neste trabalho pode ser utilizado para treinamento dinâmico de agentes de IA em tempo de desenvolvimento. Ou seja, o próprio desenvolvedor do jogo (ou vários perfis de jogadores selecionados) pode jogar em um ambiente controlado contra um agente de IA adaptativo durante a fase de desenvolvimento do jogo, pelo tempo que for necessário até que se atinjam resultados satisfatórios. Os agentes treinados poderiam ser salvos em disco e posteriormente utilizados já treinados em tempo de execução contra jogadores de fato. Esta abordagem poderia ser especialmente interessante em tipos de jogos onde não é trivial definir uma série de regras lógicas que definem um bom comportamento devido à complexidade do jogo como, por exemplo, jogos de tiro multijogador.

Em relação à praticidade de uso do método de treinamento de sistemas TSK em GPU, é importante ressaltar que ele é independente do problema tratado, tanto na sua modelagem quanto na implementação feita para os testes referidos neste trabalho.

7.1 Trabalhos futuros

Dentre os trabalhos futuros considerados de especial relevância para esta pesquisa, podemos citar no âmbito dos métodos desenvolvidos para treinamento de sistemas nebulosos em GPU a realização de testes com sistemas mais complexos, especialmente com uma quantidade grande de dimensões de entrada e como o desempenho do método estudado se compara com outros similares nesses casos, particularmente a GPU-FNN desenvolvida por JUANG; CHEN; CHENG (2011).

Ainda em relação aos testes com código em GPU, seria bastante interessante investigar algumas otimizações de baixo nível que poderiam ser utilizadas para melhorar gradativamente a performance do método. Algumas alternativas incluem a utilização das memórias constantes e de textura para otimização de certos padrões de acesso e a modificação do segundo kernel de cálculo do gradiente para uma redução de dois níveis (ou seja, primeiro dentro de cada bloco de processamento e posteriormente agregando o resultado de cada bloco).

Além disso, dado o custo consideravelmente mais alto da operação de mínimos quadrados e o fato de seu retorno ser sempre um mínimo global, uma tentativa interessante seria intercalar uma passada de mínimos quadrados com vários passos do método do gradiente, possivelmente com um tamanho de passo pequeno para que seja feito um ajuste mais fino. Uma outra opção seria tentar realizar ajustes finos também nos conjuntos de entrada com o método do gradiente enquanto isso.

No domínio dos testes com jogos, seria desejável estender as ideias aqui apresentadas para outros gêneros de jogos, em especial com protótipos simples que "destilem"certas mecânicas e possam ser estendidos aos poucos até a validação em um jogo completo. Por exemplo, um jogo de tiro simples que consista em um "duelo"entre dois jogadores que iniciam um jogo em extremos opostos de uma sala, de

costas um para o outro, e devem apenas virar-se e atirar no oponente depois de uma contagem. Este jogo poderia ser estendido posteriormente para incluir movimentação, ambientes mais complexos, outros jogadores, até que se forme um jogo de tiro completo.

Finalmente, um experimento que vale a pena ser realizado é a extensão dos testes envolvendo a combinação das duas frentes de pesquisa exploradas neste trabalho: a comparação entre o treinamento dinâmico, dentro de um jogo, com sistemas nebulosos do tipo TSK tanto em CPU quanto em GPU. Conforme visto na seção 6.5, os testes preliminares realizados para este trabalho trouxeram um resultado pouco animador, porém sua realização foi bastante simples e deixou de considerar alguns fatores presentes em cenários reais, principalmente a carga pesada tanto na CPU quanto na GPU.

Neste caso, as métricas a serem avaliadas seriam, primariamente, relacionadas ao impacto que esta transição de processamento causaria no desempenho do jogo como um todo, em situações que ocupem os recursos da máquina de maneiras diferentes: uso leve de CPU e GPU (como foi o caso do teste realizado na seção 6.5), uso pesado de apenas uma das duas, e em uso pesado de ambas. Finalmente, seria interessante a análise de um mecanismo para balancear dinamicamente o processamento de IA entre GPU e CPU, dependendo das necessidades do jogo e da utilização das diferentes partes da máquina em um dado momento.

REFERÊNCIAS

ALVIM, L. G. M. **Uma máquina de estados nebulosa para um modelo de emoções aplicado a um personagem de um jogo eletrônico**. 2008. Dissertação (Mestrado Informática) — Instituto de Matemática, Instituto Tercio Pacitti de Aplicações e Pesquisas Computacionais, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2008.

ALVIM, L. ; CRUZ, A. J. O. A Fuzzy state machine applied to an emotion model for electronic game characters. In: IEEE INTERNATIONAL CONFERENCE ON FUZZY SYSTEMS, 2008. FUZZ-IEEE 2008. (IEEE WORLD CONGRESS ON COMPUTATIONAL INTELLIGENCE). 2008, Hong_Kong. **Proceedings …** [S.l.]: IEEE, 2008. p.1956 –1963.

ANDERSON, D. ; COUPLAND, S. Parallelisation of fuzzy inference on a graphics processor unit using the compute unified device architecture. In: UK WORKSHOP ON COMPUTATIONAL INTELLIGENCE, 2008. [S.l.], **Proceedings …** [S.l.: s.n.], 2008.

ARAUJO, B. B. P. L. ; FEIJÓ, B. **Um estudo sobre adaptatividade dinâmica de dificuldade em jogos**. 2012. Dissertação (Mestrado em Informática) — Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2012.

BRAGA, A. P. ; CARVALHO, A. P. L. F. ; LUDERMIR, T. B. **Redes neurais artificiais** - teoria e aplicações. 2.ed. Rio de Janeiro: LTC, 2007.

CRUZ, A. J. O. ; DEMASI, P. Modelagem fuzzy para um jogo de naves espaciais. In: BRAZILIAN WORKSHOP IN GAMES AND DIGITAL ENTERTAINMENT, 1., 2002, Fortaleza. **Anais …**, Fortaleza: SBC, 2002. v.1. p. 100-106.

CRUZ, A. J. O. ; DEMASI, P. Aprendizado de regras nebulosas em tempo real para jogos eletrônicos. In: WJOGOS 2003 – WORKSHOP DE JOGOS E ENTRETENIMENTO DIGITAL, 2., 2003, Salvador. **Anais …** Porto Alegre: SBC, 2003. v. 1. p. 1-9.

CRUZ, A. J. O. ; DEMASI, P. Anticipating opponent behaviour using sequential prediction and Real-time fuzzy learning. In: INTERNATIONAL CONFERENCE ON INTELLIGENT GAMES AND SIMULATION, 4., 2003, London. **Proceedings …** Erlangen: SCS Publishing House, 2004. p.101–105.

DEMASI, P. **Estratégias adaptativas e evolutivas em tempo real para jogos eletrônicos.** 2003. Dissertação (Mestrado em Informática) — Programa de Pós-Graduação em Informática, Instituto de Matemática, Instituto Tercio Pacitti de Aplicações e Pesquisas Computacionais, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2003.

GRIECO, B. P. A. **Criação de oponentes virtuais com características realistas em jogos eletrônicos através de técnicas de inteligência computacional**. 2007. Dissertação (Mestrado em Informática) — Instituto de Matemática, Instituto Tercio Pacitti de Aplicações e Pesquisas Computacionais, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2007.

JANG, H. ; PARK, A. ; JUNG, K. Neural network implementation using CUDA and OpenMP. In: DIGITAL IMAGE COMPUTING: TECHNIQUES AND APPLICATIONS, DICTA'08. 2008, Canberra, Australia. **Proceedings …** Los Alamitos: IEEE, 2008. p.155 –161.

JANG, J.-S. ANFIS: adaptive-network-based fuzzy inference system. **IEEE Transactions on Systems, Man and Cybernetics**, Piscataway, v. 23, n. 3, p.665 –685, May/Jun 1993.

JUANG, C.-F.; CHEN, T.-C.; CHENG, W.-Y. Speedup of implementing fuzzy neural networks with high-dimensional inputs through parallel processing on graphic processing units. **IEEE Transactions on Fuzzy Systems**, Piscataway, v.19, n. 4, p. 717 –728, Aug. 2011.

JUANG, C.-F. ; LIN, C.-T. An online self-constructing neural fuzzy inference network and its applications. **IEEE Transactions on Fuzzy Systems**, Piscataway, v. 6, n. 1, p. 12 –32, Feb 1998.

KOSKO, B. Comparison of fuzzy and neural truck backer-upper control system. In: IJCNN INTERNATIONAL JOINT CONFERENCE, 1992. Baltimore. **Proceedings ...** [S.l.]: IEEE, 1992. p.339–361.

LOPES, R. O. **Um simulador de emoções utilizando lógica nebulosa**. 2011. Trabalho Final de Curso (Bacharel em Informática) – Instituto de Matemática, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2011.

LUKE, R. H. et al. Fuzzy logic-based image processing using graphics processor units. In: IFSAWC-EUSFLATC, 2009, Lisbon. **Proceedings …** Lisbon: IFSA/EUSFLAT, 2009. p.288–293, 2009.

MAMDANI, E. H. Application of fuzzy logic to approximate reasoning using linguistic synthesis. **IEEE Transactions on Computers**, Piscataway, v. C-16, n. 12, p. 1182–1191, Dec. 1977.

MOTA, T. C. **Análise e proposta de controladores para navegação autônoma de um robô inteligente**. 2010. Dissertação (Mestrado em Informática) — Programa de Pós-Graduação em Informática, Instituto de Matemática, Instituto Tercio Pacitti de Aplicações e Pesquisas Computacionais, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2010.

NAGESWARAN, J. et al. Efficient simulation of large-scale Spiking neural networks using CUDA graphics processors. In: IJCNN 2009 - INTERNATIONAL JOINT CONFERENCE ON NEURAL NETWORKS, 2009, Piscataway. **Proceedings …** Piscataway: IEEE, 2009. p.2145 –2152.

NGUYEN, D. ; WIDROW, B. The truck backer-upper: an example of self-learning in neural network. **IEEE Control System Magazine**, New York, v. 10, n. 3, p.18–23, Apr. 1990.

OWENS, J. D. et al. A Survey of general-purpose computation on graphics hardware. **Computer Graphics Forum**, Amsterdam, v. 26, n. 1, p. 80–113, 2007.

PASSINO, K. M. ; YURKOVICH, S. **Fuzzy control**. Menlo Park: Addison- Wesley, 1998.

RAO, C. et al. **Linear models**: least squares and alternatives. New York: Springer, 1999. (Springer Series in Statistics).

ROSS, T. J. **Fuzzy logic with engineering applications**. 3. ed. Chichester: John Wiley & Sons, 2010.

RUSSELL, S. J. ; NORVIG, P. **Artificial intelligence: a modern approach**. 2.ed. Harlow: Pearson Education, 2003.

SCHWAB, B. **AI game engine programming**. Rockland, MA: Charles River Media, 2004. (Game Development Series).

SUGENO, M. ; KANG, G. T. Structure identification of fuzzy model. **Fuzzy Sets and Systems**, [S.l.], v. 28, n. 1, p.15–33, Oct. 1988.

TAKAGY, T. ; SUGENO, M. Fuzzy identification of systems and its applications to modeling and control. **IEEE Transactions on System, Man and Cybernetics**, Piscataway, v. 15, n. 1, p.116–132, Jan./Feb. 1985.

UETZ, R. ; BEHNKE, S. Large-scale object recognition with CUDA-accelerated hierarchical neural networks. In: ICIS 2009 - IEEE INTERNATIONAL CONFERENCE ON INTELLIGENT COMPUTING AND INTELLIGENT SYSTEMS, 2009. Shanghai. **Proceedings …** [S.l.]: IEEE, 2009. v.1, p.536 –541.

WANDERLEY, M. F. B. et al. A maximum margin-based kernel width estimator and its application to the response to neoadjuvant chemotherapy. **Brazilian Journal of Biomedical Engineering,** Rio de Janeiro, v. 30, n. 1, mar. 2014.

ZADEH, L. A. Fuzzy sets. **Information and Control**, New York, v.8, p.338–353, 1965.