PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Daniel Salles Chevitarese

ANNCOM – Biblioteca de Redes Neurais Artificiais para Alto Desempenho Utilizando Placas de Vídeo

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo programa de Pós Graduação em Engenharia Elétrica do Departamento de Engenharia Elétrica da PUC-Rio.

Orientadores: Marley M. B. Rebuzzi Vellasco
Dilza Mattos Szwarcman

Rio de Janeiro
Março de 2010

Livros Grátis

http://www.livrosgratis.com.br

Milhares de livros grátis para download.

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Daniel Salles Chevitarese

ANNCOM – Biblioteca de Redes Neurais Artificiais com Treinamento Acelerado por Placas Gráficas

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-Graduação em Engenharia Elétrica do Departamento de Engenharia Elétrica do Centro Técnico Científico da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Profa. Marley Maria Bernardes Rebuzzi Vellasco
Orientadora
Departamento de Engenharia Elétrica – PUC-Rio

Profa. Dilza Mattos Szwarcman
Co-Orientadora
Departamento de Engenharia Elétrica – PUC-Rio

Profa. Cristiana Bentes
UERJ

Prof. Ricardo Cordeiro de Farias
Universidade Federal do Rio de Janeiro

Prof. Carlos Roberto Hall Barbosa
Programa de Pós-Graduação em Metrologia – PUC-Rio

Prof. José Eugenio Leal
Coordenador Setorial do Centro
Técnico Científico

Todos os direitos reservados. É proibida a reprodução total ou parcial do trabalho sem autorização da universidade, do autor e do orientador.

Daniel Salles Chevitarese

Graduou-se em Engenharia de Computação pela PUC-Rio em 2007

Ficha Catalográfica

Chevitarese, Daniel Salles

ANNCOM – Biblioteca de redes neurais artificiais para desempenho utilizando placas de vídeo / Daniel Salles Chevitarese; orientadores: Marley M. B. Rebuzzi Vellasco, Dilza Mattos Szwarcman. – 2010. 97 f. : il. (color.) ; 30 cm

Dissertação (mestrado) – Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Engenharia Elétrica, 2010.

Inclui bibliografia

1. Engenharia elétrica – Teses. 2. Redes neurais artificiais. 3. Engenharia de software. 4. Computação de alto desempenho. 5. GPGPU. 6. CUDA. I. Vellasco, Marley M. B. Rebuzzi. II. Szwarcman, Dilza Mattos. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Engenharia Elétrica. IV. Título.

CDD: 621.3

Para minha Vivi.

Agradecimentos

À minha Orientadora Profa. Marley M. B. Rebuzzi Vellasco pelo estímulo e parceria para a realização deste trabalho.

À Dra. Dilza Mattos Szwarcman que foi muito mais que minha co-orientadora, mas uma grande amiga.

À CAPES e à PUC-Rio, pelos auxílios concedidos, sem os quais este trabalho não poderia ter sido realizado.

Ao ICA pelo conhecimento e treinamento.

Aos professores que participaram da Comissão Examinadora.

A todos os amigos e familiares que de uma forma ou de outra me estimularam e ajudaram.

Aos meus pais que sempre procuraram me guiar pelo caminho do bem.

Resumo

Chevitarese, Daniel Salles; Vellasco, Marley M. B. Rebuzzi **ANNCOM – Biblioteca de Redes Neurais Artificiais para Alto Desempenho Utilizando Placas de Vídeo**. Rio de Janeiro, 2010. 97p. Dissertação de Mestrado - Departamento de Engenharia Elétrica, Pontifícia Universidade Católica do Rio de Janeiro.

As Redes Neurais Artificiais têm sido utilizadas com bastante sucesso em problemas de previsão, inferência e classificação de padrões. Por essa razão, já se encontram disponíveis diversas bibliotecas que facilitam a modelagem e o treinamento de redes, tais como o NNtool do Matlab ou o WEKA. Embora essas bibliotecas sejam muito utilizadas, elas possuem limitações quanto à mobilidade, à flexibilidade e ao desempenho. Essa última limitação é devida, principalmente, ao treinamento que pode exigir muito tempo quando existe uma grande quantidade de dados com muitos atributos. O presente trabalho propõe o desenvolvimento de uma biblioteca (ANNCOM) de fácil utilização, flexível, multiplataforma e que utiliza a arquitetura CUDA (*Compute Unified Device Architecture*) para reduzir os tempos de treinamento das redes. Essa arquitetura é uma forma de GPGPU (*General-Purpose computing on Graphics Processing Units*) e tem sido utilizada como uma solução em computação paralela na área de alto desempenho, uma vez que a tecnologia utilizada nos processadores atuais está chegando ao limite de velocidade. Adicionalmente, foi criada uma ferramenta gráfica que auxilia o desenvolvimento de soluções aplicando as técnicas de redes neurais de forma fácil e clara usando a biblioteca desenvolvida. Para avaliação de desempenho da ANNCOM, foram realizados seis treinamentos para classificação de clientes de baixa tensão de uma distribuidora de energia elétrica. O treinamento das redes, utilizando a ANNCOM com a tecnologia CUDA, alcançou um desempenho quase 30 vezes maior do que a ANNCOM auxiliada pela MKL (*Math Kernel Library*) da Intel, também utilizada pelo Matlab.

Palavras-chave

Redes Neurais Artificiais; Engenharia de Software; Computação de Alto Desempenho; GPGPU; CUDA.

Abstract

Chevitarese, Daniel Salles; Vellasco, Marley M. B. Rebuzzi (Advisor) **ANNCOM – Artificial Neural Network Library for High Performance Computing using Graphic Cards**. Rio de Janeiro, 2710. 95p. MSc Dissertation - Departamento de Engenharia Elétrica, Pontifícia Universidade Católica do Rio de Janeiro.

The Artificial Neural Networks have been used quite successfully in problems of prediction, inference and classification standards. For this reason, are already available several libraries that facilitate the modeling and training networks, such as NNtool Matlab or WEKA. While these libraries are widely used, they have limited mobility, flexibility and performance. This limitation is due mainly to the training that can take a long time when there is a large amount of data with many attributes. This paper proposes the development of a library (ANNCOM) easy to use, flexible platform and architecture that uses the CUDA (Compute Unified Device Architecture) to reduce the training times of the networks. This architecture is a form of GPGPU (GeneralPurpose computing on Graphics Processing Units) and has been used as a solution in parallel computing in the area of high performance, since the technology used in current processors are reaching the limit of speed. Additionally created a graphical tool that helps the development of solutions using the techniques of neural networks easily and clearly using the library developed. For performance evaluation ANNCOM were conducted six trainings for customer classification of a low voltage electricity distribution. The training of networks using ANNCOM with CUDA technology, achieved a performance nearly 30 times greater than the ANNCOM aided by MKL (Math Kernel Library) by Intel, also used by Matlab.

Keywords

Artificial Neural Networks; Software Engineering; High Performance Computing; GPGPU; CUDA.

Sumário

1 Introdução 14
1.1. Motivação 14
1.2. Objetivos 15
1.3. Descrição do trabalho 16
1.4. Estrutura da Dissertação 17

2 Conceitos da Engenharia de Software 18
2.1. Requisitos de Software 18
2.2. Arquitetura de Software 19
2.2.1. Decomposição em Módulos 21
2.3. Padrões de Projeto 23
2.3.1. Composição (*Composite*) 25
2.3.2. Cadeia de Responsabilidade (*Chain of Responsibility*) 26
2.3.3. Estratégia (*Strategy/Policy*) 27
2.4. .NET Framework 29
2.4.1. Linguagem Comum em Tempo de Execução (CLR) 30

3 Conceitos da Computação de Alto Desempenho 33
3.1. Conceitos da Computação Paralela 34
3.1.1. Leis de *Amdahl* e *Gustafson* 35
3.1.2. Tipos de Paralelismo 37
3.2. Os Processadores Gráficos e a GPGPU 39
3.3. CUDA – NVIDIA 42
3.3.1. O Modelo de Programação 44
3.3.2. A Implementação do Hardware 48

4 Desenvolvimento de uma Biblioteca de Redes Neurais de Alto Desempenho – ANNCOM 51
4.1. Levantamento de Requisitos e Arquitetura 52
4.2. Estrutura Básica 56
4.2.1. Componente NeuralNet 57

4.2.2. Modelos de Redes Neurais Artificiais 59
4.2.3. Estrutura NetOutput 60
4.2.4. Estruturas para Cálculo de Erro 61
4.3. Modelo de Treinamento 63
4.3.1. O Treinamento em GPGPU 70
4.3.2. Implementação na GPU 71
4.3.3. Segunda Parte: Inversão Matricial 71
4.4. Ferramenta Gráfica para Criação de Soluções Utilizando Redes Neurais – Clinn 74
4.4.1. Interface utilizando Docas Flutuantes 74
4.4.2. Processo de Criação Automatizada 81
4.4.3. Processo de Treinamento 82

5 Estudo de Casos 84
5.1. Caso 1 – Treinamento dos Comitês para Classificação do Cliente Fraudador da Light 84
5.1.1. Estrutura do Sistema 86
5.1.2. Resultados do Treinamento e Testes de Desempenho 88

6 Conclusões e Trabalhos Futuros 93

Referências Bibliográficas 95

Lista de Figuras

Figura 1 – Início do processo de desenvolvimento de um software. 20
Figura 2 – Parte do diagrama de classes de objetos da ANNCOM em UML. 21
Figura 3 – Estrutura recursiva da ANNCOM. 25
Figura 4 – Cadeia de responsabilidade do método *propagate*. 26
Figura 5 – Algoritmos de treinamento da ANNCOM encapsulados. 28
Figura 6 – O *.NET Framework* e o resto do sistema (MICROSOFT, 2009). 30
Figura 7 – O tempo de execução e o aumento de velocidade de um programa com paralelismo. 37
Figura 8 – Representação gráfica da lei de Amdahl. 37
Figura 9 – Comparativo entre CPU e GPU com relação às operações de ponto flutuante por segundo (NVIDIA, 2009). 40
Figura 10 – Comparativo entre CPU e GPU com relação à largura de banda de memória memória (NVIDIA, 2009). 41
Figura 11 – Diferença entre GPU e CPU com relação à utilização de transistores (NVIDIA, 2009). 42
Figura 12 – Suporte de CUDA para várias linguagens (NVIDIA, 2009). 43
Figura 13 – Grade de blocos de *threads* (NVIDIA, 2009). 44
Figura 14 – Os diferentes níveis de memória nas placas de vídeo da NVIDIA (NVIDIA, 2009). 46
Figura 15 – Modelo de programação heterogêneo (NVIDIA, 2009). 47
Figura 16 – Escalonamento automático feito em CUDA (NVIDIA, 2009). 49
Figura 17 – Modelo de hardware de uma placa gráfica com suporte a CUDA (NVIDIA, 2009). 50
Figura 18 – Visão geral dos componentes principais da ANNCOM. 53
Figura 19 – Suporte a conexão com vários tipos de SGBD. 54
Figura 20 – Suporte da ANNCOM para tempo de execução no Visual Studio. 55
Figura 21 – Adição e edição de coleções na ANNCOM. 56
Figura 22 – Visão geral do desvio de parte do treinamento para GPU. 56

Figura 23 – Esquema com todos os espaços de nomes da biblioteca ANNCOM. 57
Figura 24 – Representação de uma rede neural MLP. 58
Figura 25 – Diagrama de classes simplificado dos modelos de redes neurais implementados nesse trabalho. 59
Figura 26 – Modelo de uma rede Elman. 60
Figura 27 – Diagrama de classes com as estruturas de decodificação das saídas das redes neurais. 61
Figura 28 – Diagrama de classes com os objetos para cálculo de erro, implementados na ANNCOM. 61
Figura 29 – Ilustração de uma descida por gradiente (WIKIPEDIA, 2010). 64
Figura 30 – Exemplo de duas “salas de aula” onde várias redes são treinadas. 68
Figura 31 – Validação cruzada automática da ANNCOM. 68
Figura 32 – Exemplo de funcionamento do Gerenciador de Erro (GE) para cálculo de erros de validação e teste durante o treinamento. 69
Figura 33 – Segundo modelo (resumido) proposto, que executa uma porção maior do código na placa gráfica. 70
Figura 34 - Divisão do processamento em kernels. 71
Figura 35 – Divisão da matriz em blocos de tamanho *n* (nesse caso, *n* é 4). 72
Figura 36 – Atualização das linhas adjacentes. 72
Figura 37 – Blocos atualizados. 73
Figura 38 – Linha de fatores para a multiplicação dos pivôs. 73
Figura 39 – Tela inicial do Clinn. 74
Figura 40 – Movimentação das docas pelo programa. 75
Figura 41 – A doca de propriedades em detalhes. À direita, a figura mostra a facilidade de navegação pelo componente. 76
Figura 42 – Editor de coleção de camadas. 77
Figura 43 – Editor de coleção de neurônios. 77
Figura 44 – Editor de coleção de sinapses. 78
Figura 45 – Doca exploradora de solução. 79
Figura 46 – Explorador de bases de dados. 79

Figura 47 – Possibilidade de visualizar e editar as informações das tabelas. 80
Figura 48 – Doca de documentos com uma lista de arquivos abertos. 81
Figura 49 – Modelo passo a passo para se criar uma nova rede neural. 82
Figura 50 – Função sigmoid de um neurônio da última camada da rede neural. 83
Figura 51 – Visão geral do processo de classificação dos clientes de baixa tensão. 87
Figura 52 – Tempo (em segundos) vs. Tamanho (em número de elementos) da matriz de entada. 91

Lista de Tabelas

Tabela 1 – Principais características das linguagens *.NET*. 32
Tabela 2 – Taxonomia Flynn. 38
Tabela 3 – Modelos de placas usadas no estudo de casos. 84
Tabela 4 – Descrição dos atributos de entrada da base de clientes da Light. 87
Tabela 5 – Tempos referentes à base de clientes comerciais. 88
Tabela 6 – Tempos referentes à base de clientes da região Leste. 89
Tabela 7 – Tempos referentes à base de clientes da região Oeste. 89
Tabela 8 – Tempos referentes à base de clientes da região Litorânea. 90
Tabela 9 – Tempos referentes à base de clientes da região Interior. 90
Tabela 10 – Tempos referentes à base de clientes da região Baixada. 91

1
Introdução

1.1.
Motivação

O cérebro possui incrível capacidade de processamento. Sua estrutura altamente complexa, não linear e paralela, permite executar alguns tipos de tarefas mais rapidamente do que qualquer computador já construído, embora seja superado pelas máquinas atuais nas operações matemáticas ou repetitivas. Para exemplificar a incrível capacidade do cérebro humano, o sistema visual precisa criar uma representação do ambiente e extrair toda a informação necessária para interação com o mesmo em uma fração de segundo. Esse exemplo está entre as tarefas rotineiras de reconhecimento perceptivo do cérebro humano, que são feitas em pouco mais de 100ms, enquanto que um computador pode levar horas, ou até dias (HAYKIN, 2007).

As redes neurais artificiais têm suas bases em várias disciplinas: neurociência, estatística, ciência da computação, engenharia, etc., e tentam imitar a capacidade de reconhecimento do cérebro biológico (HAYKIN, 2007). Elas são utilizadas em muitas áreas, como modelagem e análise de séries temporais, reconhecimento, agrupamento e classificação de padrões, processamento de sinais e controle, mas, assim como o cérebro biológico, precisam ser treinadas e isso pode consumir muito tempo.

Com o aumento da quantidade de informação e a menor disponibilidade de tempo, os sistemas baseados em redes neurais precisam responder cada vez mais rapidamente com cada vez mais informação. Mais do que isso, elas devem ser extremamente flexíveis de forma a se adaptarem aos mais diferentes tipos de problemas, minimizando os custos com modelagem, treinamento, desenvolvimento e implantação.

Algumas bibliotecas de redes neurais já foram desenvolvidas, como a do Matlab ou do Scilab, que são configuradas por meio de *scripts* e, com isso, oferecem muita flexibilidade. Entretanto, esse tipo de configuração dificulta a modela-

gem dos problemas, uma vez que o arquiteto da solução precisa ter conhecimento específico do *script* daquele programa. Além disso, o uso de *script* prejudica muito o desempenho, uma vez que este precisa ser interpretado, compilado e, só então, processado.

Por outro lado, existem programas que possuem um ambiente gráfico, como é o caso do Weka. Entretanto, esses programas, normalmente, oferecem pouca ou nenhuma flexibilidade com relação à quantidade de algoritmos de treinamento e modelos de redes neurais. O Matlab possui tanto um ambiente gráfico (através do NNtool) quanto um ambiente de programação por *script*, conforme já foi dito anteriormente, mas seu desempenho deixa muito a desejar. Os trabalhos envolvendo CUDA e redes neurais estão em fase inicial de desenvolvimento e apresentam apenas treinamentos com gradiente decrescente simples, como por exemplo (LAHABAR, AGRAWAL e NARAYANAN, 2009), e outros casos apresentam apenas a propagação de redes, como por exemplo, (JANG, PARK e JUNG, 2008), mas sem possibilidade de configuração de modelos ou de treinamentos.

1.2. Objetivos

A presente dissertação propõe uma biblioteca de componentes de redes neurais com dois principais objetivos: flexibilidade e desempenho. O primeiro objetivo requer que a biblioteca seja completamente configurável e suporte situações não previstas como uma topologia nova, na qual os neurônios possuem outro tipo de comportamento ou suas camadas sejam organizadas de forma não convencional. O segundo objetivo é o mais desafiador, pois exige que a biblioteca forneça suporte ao treinamento de sistemas com várias redes ou com vários comitês de redes e utilize bases de dados muito grandes em tempos muito curtos. Essa restrição de desempenho implica no desenvolvimento de um algoritmo de treinamento que rode em placas gráficas, de modo a paralelizar ao máximo a execução do problema.

Além disso, o trabalho apresenta um sistema de janelas gráficas que facilita a modelagem, o treinamento e a implantação das redes neurais, tornando estes procedimentos mais rápidos e disponíveis para diversos tipos de usuário.

1.3. Descrição do trabalho

Este trabalho se desenvolveu de acordo com os seguintes passos: (a) estudo dos modelos de redes neurais existentes e seus processos de treinamento com os diferentes tipos de algoritmos; (b) levantamento das vantagens e desvantagens dos diversos sistemas existentes no mercado, e estudo das áreas de engenharia de software e de computação de alto desempenho; (c) definição e desenvolvimento de uma biblioteca orientada a objetos com capacidade de modelar os diversos tipos de redes neurais e de treinamentos, priorizando o desempenho; (d) desenvolvimento de uma ferramenta gráfica baseada na biblioteca de redes definida e avaliação dos resultados por meio de um estudo de caso com sete experimentos.

O estudo sobre as redes neurais artificiais constituiu-se do levantamento de material bibliográfico sobre os diversos modelos de redes neurais existentes, suas aplicações e os principais algoritmos de treinamento.

O levantamento dos sistemas existentes exigiu uma busca de dois principais grupos de software: os gratuitos e os pagos. Percebeu-se que existe uma grande carência de programas gratuitos que possam ser utilizados em projetos na área de apoio à decisão. Além disso, a grande maioria dos sistemas pagos, baseados em redes neurais, apresenta uma arquitetura pobre, com relação à interface com o usuário e configurações disponíveis, e não respondem rapidamente em vários problemas reais.

O desenvolvimento de uma biblioteca, que respeitasse as restrições de arquitetura e desempenho, exigiu o estudo de duas áreas da computação: engenharia de software e computação de alto desempenho. Os conceitos da primeira foram usados na modelagem, arquitetura e ciclo de vida do sistema e os conceitos da segunda serviram para modificar partes chaves do código visando à redução do tempo de processamento dos algoritmos de treinamento.

Outro ponto importante percebido nos sistemas de redes neurais existentes foi a interface. Essa, normalmente, exige noções de programação dos clientes ou possui uma interface estática que facilita a utilização do sistema, mas torna a personalização dos modelos limitada ou impossível.

Finalmente, foram realizados sete experimentos de um estudo de caso para se avaliar o desempenho do sistema desenvolvido neste trabalho. Foram obtidos

seus gráficos comparativos e o desempenho das soluções foi comparado com os resultados das diferentes abordagens presentes na bibliografia.

1.4. Estrutura da Dissertação

Esta dissertação está dividida em cinco capítulos adicionais, descritos a seguir.

O Capítulo 2 apresenta os principais conceitos da Engenharia de Software, incluindo o levantamento de requisitos, projeto e arquitetura, os padrões de projeto utilizados no trabalho e o arcabouço da Microsoft, que foi utilizado como base para o desenvolvimento deste trabalho.

O Capítulo 3 descreve as várias abordagens utilizadas na história da computação para se melhorar o desempenho dos sistemas e como cada uma funciona.

O Capítulo 4 apresenta a biblioteca de componentes proposta nesta dissertação, detalhando a sua estrutura, seus componentes de redes neurais e algoritmos de treinamento, além de uma ferramenta gráfica para criação de soluções com redes neurais artificiais. Por último, são apresentadas as modificações feitas na arquitetura da biblioteca de modo que esta fosse executada, em parte, por uma placa gráfica.

O Capítulo 5 descreve os experimentos do estudo de caso com seus respectivos modelos de redes neurais e discute os resultados obtidos em função das referências bibliográficas.

Por fim, o Capítulo 6 apresenta as conclusões do trabalho e sugere novas direções possíveis para a continuação da pesquisa apresentada.

2
Conceitos da Engenharia de Software

> Engenharia de Software é a criação e a utilização de sólidos princípios de engenharia a fim de obter software de maneira econômica, que seja confiável e que trabalhe eficientemente em máquinas reais.
>
> Friedrich Ludwig Bauer

Os sistemas com base em computadores têm se tornado cada vez mais complexos. Atualmente, grande parte dos serviços e dos produtos faz uso ou incorpora computadores ou sistemas, seja para controle ou para produção e, nestes, o software representa um grande e crescente percentual do custo total. Assim, a economia mundial vem se tornando fortemente atenta à razão custo-benefício dos sistemas de software.

A Engenharia de Software é uma disciplina que busca maximizar a relação custo-benefício no desenvolvimento de sistemas de software. O fato de o software ser abstrato e intangível torna seu desenvolvimento simples, já que ele não precisa respeitar leis da física, por exemplo. Por outro lado, essa característica pode deixá-lo muito complexo e difícil de ser compreendido.

Os primeiros conceitos de Engenharia de Software datam de 1968, na chamada "crise do software", quando a terceira geração de computadores ficou disponível e seu poder de processamento era, naquela época, inimaginável. Os sistemas subseqüentes se transformaram em grandes sistemas de informação e a abordagem até então utilizada para construí-los se mostrou ineficaz (SOMMERVILLE, 2004).

2.1.
Requisitos de Software

Os desafios enfrentados ao se desenvolver um software podem ser extremamente complexos. Para compreender o universo do problema e para estabelecer o que um sistema precisa fazer é necessário descrever todas as suas funções e res-

trições. O processo de descobrir, analisar, documentar e verificar essas funções e restrições é denominado Engenharia de Requisitos.

Um requisito é, ao mesmo tempo, uma declaração abstrata e uma definição detalhada de uma funcionalidade ou restrição que o sistema deve possuir, (SOMMERVILLE, 2004). Um exemplo que facilita entender esse paradoxo é o de uma empresa que precisa estabelecer um contrato para o desenvolvimento de um grande projeto de software. Para isso, ela deve definir suas necessidades de maneira suficientemente abstrata, para que uma solução não seja predefinida. Entretanto, os requisitos devem ser rígidos de forma que os diversos fornecedores apresentem suas propostas, podendo até oferecer diversas maneiras de atender às necessidades do cliente. A partir do contrato, o fornecedor precisa preparar uma definição de sistema com mais detalhes para que o cliente compreenda e valide o software a ser feito. Esse dois documentos são chamados de Documentos de Requisitos do Sistema (DAVIS, 1993).

Os requisitos de sistema são classificados, normalmente, como funcionais, não funcionais e de domínio. Os requisitos funcionais definem **o que** o sistema deve ou não deve fazer e **como** o sistema deve reagir às diferentes situações de uso. Os requisitos não funcionais são restrições que o sistema deve respeitar. Por exemplo, restrições de desempenho, de padrões, de segurança, etc. Os requisitos de domínio se referem ao domínio de aplicação do sistema e podem ser funcionais ou não funcionais (SOMMERVILLE, 2004).

2.2. Arquitetura de Software

Normalmente, um sistema de médio ou grande porte necessita ser subdividido para que seu desenvolvimento seja simplificado. O processo de identificar os subsistemas, produtos dessa divisão, e estabelecer um arcabouço para o controle e a comunicação entre eles é chamado de Projeto de Arquitetura. A saída desse processo é uma descrição da arquitetura do software.

O Projeto de Arquitetura é o primeiro estágio do desenvolvimento de um software e possui uma forte ligação com a Engenharia de Requisitos descrita na seção anterior. A decomposição da arquitetura em subsistemas ajuda a estruturar e organizar a especificação dos requisitos do software em questão. A Figura 1 apre-

senta as primeiras etapas do processo de desenvolvimento de um software, onde as especificações abstratas são as especificações de cada um dos subsistemas. Nessa figura, nota-se que o Projeto de Arquitetura gera a Arquitetura do Sistema e a Especificação Abstrata gera a Especificação mais formal do software.

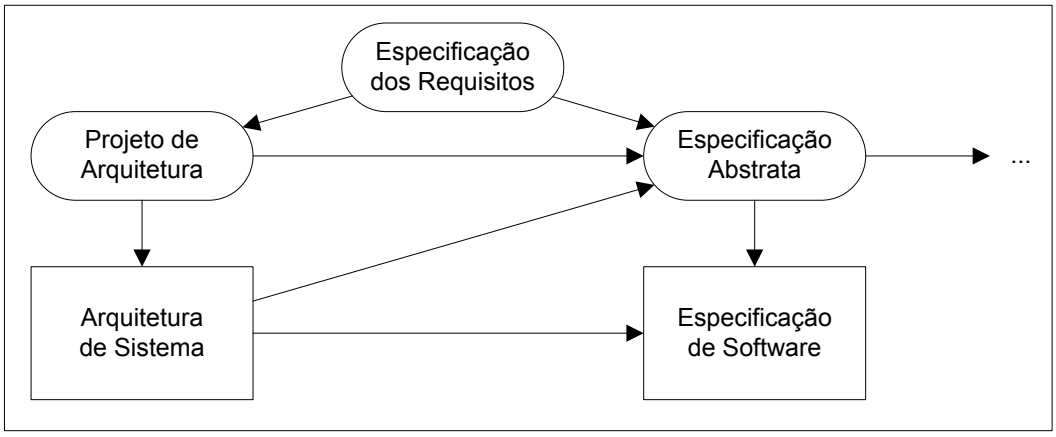


Figura 1 – Início do processo de desenvolvimento de um software.

O projeto da arquitetura precisa estabelecer um arcabouço básico do sistema e envolve a identificação dos seus componentes principais e do processo de comunicação entre esses componentes (BASS, CLEMENTS e KAZMAN, 2003). A documentação dessa arquitetura facilita a comunicação com os *stakeholders*, porque apresenta o sistema em alto nível, e facilita a análise do sistema quanto, por exemplo, ao desempenho, à confiabilidade, etc. Tal documentação ainda possibilita a reutilização em larga escala do projeto no desenvolvimento de outros sistemas com requisitos similares (GAMMA, HELM, *et al.*, 1994).

Os documentos de arquitetura incluem as representações gráficas dos modelos do sistema e um texto descrevendo esses modelos. Além disso, eles devem apresentar como o sistema é dividido em subsistemas e como esses são quebrados em módulos. Alguns modelos gráficos freqüentemente utilizados são:

- Modelo estrutural estático: apresenta os subsistemas ou componentes a serem desenvolvidos;
- Modelo do processo dinâmico: mostra a organização dos processos do sistema em tempo de execução;
- Modelo de interface: define os serviços públicos oferecidos pelos subsistemas;

- Modelos de relacionamento: apresenta o fluxo de dados entre os subsistemas.

Esses modelos podem ser desenvolvidos utilizando-se linguagens de descrição de arquitetura, conhecidas como ADL – *Architectural Description Languages*, que são formadas por componentes e conectores, além de incluírem regras e diretrizes para arquiteturas. Entretanto, tais linguagens não são muito comuns, porque são conhecidas apenas pelos arquitetos de software e isso dificulta a análise dos especialistas de domínio. Por isso, notações informais, como a UML – *Unified Modeling Language*, são mais utilizadas. No desenvolvimento deste trabalho, foi utilizada a UML nos documentos de arquitetura.

2.2.1. Decomposição em Módulos

Depois de montar uma arquitetura da estrutura do sistema, o próximo passo é decompor os subsistemas em módulos. Existem dois modelos principais que podem ser utilizados nessa decomposição: (a) o modelo orientado a objetos e (b) o modelo de fluxo de dados. Ambos podem ser implementados como componentes ou como processos.

No modelo orientado a objetos, os módulos se transformam em conjuntos de objetos minimamente acoplados, mas com interfaces bem definidas, e esses objetos possuem serviços que são invocados por outros objetos. A Figura 2 apresenta uma parte do diagrama de classes de objetos da ANNCOM.

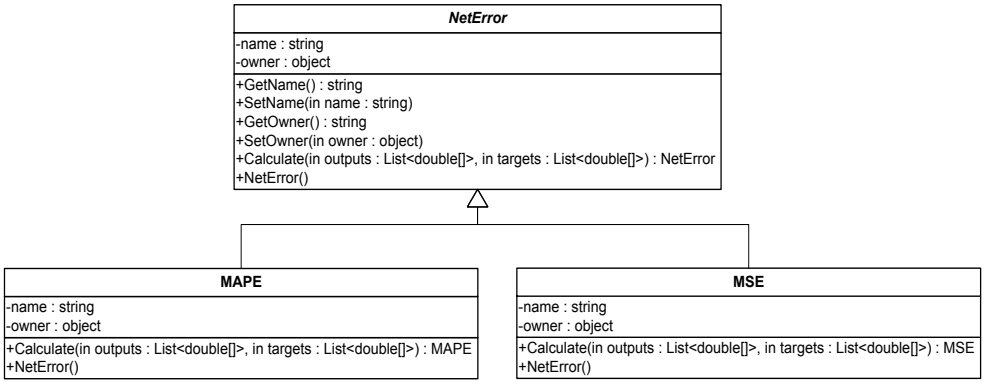


Figura 2 – Parte do diagrama de classes de objetos da ANNCOM em UML.

No diagrama de classes da Figura 2 foi utilizada a linguagem UML, onde as classes têm nomes e um conjunto de atributos e operações. As ligações com seta entre as classes representam uma hierarquia de pai e filho. A saída do processo de decomposição, usando a abordagem orientada a objetos, é um software cujos objetos são criados a partir das classes e com algum modelo de controle para coordenar suas operações.

Algumas vezes, para impedir ambigüidade entre itens que possuem o mesmo nome, os objetos podem ser organizados em Pacotes ou Espaços de Nomes. Esses pacotes podem ser definidos como contêineres abstratos que estabelecem um contexto para os objetos que armazena. Tendo em vista que cada contêiner trata de um contexto distinto, o significado de um nome pode variar de acordo com o Espaço de Nomes ao qual ele pertence.

A abordagem discutida acima apresenta várias vantagens, entre elas a facilidade de manutenção e alteração de partes do programa sem alteração do resto do software, uma vez que os objetos devem ser minimamente acoplados. Além disso, é muito mais fácil compreender um sistema se este é modelado com entidades que correspondem às do mundo real, e que neste caso são representadas por objetos.

Por outro lado, a orientação a objeto introduz alguns problemas. Por exemplo, se for necessário alterar a interface de alguma classe, deverá ser avaliado o impacto disso nas outras classes do sistema. Outro fato que deve ser levado em consideração é a dificuldade de mapear entidades complexas do mundo real em classes.

O outro modelo citado, que é utilizado para decomposição dos subsistemas em módulos, é o de fluxo de dados. Nesse, as transformações funcionais processam suas entradas e produzem saídas. Os dados vão sendo transformados à medida que vão passando de uma função para outra do sistema. Cada etapa do processo é desenvolvida na forma de uma transformação. Essas transformações podem ser efetuadas em série ou em paralelo.

Algumas vantagens do modelo de fluxo de dados são: (a) reuso das transformações; (b) é intuitivo, já que muitas pessoas imaginam seu trabalho na forma de processamento de entradas e saídas; (c) a adição de novas transformações é muito simples de ser projetada e desenvolvida; (d) é, geralmente, mais simples de ser implementado se comparado à orientação a objetos.

A grande desvantagem desse modelo vem da necessidade de se ter um formato comum entre as transformações. Por exemplo, o Unix utiliza um método chamado *pipeline*, onde uma seqüência de caracteres é trocada entre as funções e estas devem analisar suas entradas para gerar a saída. Essa análise cria um tempo extra (*overhead*) no sistema e pode significar que é impossível integrar transformações com entradas e saídas incompatíveis (SOMMERVILLE, 2004).

As duas abordagens apresentadas para decompor um sistema em módulos são exemplos de modelagens já conhecidas para resolver determinados tipos de problemas. Existem na literatura diversas abordagens que auxiliam na solução de problemas, algumas das quais serão descritas a seguir.

2.3. Padrões de Projeto

Atualmente, a maioria dos sistemas é decomposta utilizando-se um modelo orientado a objetos. Como visto anteriormente, essa abordagem possui diversas vantagens mas, em algumas situações, pode impor dificuldades na modelagem de entidades complexas. Mais do que isso, o projetista deve criar um modelo específico para o problema em questão, mas suficientemente abstrato para endereçar problemas correlatos futuros e evitar a remodelagem de sistemas semelhantes.

Para contornar esses problemas, os projetistas utilizam, em projetos novos, soluções e princípios utilizados em projetos anteriores; as boas soluções são reutilizadas sempre que possível. É comum encontrar padrões de classes e de comunicação entre objetos nos diferentes sistemas. Esses padrões resolvem problemas específicos de arquitetura e tornam a orientação a objetos mais flexível, elegante e extremamente reutilizável. Além disso, se um projetista estiver familiarizado com esses padrões, ele poderá aplicá-los rapidamente sem ter que redescobri-los (GAMMA, HELM, *et al.*, 1994).

De acordo com o arquiteto civil Christopher Alexander, um padrão de projeto descreve um problema que ocorre repetidas vezes em um ambiente, e descreve o núcleo da solução para este problema de forma que esta possa ser usada muitas vezes sem precisar ser refeita. Isso também é verdade quando se trata de padrões de projeto de software (GAMMA, HELM, *et al.*, 1994).

Em (GAMMA, HELM, *et al.*, 1994), considera-se que os padrões de projeto são constituídos por quatro partes principais:

- Nome;
- Problema;
- Solução;
- Conseqüências

O nome do padrão resume o problema, suas soluções e conseqüências em uma ou duas palavras. Esse vocabulário de padrões pode ser usado para aumentar o nível de abstração, facilitar a comunicação entre projetistas e a documentação do sistema.

O item Problema indica **quando** o padrão deve ser usado, explicando o problema em questão e seu contexto. Nessa parte, os algoritmos são apresentados como classes de objetos e estruturas que são inflexíveis. Além disso, nesse item podem ser apresentados os pré-requisitos para a aplicação do padrão.

No item Solução são descritos os elementos da arquitetura e seus relacionamentos, suas responsabilidades e colaboradores. Nesse item não é apresentada nenhuma implementação ou arquitetura concreta do sistema, porque os padrões não são específicos para um determinado problema, mas para muitos problemas em diversas situações. Entretanto, a solução provê uma descrição abstrata de um problema arquitetural e como as classes e os objetos o resolvem.

A Conseqüência descreve os resultados e o "preço" pago por se usar o determinado padrão. Uma análise de custo-benefício deve ser feita antes de se utilizar o padrão, de modo que alternativas possam ser estudadas. Normalmente, essas conseqüências se referem a tempo de processamento e espaço em memória, mas podem endereçar detalhes de implementação e linguagem. Como o reuso é uma das principais vantagens da orientação a objetos, as conseqüências incluem impactos na flexibilidade, na portabilidade e na extensibilidade do sistema (GAMMA, HELM, *et al.*, 1994).

A seguir serão apresentados os padrões de projeto utilizados para o desenvolvimento da ANNCOM e uma explicação resumida de cada um.

2.3.1. Composição (*Composite*)

O objetivo desse padrão é compor objetos em estruturas de árvore. Com isso, objetos individuais ou composições desses objetos podem ser tratados de forma igual. No caso da ANNCOM, a complexa relação entre camadas, neurônios e sinapses foi montada de forma que esses objetos mais simples formassem uma rede neural artificial completa. Um detalhe existente no projeto da biblioteca é a inexistência da classe abstrata para representar todos os elementos, pois a classe *NeuralNet*, que será apresentada no capítulo 4, já faz o papel de compor tudo e distinguir o tratamento de cada "sub" classe facilitando a interface com o usuário. A Figura 3 mostra a estrutura recursiva da ANNCOM.

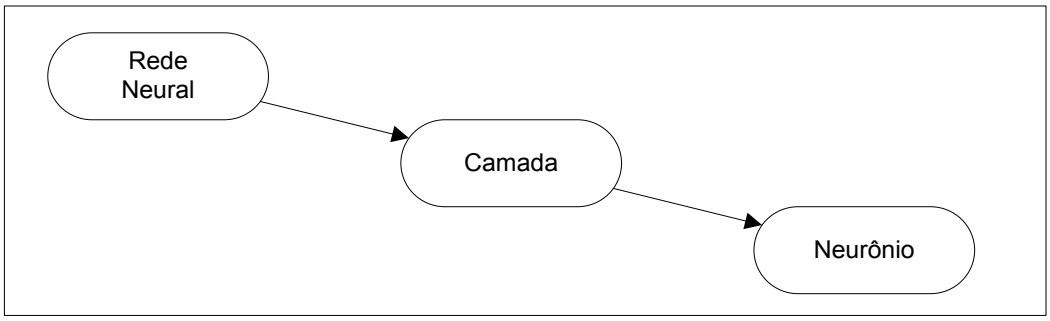


Figura 3 – Estrutura recursiva da ANNCOM.

Os participantes deste padrão de projeto são:

- Componente: os objetos *NeuralNet* e *Synapse* declaram os objetos das "sub" classes, endereçam o acesso a todos os objetos contidos nele, definem o comportamento do grupo e a estrutura recursiva na forma de "pai e filho";
- Folha: os objetos *Layer*, *Neuron* e *InputSynapse* não possuem filhos e definem seus respectivos comportamentos na composição;
- Composição: este representa a rede neural e define o comportamento dos componentes, armazena os objetos que fazem parte da composição e implementa as operações de todo o conjunto;
- Cliente: o aplicativo que será apresentado no capítulo 4, chamado *Clinn*, é o cliente e manipula toda a estrutura.

A primeira conseqüência da Composição é o modelo hierárquico da biblioteca. Além disso, vale destacar: a facilidade de manipulação por parte do cliente, pois este não precisa se preocupar em como a estrutura é formada; a simplicidade

para adição de novas “sub” classes, uma vez que sua estrutura interna deve ser muito parecida com a de outra “sub” classe irmã; e a generalidade do modelo. Esta última característica é extremamente importante no caso da ANNCOM, porque permite a criação de diversos tipos de redes neurais utilizando os mesmos componentes.

2.3.2. Cadeia de Responsabilidade (*Chain of Responsibility*)

A Cadeia de Responsabilidade tem como objetivo evitar o acoplamento entre o objeto solicitante de um pedido e o objeto que foi solicitado, criando uma cadeia onde vários objetos têm a possibilidade de manipular a requisição. Na ANNCOM, um exemplo de uma cadeia é o pedido de propagação de um sinal feito pelo cliente à rede neural. Essa solicitação pode ser tratada pela rede, camadas ou neurônios. Um problema dessa abordagem é que o objeto solucionador não tem informação sobre o objeto que foi requisitado. Entretanto, no nosso caso, essa informação não é importante. A Figura 4 mostra como o objeto *NeuralNet* passa o pedido de propagação para os objetos subseqüentes.

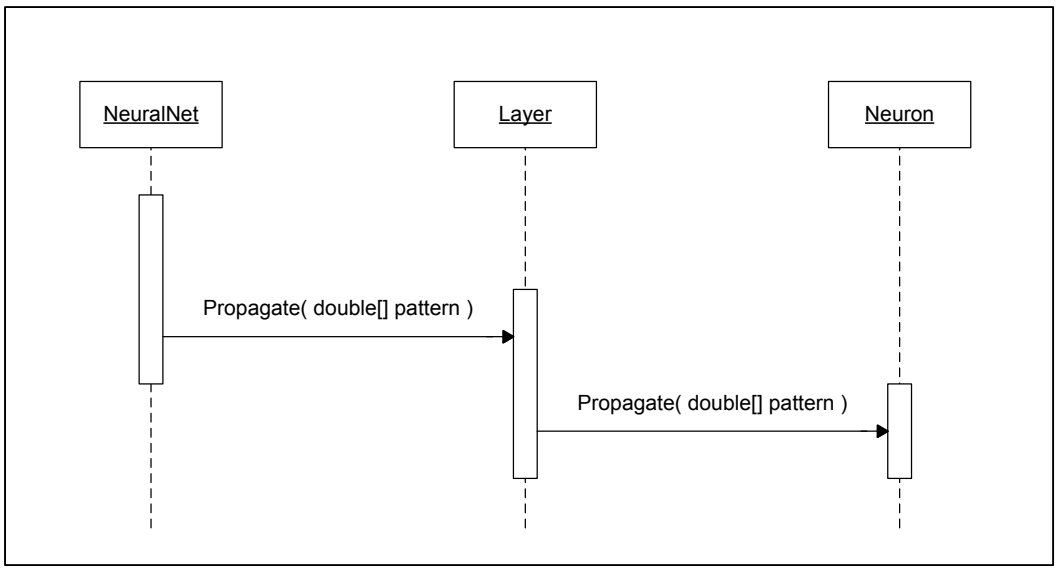


Figura 4 – Cadeia de responsabilidade do método *propagate*.

Para que o pedido seja passado por toda a cadeia, é necessário que cada objeto dentro dela compartilhe uma interface comum para atender ao pedido e passá-

lo ao seu sucessor. Deve-se perceber que os métodos na Figura 4 possuem a mesma interface.

No caso da cadeia de responsabilidade os participantes estão listados com suas respectivas descrições abaixo:

- Manipulador: o objeto *NeuralNet* define a interface para o cliente iniciar o processo de propagação;
- Manipulador concreto: os objetos *Layer* e *Neuron* tratam os pedidos pelos quais são responsáveis ou encaminham para o sucessor;
- Cliente: como no exemplo do padrão anterior, o *Clinn* é um cliente, pois inicia o processo.

Ao se utilizar a Cadeia de Responsabilidade é possível obter uma redução de acoplamento entre os objetos, uma vez que o objeto está livre da responsabilidade de saber quem irá tratar determinada solicitação; deve simplesmente saber que alguém irá tratar. Outro ponto positivo é a flexibilidade na delegação de responsabilidades para os objetos. Por exemplo, na ANNCOM, ao se propagar um sinal pela rede, as responsabilidades são distribuídas pelas camadas e, conseqüentemente, pelos neurônios que são os mais especializados. Uma conseqüência negativa da Cadeia de Responsabilidade é a não garantia de tratamento da solicitação. No caso da ANNCOM, a camada recebe o pedido de propagação e a repassa para os neurônios, mas sem saber se este será ou não tratado.

2.3.3. Estratégia (*Strategy/Policy*)

O padrão denominado Estratégia surge da necessidade de se definir uma família de algoritmos. Por meio dessa abordagem, é possível encapsulá-los e torná-los intercambiáveis facilitando a independência de uso para os clientes.

Ao se treinar uma rede neural na ANNCOM, é possível escolher vários tipos de treinamento, que são encapsulados por um único treinador. Se o padrão não tivesse sido usado, cada vez que se quisesse treinar um determinado tipo de rede neural, seria necessário adicionar o algoritmo de treinamento específico dentro do cliente e essa adição consumiria mais recursos, pois o cliente teria que ser modificado. Mais do que isso, nem sempre todos os algoritmos são utilizados e não pre-

cisam estar carregados junto com o resto do sistema. A Figura 5 mostra como é possível encapsular os diferentes tipos de algoritmo.

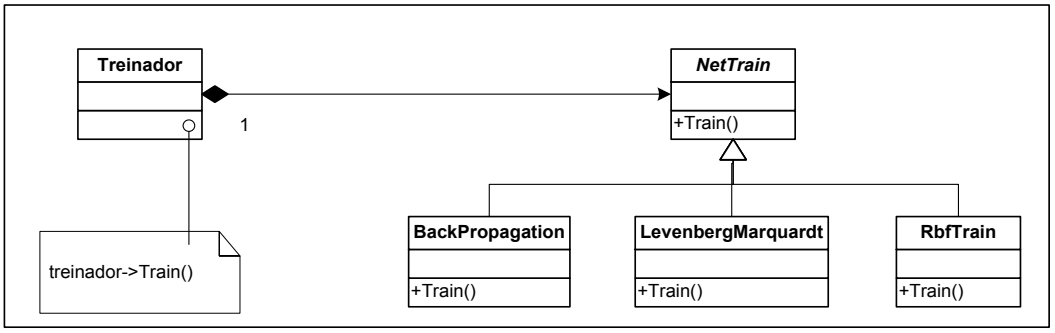


Figura 5 – Algoritmos de treinamento da ANNCOM encapsulados.

Os participantes do padrão Estratégia estão listados abaixo:

- Estratégia: este é o *NetTrain*, que declara a interface comum para todos os tipos de treinamento;
- Estratégia concreta: implementa o algoritmo usando a interface da Estratégia, por exemplo, o *BackPropagation* e *LevenbergMarquardt*.
- Contexto: o treinador *NetTrain* é um exemplo deste participante. Ele configura um tipo de treinamento, referencia-o e fornece à Estratégia o acesso aos dados.

O padrão Estratégia apresenta como principal benefício a definição de famílias de algoritmos, comportamentos e contextos relacionados para reuso. Um segundo benefício é a eliminação da necessidade de declarações condicionais encapsulando os diferentes contextos em classes de Estratégia separadas. Outro ponto positivo é a possibilidade de escolha de várias implementações para um mesmo comportamento. As conseqüências negativas são: aumento no número de objetos, o que pode ser minimizado ao se utilizar o padrão *Flyweight* (GAMMA, HELM, *et al.*, 1994); excesso de comunicação entre estratégias e contextos; e a necessidade dos clientes entenderem como cada estratégia funciona para poder escolher a mais adequada.

Existem outros vários padrões apresentados em (GAMMA, HELM, *et al.*, 1994) e muitos outros foram desenvolvidos depois da publicação do livro. Entre eles destacam-se o MVC e o DAO.

2.4. .NET Framework

Até agora foram discutidas várias abordagens para se melhorar a qualidade de sistemas de software e minimizar seus custos de desenvolvimento. Nesta seção será apresentada a base sobre a qual a ANNCOM foi montada; o *.NET framework* – o arcabouço da Microsoft.

O arcabouço da Microsoft provê suporte à programação orientada a objetos, ambiente para execução, que minimiza custos com implantação e controle de versão. Além de promover a execução segura de sistemas e permitir programação dinâmica. Essas características são possíveis, em grande parte, por causa da Linguagem Comum em Tempo de Execução (CLR – *Common Language Runtime*), que será explicada na próxima seção. Pode-se entender o componente responsável pela CLR como um agente que gerencia o código em tempo de execução provendo serviços centrais como gerência de memória, tratamento de *threads*, acesso remoto e etc. Os sistemas que são compilados para a CLR são chamados de gerenciáveis enquanto que os outros sistemas são chamados não gerenciáveis.

Os sistemas podem ser criados com código gerenciável e com código não gerenciável, possibilitando que o software usufrua das características dos dois "mundos". Por exemplo, o navegador *Web* da Microsoft foi desenvolvido em código não gerenciável, mas executa Formulários Windows e componentes gerenciáveis em documentos de Internet.

Outra característica do arcabouço da Microsoft são os modos de execução (*runtimes*). Entre eles, talvez o mais usado seja o ASP.NET, que provê execução gerenciada e escalar em servidores *Web*, permitindo o funcionamento de aplicações e serviços, ambos *Web*. A Figura 6 ilustra o relacionamento da CLR e das bibliotecas de classes com as aplicações e o resto do sistema. Nela podem-se ver as bibliotecas de classes e de componentes sendo executadas sobre o *runtime* básico e as aplicações *Web*, sendo executadas sobre o *runtime* ASP.NET (MICROSOFT, 2009).

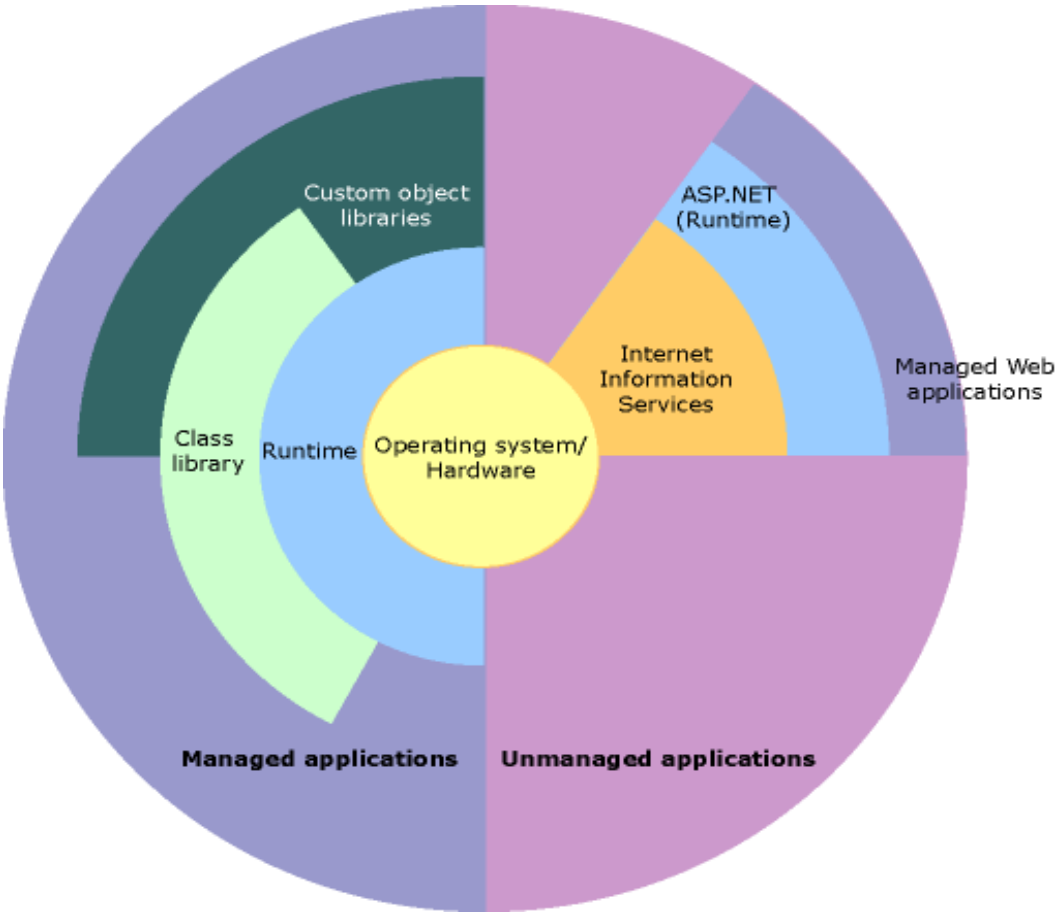


Figura 6 – O *.NET Framework* e o resto do sistema (MICROSOFT, 2009).

2.4.1. Linguagem Comum em Tempo de Execução (CLR)

Os compiladores e as ferramentas da CLR expõem as funcionalidades de cada *runtime* e permitem que o desenvolvedor escreva código usufruindo dos benefícios desses ambientes gerenciáveis de execução. O código gerenciável apresenta, além dos benefícios apresentados acima, integração entre linguagens, manipulação de exceção, segurança, controle de versões e suporte de implantação, um modelo simplificado para interação de componentes, depuração e verificação de desempenho (*profiling*). Para permitir que o *runtime* forneça esses benefícios, os compiladores CLR devem gerar um dicionário de dados (metadados), que descreve os tipos, membros e referências no código. Esses dicionários são armazenados com o código e todo componente executável contém um. O *runtime* usa os metadados para localizar e carregar classes, estabelecer instâncias na memória, resolver invocações de método, gerar código nativo, reforçar a segurança, e definir limites de contexto.

O *runtime* automaticamente trata o formato dos objetos e gerencia suas referências, liberando-os quando eles não estão mais sendo usados. Objetos cujas instâncias são geridas dessa forma são chamados dados gerenciados. O coletor de lixo (*Garbage Collector*) elimina vazamentos de memória, bem como alguns outros erros comuns de programação. Se um código é gerenciado, é possível usar dados gerenciados, dados não gerenciados, ou ambos em seu sistema. Como compiladores CLR resolvem seus próprios tipos, como os tipos primitivos, não existe a necessidade de saber se os dados estão sendo gerenciados. Além disso, a CLR facilita a criação de componentes e aplicativos cujos objetos interagem entre linguagens. Objetos escritos em linguagens diferentes podem se comunicar uns com os outros, e seus comportamentos podem ser fortemente integrados. Além disso, é possível passar uma instância de uma classe para um método de outra em linguagem diferente. Essa integração entre linguagens é possível porque os compiladores CLR e ferramentas, que usam o mesmo *runtime*, possuem seus tipos definidos por esse *runtime* e seguem suas regras para a definição de novos tipos, bem como para a criação, uso e persistência dos tipos.

Como parte de seus metadados, todos os componentes gerenciados carregam as informações sobre os objetos e recursos que foram construídos. O *runtime* usa essas informações para garantir que o sistema manterá versões específicas de tudo o que precisa, o que torna o código menos propenso a erro por causa de alguma dependência faltante. Os registros com informações e estados dos dados, como não são armazenados no Registro, são fáceis de estabelecer e manter. As informações sobre os tipos definidos, e suas dependências, são armazenadas com o código na forma de metadados. Isso torna as tarefas de replicação e remoção de componentes muito menos custosas.

Para concluir, é possível encarar os compiladores CLR e as ferramentas como facilitadores no desenvolvimento de sistemas. Entretanto, algumas características podem ser mais visíveis em um ambiente do que em outro. Na Tabela 1, são destacadas algumas das principais características das três linguagens suportadas pela versão 3.5 do arcabouço da Microsoft.

Linguagem *Visual Basic*	Melhorias de desempenho.
	Capacidade de usar com facilidade os componentes desenvolvidos em outras linguagens.
	Tipos extensíveis fornecidos por uma biblioteca de classes.

	Novos recursos da linguagem, como herança, interfaces e sobrecarga de programação orientada a objeto; suporte para *threading*, que permite a criação de vários *threads*, aplicações escaláveis; suporte para manipulação de exceção estruturada e atributos personalizados.
Linguagem C++	Integração entre linguagens, especialmente herança entre linguagens.
	Coletor de lixo, que gere a vida útil dos objetos para que a contagem de referência seja desnecessária.
	Objetos que se descrevem, torna desnecessário o uso da Linguagem de Definição de Interface (IDL).
	A capacidade de compilar uma vez e rodar em qualquer CPU e sistema operacional que suporta o *runtime*.
Linguagem C#	Orientação a objetos completa.
	Muita segurança no controle dos tipos fortemente tipados.
	Uma boa mistura entre a simplicidade do *Visual Basic* e o poder do C++.
	Coletor de lixo.
	Sintaxe e palavras-chave semelhantes ao C e C++.
	Uso de delegados, em vez de ponteiros de função, para maior segurança de tipo e de segurança. Ponteiros de função estão disponíveis através do uso da palavra-chave *unsafe* a opção */unsafe* do compilador C# (Csc.exe) para o código não gerenciado e os dados.

Tabela 1 – Principais características das linguagens *.NET*.

3
Conceitos da Computação de Alto Desempenho

A complexidade de custos de componentes mínimos tem aumentado a uma taxa de aproximadamente um fator de dois por ano...

Gordon Moore

No capítulo anterior discutiu-se como é possível construir um software para um problema complexo em menos tempo e com maiores chances de sucesso. Neste capítulo, apresenta-se uma base teórica para se construir software priorizando seu desempenho e escalabilidade.

O termo Computação de Alto Desempenho (HPC – *High-performance Computing*) está, normalmente, associado à computação científica, pois essa área compreende cálculos maiores e mais complexos. Entretanto, devido ao tamanho e a complexidade cada vez maiores dos sistemas de informação atuais, as empresas começaram a investir em soluções com maior capacidade de memória e de processamento.

Com o início da substituição da válvula, largamente utilizada nas décadas de 1950 e 1960, pelo transistor, a capacidade de processamento começou a ser definida pela quantidade e pela velocidade desses novos componentes. Na década de 1970, a criação dos circuitos integrados e, subseqüentemente, dos multiprocessadores permitiu o aumento do número desses transistores juntamente com a redução dos custos de produção dos mesmos. Até então, não havia nenhuma previsão sobre o futuro do hardware e Gordon Moore, presidente da Intel na época, declarou que o número de transistores aumentaria a uma taxa de duas vezes por ano e que em 1975 o número desses componentes chegaria a 65.000 em um único chip. Essa afirmação se provou verdadeira até os últimos anos, mas tal taxa tem começado a diminuir nos últimos anos, devido aos custos, cada vez mais altos, para pesquisa de novos processadores (JOHNSON, 2009) e limitações físicas dos chips atuais (ADVE, ADVE, *et al.*, 2008).

Para se contornar o problema da dificuldade de aumento de desempenho dos computadores modernos, novos paradigmas têm sido estudados tanto em software

quanto em hardware. Por exemplo, a computação quântica (WIKIPEDIA, 2010), a computação paralela e computação reconfigurável (WIKIPEDIA, 2010).

Entre os novos paradigmas, os que apresentam os melhores resultados são aqueles que abordam o paralelismo no desenvolvimento tanto do software quanto do hardware. Entretanto, essa mudança na forma de se conceber um novo sistema, gera vários problemas. Por exemplo, ao se implementar um algoritmo usando uma solução paralela, é necessário saber quais partes desse algoritmo podem ser processadas ao mesmo tempo e quais partes não. Além disso, os desenvolvedores precisam identificar quais pontos precisam ser sincronizados para que não se gerem inconsistências. No caso da ANNCOM, apenas uma pequena parte do algoritmo de treinamento pode ser paralelizada, já que o modelo utilizado para treinar a rede neural é iterativo e os cálculos possuem dependência entre os passos.

Além das questões de software envolvidas na escolha por sistemas paralelos, existem várias outras implicações em hardware que precisam ser levadas em conta. Por exemplo, os processadores atuais possuem vários núcleos de processamento, mas com unidades de memória de acesso rápido (*Cache Memory*) independentes, o que, em termos de desempenho, em princípio, não seria a melhor solução. Entretanto, o sincronismo dessas unidades exige um hardware muito mais complexo e mais caro, por isso os projetistas optaram por unidades de memória separadas.

Por outro lado, os sistemas paralelos apresentam grandes vantagens, dentre as quais se destacam duas: a relação “operação por dólar” e a escalabilidade. A primeira, que contabiliza o custo de cada operação em um computador, vem se tornando cada vez maior, pois a indústria de hardware tem direcionado seus esforços para a computação paralela. A segunda representa a possibilidade de o paralelismo aumentar o poder de processamento de um sistema de acordo com a demanda, pois o software distribuirá suas tarefas pelas unidades de processamento independentemente do número de unidades.

3.1. Conceitos da Computação Paralela

Na seção anterior, foram discutidos quais fatores levaram os projetistas a optar pela utilização de sistemas paralelos e foram apresentadas algumas conse-

qüências dessa escolha. Nesta seção, se descreve como esses sistemas contornam o problema do limite de desempenho dos chips atuais explicando algumas abordagens apresentadas na literatura.

A Computação Paralela é uma forma de computação onde vários cálculos são realizados simultaneamente (ALMASI e GOTTLIEB, 1990), baseando-se no fato de que grandes problemas podem ser divididos em partes menores, e essas podem ser processadas simultaneamente. Nesse tipo de sistema, o paralelismo pode se dar em diversos níveis: de bit, de instrução, de dado ou de tarefa. Cada um desses modelos será discutido mais adiante.

A técnica de paralelismo já é empregada há vários anos, principalmente na computação de alto desempenho, mas recentemente, conforme mencionado anteriormente, o interesse no tema cresceu devido a várias limitações físicas que previnem o aumento de freqüência de processamento. Além disso, com o aumento da preocupação com o consumo de energia dos computadores, a computação paralela se tornou o paradigma dominante nas arquiteturas de computadores, sob forma de processadores com mais de um núcleo. O consumo de energia de um chip é dado pela Equação 1 (YOURDON, 1988).

$$P = VF \qquad \text{Equação 1}$$

Na Equação 1, C é a capacitância, V é a tensão e F é a freqüência. Em 2004 a Intel cancelou seus modelos novos de processadores, *Tejas* e *Jayhawk*, por causa desse aumento de consumo em relação à freqüência de processamento. Esse episódio é normalmente citado como o fim da freqüência de processamento como paradigma predominante nas arquiteturas de computador (FLYNN, 2004).

3.1.1. Leis de *Amdahl* e *Gustafson*

Com a utilização de vários núcleos de processamento em um único chip, o aumento de velocidade com o paralelismo deveria ser idealmente linear, de forma que com n núcleos, o tempo de execução se reduzisse n vezes. Entretanto, muitos poucos algoritmos paralelos atingem essa situação, uma vez que a maioria deles fornece aumento de velocidade quase linear para poucos núcleos de processamento, tendendo a um valor constante para uma quantidade maior de elementos.

O aumento de velocidade potencial de um algoritmo em uma plataforma de computação paralela é dado pela Lei de Amdahl (AMDAHL, 1967), formulada por Gene Amdahl na década de 1960. Essa lei afirma que uma pequena parte do programa, que não pode ser paralelizada, limitará o aumento de velocidade geral que o paralelismo poderia oferecer. Os algoritmos consistem, normalmente, de partes paralelizáveis e partes seqüenciais. A relação entre os dois é dada pela Equação 2.

$$S = \frac{1}{(1 - P)} \qquad \text{Equação 2}$$

Na Equação 2, *S* é o aumento de velocidade do programa e *P* é a fração paralelizável. Se a parte seqüencial de um programa representar 10% do tempo de execução, não se poderá obter mais do que dez vezes de aumento de velocidade, não importando o número de processadores, o que tornará inútil a adição de mais unidades de execução. Na Figura 7, a curva azul ilustra o aumento linear que um programa teria no caso ideal, enquanto a curva rosa indica o aumento real. Da mesma forma, a curva amarela indica o tempo de execução no caso ideal, enquanto a curva vermelha indica o tempo de execução real.

Uma representação gráfica é apresentada na Figura 8, assumindo que uma tarefa possui duas partes independentes: *A*, que ocupa cerca de 70% de todo tempo de execução, e *B*, que ocupa o restante. Nesse exemplo, se um programador tornar a parte *B* cinco vezes mais rápida, o tempo total de execução será reduzido em mais de 20%. Por outro lado, com menos esforço pode-se tornar a parte *A* duas vezes mais rápida, o que reduzirá o tempo total de execução em mais de 30%.

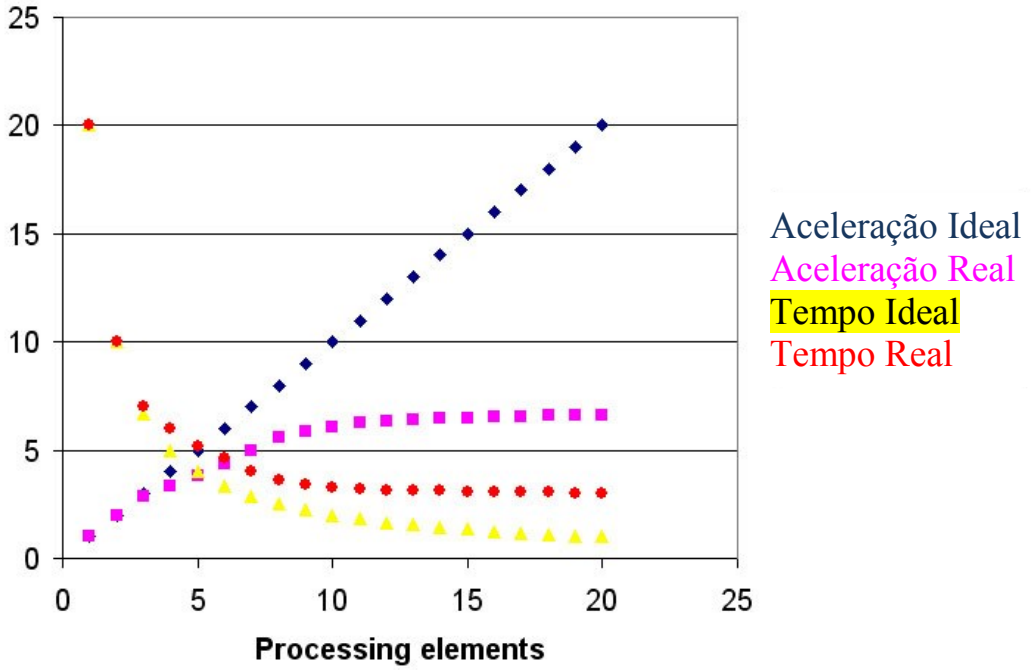


Figura 7 – O tempo de execução e o aumento de velocidade de um programa com paralelismo.

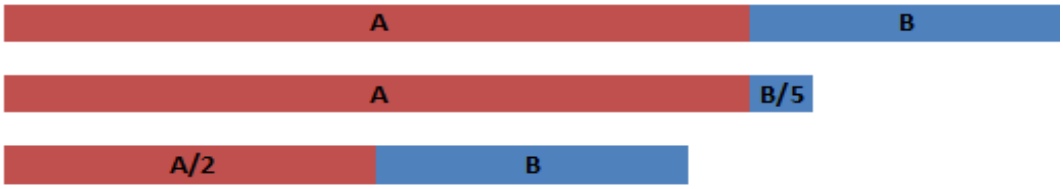


Figura 8 – Representação gráfica da lei de Amdahl.

A Lei de Gustafson (GUSTAFSON, 1988) também relaciona aumento de velocidade com fração paralelizável, mas não assume um tamanho de problema fixo e nem que o tamanho da parte seqüencial é independente do número de processadores. A Equação 3 formula a Lei de Gustafson, onde P é o número de processadores, S é o aumento de velocidade e α é a parte não paralelizável do processo.

$$S(P) = P - \alpha\ (P - 1) \qquad \text{Equação 3}$$

3.1.2. Tipos de Paralelismo

Uma das primeiras metodologias de classificação para computadores e programas paralelos e seqüenciais foi criada por Michael J. Flynn (FLYNN, 2004).

Atualmente conhecida como taxonomia de Flynn, ela classifica os programas e computadores por quantidade de fluxos de instruções e por quantidade de dados usada por tais instruções. A Tabela 2 apresenta as quatro classes propostas por Flynn, onde a classe SISD representa um sistema completamente seqüencial e o MIMD representa um dos sistemas paralelos mais usados atualmente. Um exemplo de um sistema MIMD é a computação distribuída, que consiste de várias máquinas ligadas em rede e cujo objetivo é realizar uma tarefa em comum.

	Única Instrução (*Single Instruction*)	Múltiplas Instruções (*Multiple Instruction*)
Único Dado (*Single Data*)	SISD	MISD
Múltiplos Dados (*Multiple Data*)	SIMD	MIMD

Tabela 2 – Taxonomia Flynn.

Além da taxonomia Flynn, pode-se classificar os sistemas paralelos segundo o nível em que se dá o paralelismo: (a) no bit; (b) na instrução; (c) no dado; (d) na tarefa.

A partir do advento da tecnologia de fabricação de chip VLSI na década de 1970 até cerca de 1986, o aumento da velocidade dos computadores era obtido dobrando-se o tamanho da palavra, que determina a quantidade de informação que pode ser processada por ciclo (CULLER, SINGH e GUPTA, 1998). Aumentando o tamanho da palavra, se reduz a quantidade de instruções que um processador deve executar para realizar uma operação em variáveis cujo tamanho é maior do que o da palavra. Historicamente, microprocessadores de quatro bits foram substituídos pelos de oito, depois pelos de dezesseis e mais tarde pelos de trinta e dois bits. A partir de então, o padrão 32-bit se manteve na computação de uso geral por duas décadas. Em 2003, a arquitetura 64-bit começou a ganhar mais espaço.

Considerando que, em sua essência, um programa de computador é um fluxo de instruções executadas por um processador, e que a execução de cada instrução exige diversas sub-tarefas, tais como busca, decodificação, cálculo e armazenamento de resultados, pode-se promover o paralelismo de tais sub-tarefas. Esse método é conhecido por paralelismo em instrução ou técnica de *pipeline*. Avanços nessa técnica dominaram as arquiteturas de computadores de meados da década de 1980 até meados da década de 1990 (CULLER, SINGH e GUPTA, 1998). Pro-

cessadores modernos possuem *pipelines* com múltiplos estágios. Cada estágio corresponde a uma ação diferente que o processador executa em determinada instrução; um processador com um *pipeline* de N estágios pode ter até N diferentes instruções em diferentes estágios de execução.

O paralelismo em dado é inerente a laços de repetição, focando em distribuir as iterações, ou o processamento de diversos dados, por diferentes nós computacionais para serem executadas em paralelo. Diversas aplicações científicas e de engenharia apresentam esse tipo de paralelismo. Uma dependência por laço é a dependência de uma iteração do laço com a saída de uma ou mais iterações anteriores, uma situação que impossibilita a paralelização de laços. Por exemplo, na ANNCOM, o treinamento é iterativo e cada época depende do resultado da anterior. Com o aumento do tamanho de um problema, geralmente aumenta-se a quantidade de paralelismo em dado disponível (CULLER, SINGH e GUPTA, 1998).

O paralelismo em tarefa é a característica de um programa paralelo em que diferentes cálculos são realizados no mesmo ou em diferentes conjuntos de dados (CULLER, SINGH e GUPTA, 1998). Isso contrasta com o paralelismo em dado, em que o mesmo cálculo é feito em diferentes conjuntos de dados. Paralelismo em tarefa geralmente não é escalável com o tamanho do problema (CULLER, SINGH e GUPTA, 1998). As placas gráficas atuais da NVIDIA adotaram uma abordagem conhecida como SIMT (*Single Instruction Multiple Thread*), e que está entre o paralelismo de tarefas e o paralelismo de dado, uma vez que a GPU executa várias *threads* ao mesmo tempo ao invés de uma *thread* com uma palavra grande. A tecnologia de placas gráficas será explicada na próxima seção.

3.2. Os Processadores Gráficos e a GPGPU

Até agora foram apresentadas algumas arquiteturas evolvendo vários processadores ou vários computadores. Entretanto, com a necessidade de cálculos muito complexos e soluções em tempo real, surge uma demanda por sistemas com tamanho grau de paralelismo que restringe a utilização das abordagens anteriores. O principal motivo disso é o custo financeiro e ambiental de se construir e se manter um espaço com vários computadores, os quais precisam ser mantidos em am-

bientes controlados e gerenciados por grandes sistemas e mão de obra especializada.

Ao passo que essa necessidade de poder computacional crescia, as placas gráficas estavam se tornando poderosos computadores, extremamente paralelizados, por causa da indústria do entretenimento digital e sua demanda por gráficos em alta definição. As Figura 9 e Figura 10 mostram um comparativo entre alguns processadores e algumas placas de vídeo, com relação ao desempenho e o acesso à memória.

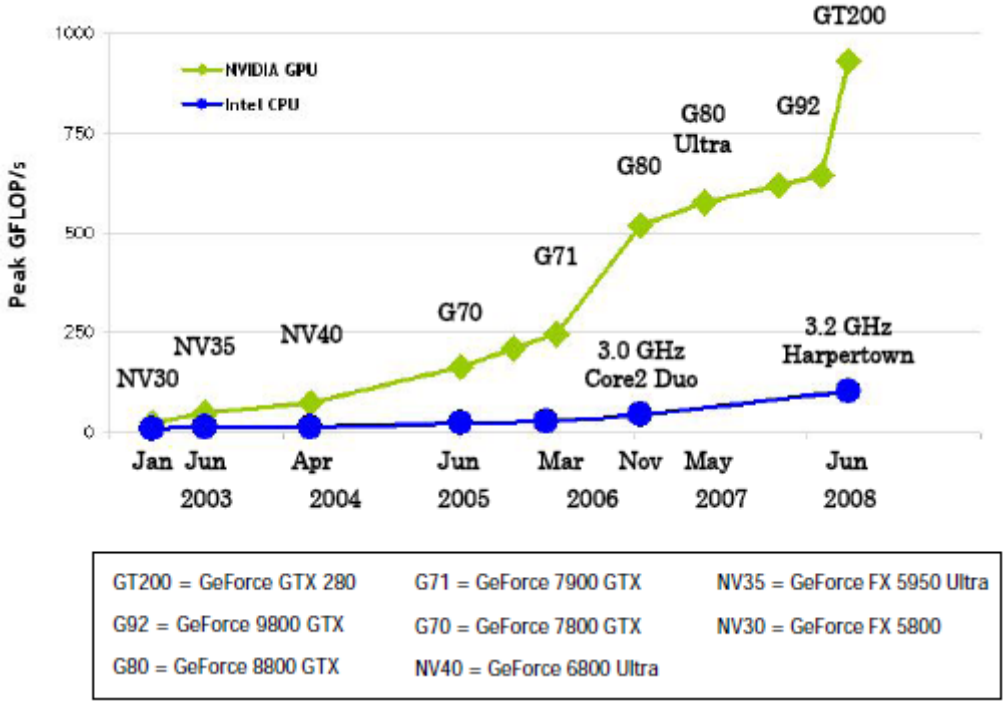


Figura 9 – Comparativo entre CPU e GPU com relação às operações de ponto flutuante por segundo (NVIDIA, 2009).

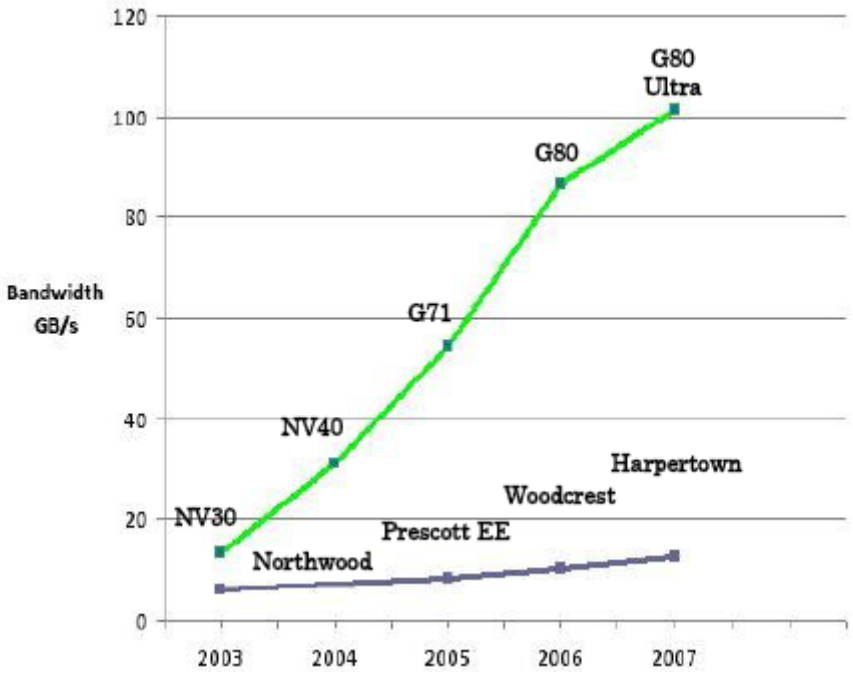


Figura 10 – Comparativo entre CPU e GPU com relação à largura de banda de memória memória (NVIDIA, 2009).

A razão da enorme discrepância entre os processadores convencionais e os processadores gráficos é que uma GPU (*Graphics Processing Unit*), ao contrário da CPU, utiliza muito mais transistores para processamento do que para controle de fluxo ou memória de acesso rápido (NVIDIA, 2009), conforme é ilustrado na Figura 11. Percebe-se que essa arquitetura se adapta melhor a problemas cujos dados possam ser quebrados em partes menores e processados de forma paralela, mas simples, como cálculos aritméticos, para que o mesmo programa seja executado nas pequenas partes ao mesmo tempo, dispensando, dessa forma, um controle de fluxo sofisticado. Além disso, como existem muitos cálculos aritméticos a serem executados, a latência de acesso à memória fica encoberta pelo tempo dos cálculos, dispensando o uso de grandes memórias de acesso rápido (NVIDIA, 2009).

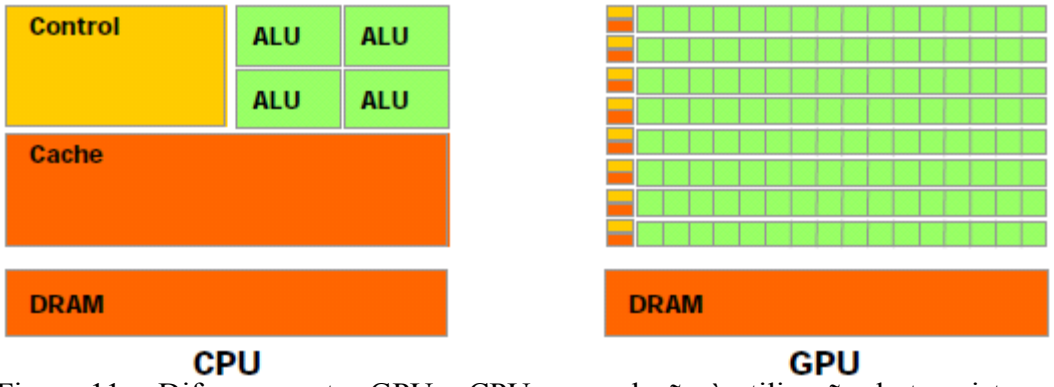


Figura 11 – Diferença entre GPU e CPU com relação à utilização de transistores (NVIDIA, 2009).

A tecnologia de chips gráficos se mostrou bem interessante para resolver alguns dos problemas antes aceitos como insolúveis considerando a tecnologia atual, além da possibilidade de acelerar outros. A utilização de um ou mais processadores gráficos para executar tarefas tradicionalmente feitas pelos processadores comuns ficou conhecida como Computação de Propósito Geral em Processadores Gráficos (*General-Purpose Computing on Graphics Processing Units* – GPGPU).

3.3. CUDA – NVIDIA

Em novembro de 2006, a NVIDIA lançou o CUDA, uma arquitetura para computação paralela de propósito geral que utiliza as placas gráficas da NVIDIA. A tecnologia CUDA oferece um ambiente que permite o desenvolvimento em linguagem C, C++, FORTRAN, OpenCL e, futuramente, DirectX Compute, conforme mostra a Figura 12 (NVIDIA, 2010).

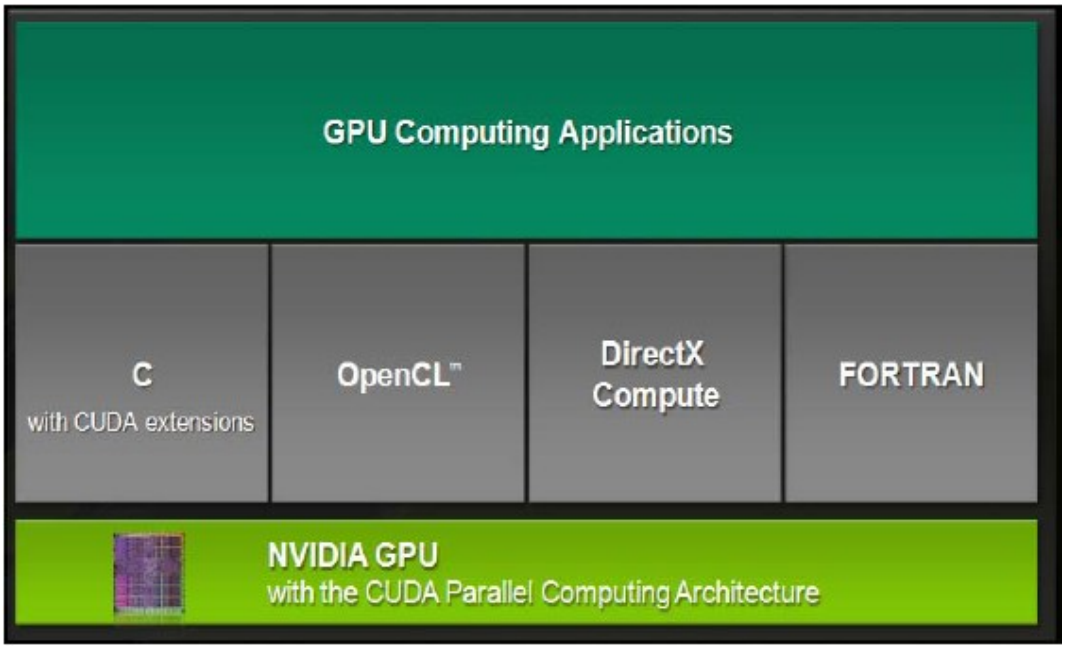


Figura 12 – Suporte de CUDA para várias linguagens (NVIDIA, 2009).

A arquitetura da NVIDIA possui três níveis de abstração: a hierarquia de grupos de *threads*, as memórias compartilhadas e as barreiras de sincronização. Isso torna a curva de aprendizado da linguagem relativamente pequena e requer poucas extensões para as linguagens de programação. Além disso, as abstrações provêem paralelismo tanto no dado quanto nas *threads*, necessitando, por parte do programador, apenas de uma simples divisão de dados e tarefas (NVIDIA, 2009).

Outra vantagem das abstrações é que elas ajudam o desenvolvedor a dividir o problema em tarefas menores, que podem ser resolvidas de forma independente e, conseqüentemente, paralela. Tal decomposição é feita de modo que as *threads* possam cooperar entre si ao resolver as sub-tarefas e, ao mesmo tempo, possibilitar a escalabilidade, uma vez que as *threads* podem ser agendadas para serem resolvidas em qualquer núcleo disponível. Um programa compilado em CUDA pode, nesse caso, ser executado em uma máquina independentemente do número de processadores, o qual será verificado em tempo de execução (NVIDIA, 2009). Dessa forma, o modelo de programação permite à arquitetura abranger vários tipos de máquinas, com placas variando de algumas dezenas até milhares de núcleos.

3.3.1.
O Modelo de Programação

Nesta seção, apresentar-se-á o modelo de programação da arquitetura CUDA. Conforme mencionado na seção 3.3, a arquitetura CUDA fornece algumas extensões para a linguagem C. Elas permitem que o programador invoque funções, chamadas *kernels*, que serão executadas *N* vezes em *N threads* diferentes. Cada *thread* possui um identificador único permitindo que o desenvolvedor acesse cada uma delas.

As *threads* são hierarquizadas em blocos tridimensionais (a Figura 13 faz uma representação bidimensional para simplificar a visualização) e esses são organizados em grades de duas dimensões – ver Figura 13.

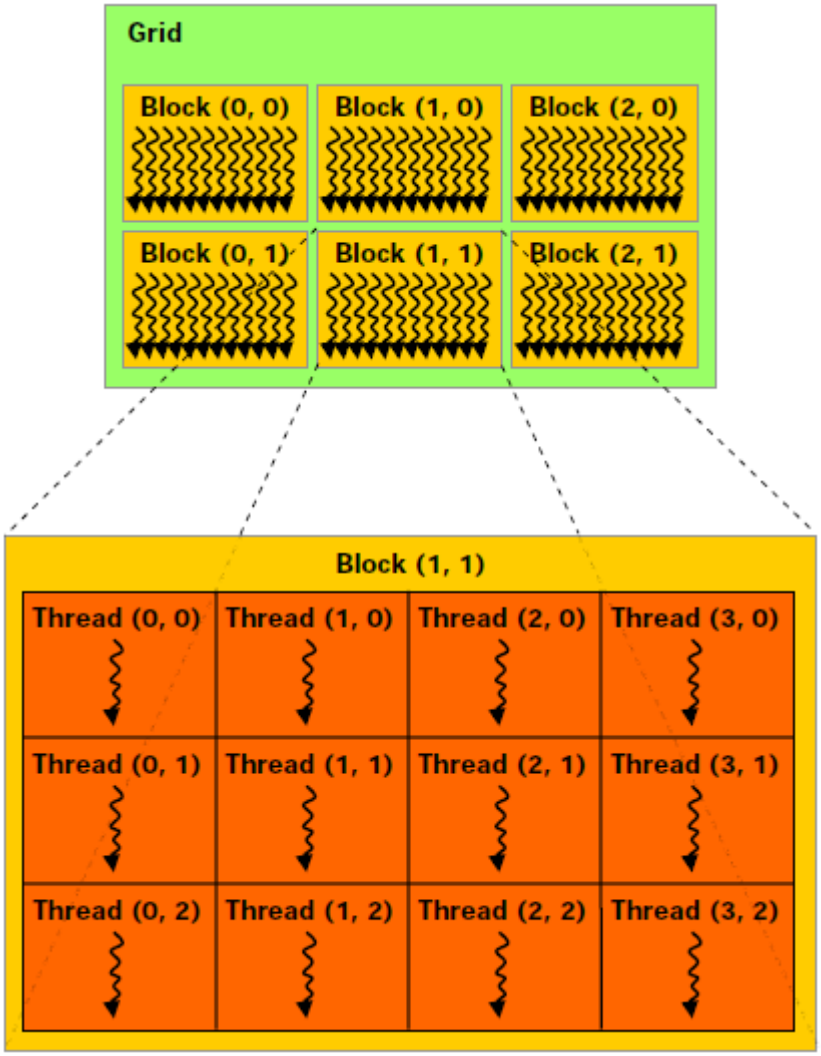


Figura 13 – Grade de blocos de *threads* (NVIDIA, 2009).

Isso torna mais fácil o entendimento ao se computar elementos vetoriais e matriciais, uma vez que os índices das *threads* são seqüenciais dentro dos blocos. Por exemplo, num espaço de duas dimensões de tamanho (D_x, D_y), a posição (x, y) terá o índice $(x + yD_x)$, e se o espaço for de três dimensões com tamanho (D_x, D_y, D_z), a posição (x, y, z) terá o índice $(x + yD_x + zD_xD_y)$.

As *threads* de um mesmo bloco podem cooperar entre si através do sincronismo de suas execuções e de uma memória compartilhada (*shared memory*). Para funcionar de forma eficiente, a memória compartilhada possui baixa latência e fica próxima dos núcleos, podendo ser comparada com a memória L1 de uma CPU. A única restrição ao uso do bloco de *threads* e sua memória compartilhada é seu tamanho limitado. Entretanto, é possível dividir todo o *kernel* em blocos iguais e organizá-los numa grade para que, o máximo possível deles, seja executado ao mesmo tempo pela placa. Nesse caso, o número de *threads* de uma função a ser executada será o número de *threads* por bloco vezes o número de blocos.

Além da memória compartilhada, a placa de vídeo possui a memória global, que pode ser acessada pelas *threads*. Essa, entretanto, é compartilhada com todo o programa, ao invés de apenas dentro dos blocos, e possui duas áreas somente para leitura, destinadas a constantes e a texturas. A última pode ser mapeada através de mapas de textura. A Figura 14 apresenta os diferentes níveis de memória existentes na placas gráficas da NVIDIA. Tal hierarquia de memória resulta do fato do modelo de programação CUDA assumir que as *threads* são executadas em um dispositivo fisicamente separado do computador, como um co-processador. Por exemplo, se um programa possuir partes em *kernel* e partes normais, a primeira será executada na GPU e a CPU executará a segunda.

O modelo de programação CUDA assume que tanto o computador quanto a placa gráfica possuem suas próprias memórias chamadas, respectivamente, de *host memory* e *device memory*. Com isso, o programa precisa gerenciar a memória global, a de constantes e a de texturas, além da memória do computador e as transferências entre computador e placa de vídeo.

As características da arquitetura permitem que sejam desenvolvidos programas chamados heterogêneos – seqüenciais e paralelos – que são executados ao mesmo tempo pela GPU e pela CPU. A Figura 15 ilustra esse tipo de programação.

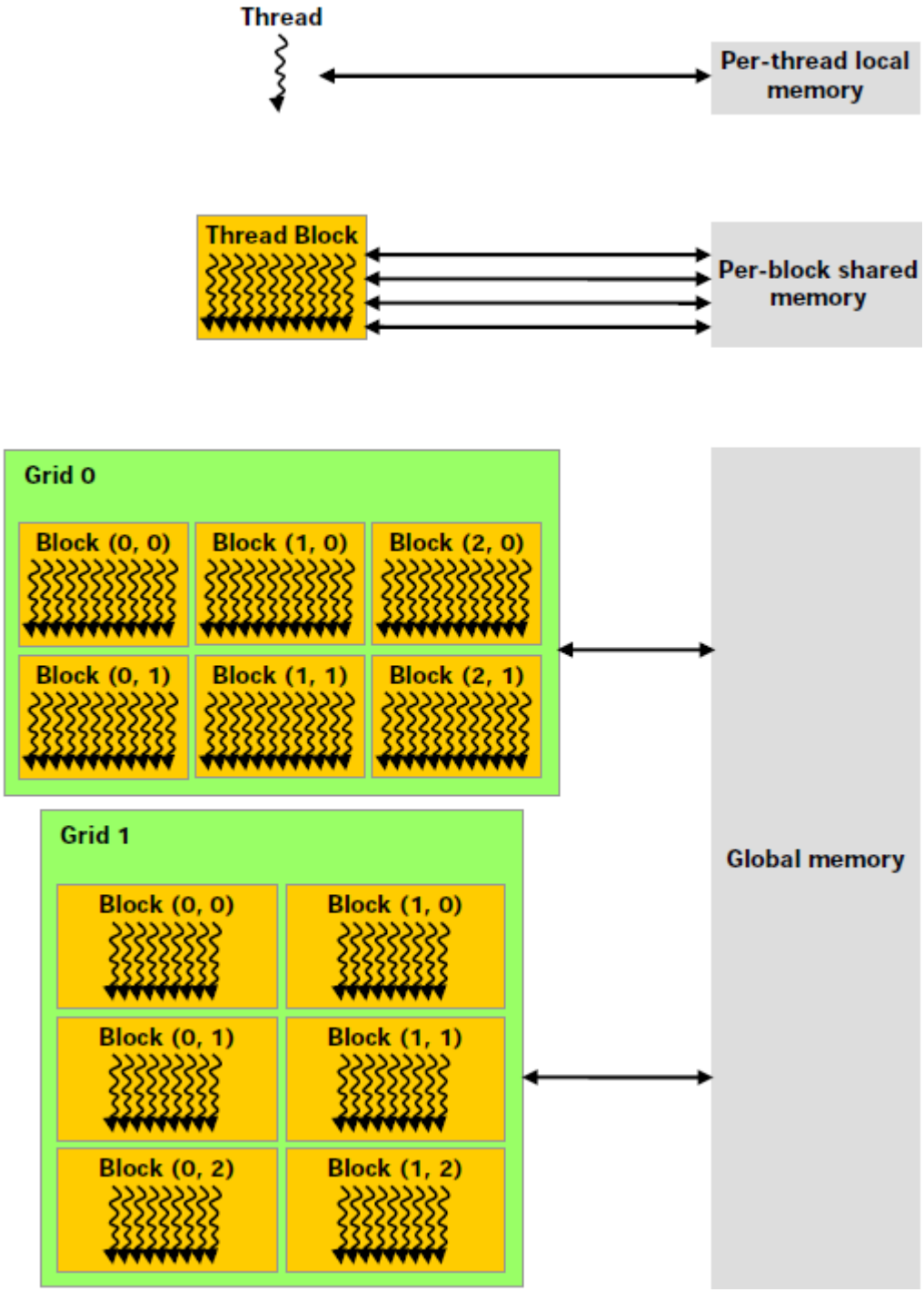


Figura 14 – Os diferentes níveis de memória nas placas de vídeo da NVIDIA (NVIDIA, 2009).

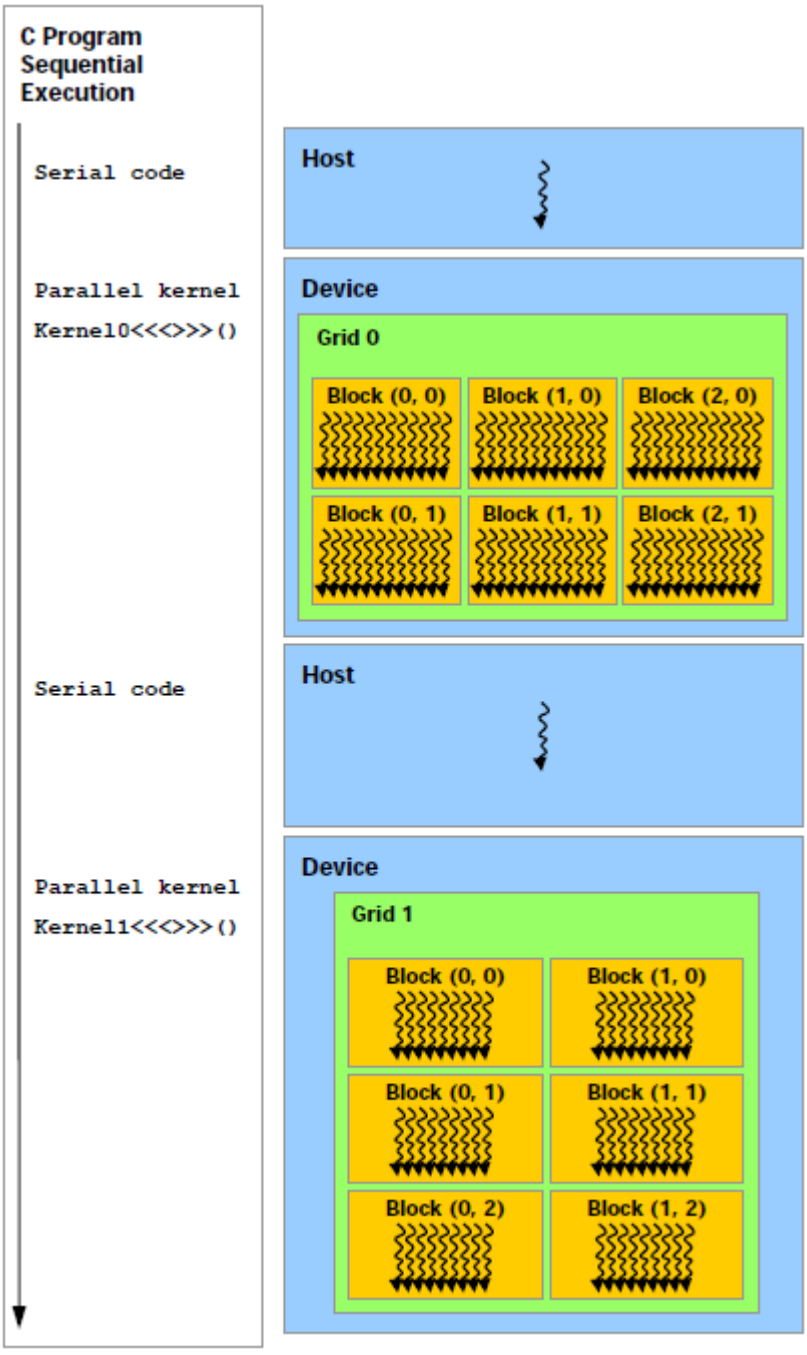


Figura 15 – Modelo de programação heterogêneo (NVIDIA, 2009).

A arquitetura CUDA suporta, atualmente, duas interfaces para se codificar programas: "*C for CUDA*" e "*CUDA Driver API*". Elas são exclusivas e não se pode misturá-las em um mesmo programa. A primeira interface vem com uma API (*Application Programming Interface*) usada em tempo de execução e adiciona apenas algumas instruções à linguagem C. Já a segunda é uma API em C, que fornece funções para carregar, como módulos, os binários CUDA, verificar seus parâmetros e executá-los. Vale salientar que o código escrito em *C for CUDA* é

mais conciso, mais simples e suporta depuração, uma vez que é montado diretamente sobre a outra interface. Entretanto, códigos usando *CUDA Driver API* oferecem maior nível de controle e são independentes de linguagem, uma vez que podem trabalhar diretamente com binários ou código *assembly* (NVIDIA, 2009).

3.3.2. A Implementação do Hardware

Na seção anterior discutiu-se o modelo de programação e as interfaces da arquitetura CUDA. Para concluir a apresentação da tecnologia de computação de propósito geral da NVIDIA, descreve-se o hardware das placas de vídeo que suportam a tecnologia.

A arquitetura CUDA foi construída sobre uma matriz de multiprocessadores (*Streaming Multiprocessors – SM*) (NVIDIA, 2009). Ao se invocar um *kernel* (dividido em uma grade) em CUDA numa CPU, os blocos de *threads* são enumerados e distribuídos para os multiprocessadores disponíveis conforme mostra a Figura 16. As *threads* de um bloco são executadas, de forma concorrente, em um multiprocessador, e quando esse bloco é finalizado, outro bloco é inicializado até que toda a grade seja finalizada. Mais do que isso, se uma placa gráfica possuir *N* multiprocessadores, ela poderá executar até *N* blocos ao mesmo tempo.

Os multiprocessadores possuem oito núcleos chamados Processadores Escalares (*Scalar Processors – SP*), duas unidades para números transcendentais (ex: π, e, etc.), uma unidade para controle das *threads* e a memória compartilhada. Os multiprocessadores criam, gerenciam e executam as *threads*, em hardware, e sem custo. Além disso, a barreira de sincronismo entre as *threads* é implementada em uma única instrução e resolvida de maneira muito eficiente (NVIDIA, 2009).

Para conseguir gerenciar centenas ou milhares de *threads* ao mesmo tempo, os multiprocessadores empregam a nova arquitetura chamada SIMT (Uma Instrução Múltiplas *Threads – Single-Instruction Multiple-Thread*). Nela cada *thread* é alocada a uma SP, que a executa independentemente. A unidade SIMT do multiprocessador cria, gerencia, agenda e executa as *threads* em grupos chamados *warps*. Os integrantes desses grupos são inicializados ao mesmo tempo, mas possuem liberdade para parar ou executar de forma independente.

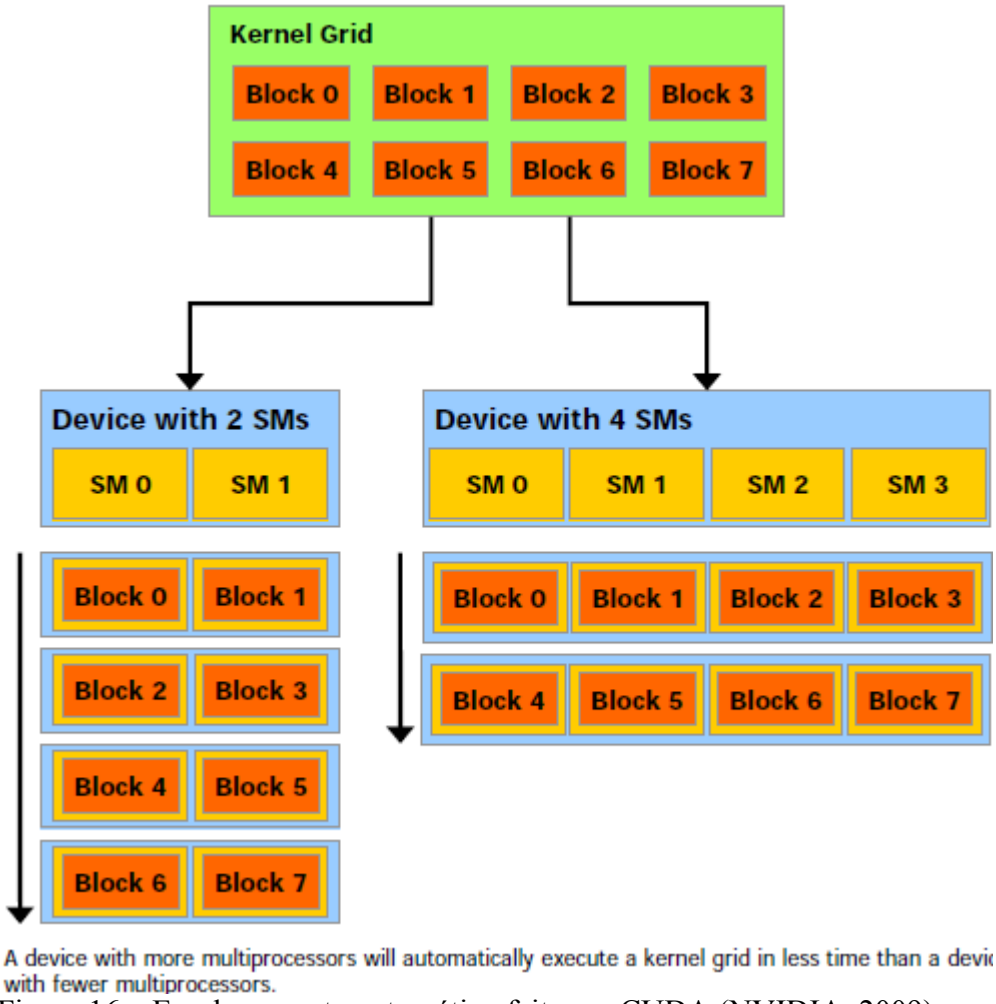
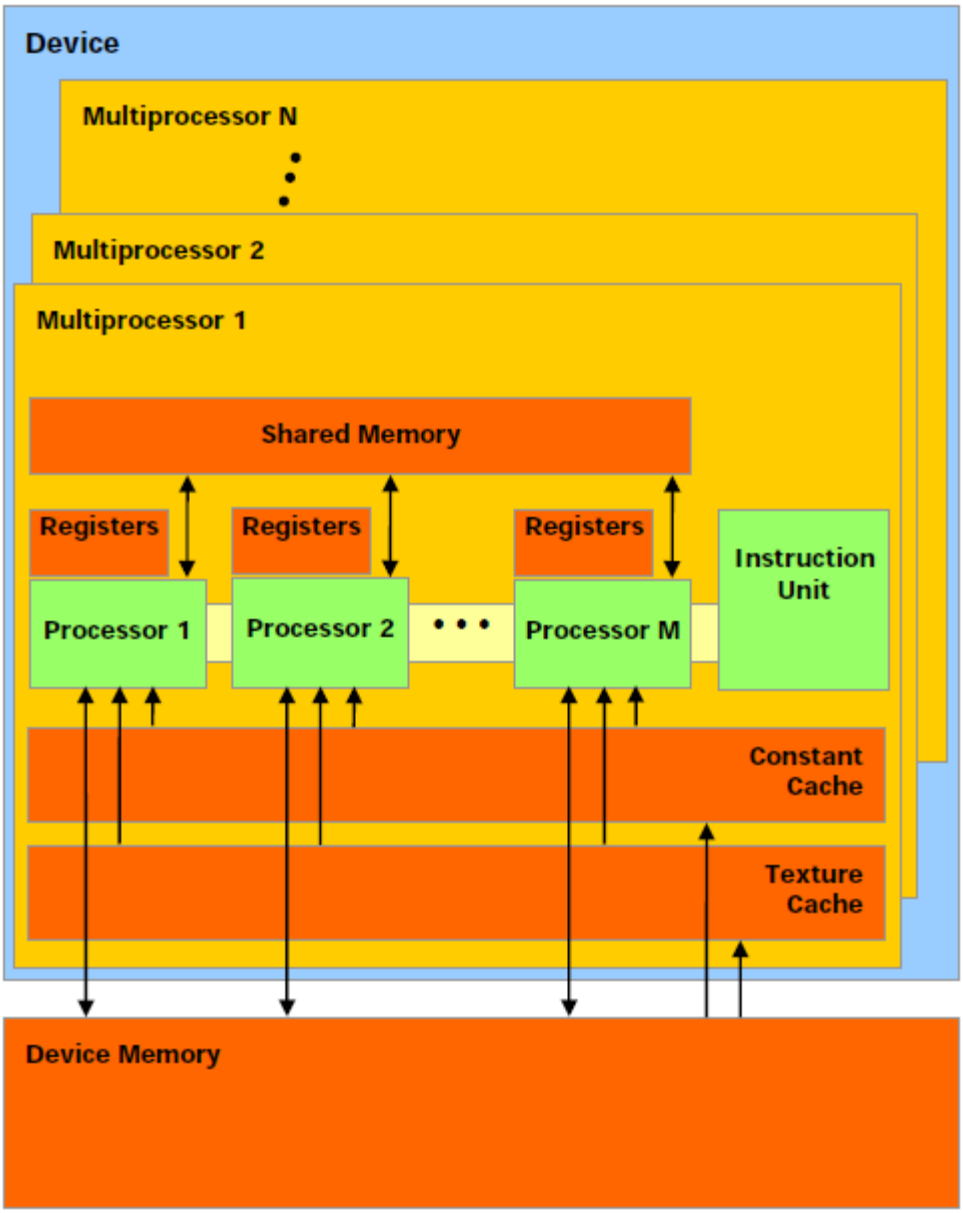


Figura 16 – Escalonamento automático feito em CUDA (NVIDIA, 2009).

A arquitetura SIMT é parecida com a arquitetura SIMD, onde um vetor de dados é processado pela mesma instrução, mas com uma diferença muito importante: na arquitetura SIMD, o tamanho do vetor é conhecido pelo hardware, enquanto que na arquitetura SIMT, as instruções especificam a execução e o comportamento de uma única *thread*.

Para concluir esta seção, a Figura 17 ilustra a estrutura da placa com os multiprocessadores e seus componentes, além dos diferentes tipos de memórias existentes.



A set of SIMT multiprocessors with on-chip shared memory.

Figura 17 – Modelo de hardware de uma placa gráfica com suporte a CUDA (NVIDIA, 2009).

4
Desenvolvimento de uma Biblioteca de Redes Neurais de Alto Desempenho – ANNCOM

Nos capítulos anteriores foram apresentados alguns conceitos sobre o desenvolvimento de sistemas considerando questões de qualidade e de desempenho, e a utilização de placas gráficas para computação de propósito geral. Neste capítulo, será apresentada uma biblioteca de redes neurais que, ao levar em consideração as questões acima, alcançou resultados muito significativos com relação às ferramentas e sistemas existentes no mercado.

As redes neurais são muito úteis para a solução de problemas complexos de previsão ou aproximação de funções, e já existem no mercado, há algum tempo, bibliotecas que auxiliam a modelagem, a criação, o treinamento e o teste dos vários tipos de redes conhecidos. Entretanto, tais bibliotecas apresentam limitações quando o tamanho do problema ultrapassa determinado limite de complexidade, por exemplo, devido à alta quantidade de informação (registros ou atributos) contida na base de dados utilizada nas fases de treinamento, validação e teste.

Além do tamanho da base, existe também um equilíbrio entre a complexidade de cada iteração do algoritmo usado no treinamento e a melhora do acerto da rede a cada iteração, pois isto afeta diretamente o tempo total de treinamento. Por exemplo, se uma rede neural é treinada por um algoritmo de retro-propagação com gradiente comum, o custo de cada passo é relativamente pequeno, mas são necessários mais passos para se treinar a rede; por outro lado, um algoritmo quasi-newton requer muito menos iterações, mas com um custo computacional muito maior.

A biblioteca de redes neurais desenvolvida neste trabalho utiliza a capacidade computacional das placas de vídeo e uma estrutura diferenciada com o objetivo de reduzir o tempo de aprendizado, de forma a suportar problemas mais complexos e bases de dados maiores. A maior parte da ANNCOM foi escrita na linguagem C#, sobre a plataforma .NET 3.5, e o restante, em linguagem C estendida para CUDA.

4.1. Levantamento de Requisitos e Arquitetura

Para iniciar o desenvolvimento da biblioteca, analisaram-se as necessidades de aplicações típicas e fez-se um levantamento das ferramentas utilizadas na área de redes neurais artificiais. Das necessidades encontradas, as mais importantes foram:

- Ter um sistema modular para criação de redes neurais que possa ser utilizado diversas vezes, minimizando o retrabalho e facilitando extensões e trabalhos de manutenção;
- Treinar rapidamente muitas redes neurais com bases de dados com muitos registros e atributos.

Essas necessidades transformaram-se em requisitos não funcionais e levaram à escolha da plataforma .NET, que suporta linguagens orientadas a objetos e oferece flexibilidade e robustez, além de ser uma plataforma voltada a alta produtividade. Outro fator que influenciou na escolha da linguagem C# foi o suporte dado pela ferramenta de desenvolvimento da Microsoft, o Visual Studio.

Ao se fazer o levantamento das ferramentas da área de redes neurais, respeitaram-se os seguintes critérios: grau de utilização (quantos laboratórios ou empresas utilizam a ferramenta) e preço. Entre os softwares testados, o Matlab (The MathWorks, Inc., 2010) foi selecionado para testes de validação e de desempenho dos treinamentos das redes, e a MKL (Intel Corporation, 2010) foi utilizada para comparações de desempenho dos cálculos numéricos.

Além dos requisitos principais apresentados anteriormente, outros requisitos levantados foram:

- Requisitos Funcionais
    - Trabalhar com arquivos XML para carregar e salvar configurações de redes e resultados de treinamento;
    - Possibilitar a inserção dos componentes de redes neurais na caixa de ferramentas do Visual Studio, permitindo a criação de redes em tempo de desenvolvimento (*Design Time*), por meio de simples clique e arraste, além da configuração por meio de janelas;

- Suportar conexões com diversos tipos de bancos de dados;
- Possibilitar a compatibilidade com o Matlab por meio de funções que gerem vetores de pesos.

- Requisitos Não Funcionais
  - Apresentação de uma interface gráfica para modelagem, criação, treinamento, validação e teste de redes neurais e comitês de redes neurais (onde, por exemplo, várias redes votam em uma classe).

A partir das bases teóricas sobre arquitetura de software apresentadas no capítulo 2 e dos requisitos supracitados, pode-se obter uma visão geral do sistema, conforme mostra a Figura 18. Nessa figura, os componentes brancos (Elman, RBF e MLP) são instâncias do componente de redes neurais, chamado *NeuralNet;* os cinza-claros são instâncias do treinamento chamado *NetTrain*, e o componente *JacobianCUDA* é a parte do sistema desenvolvida utilizando-se a tecnologia da NVIDIA, com o objetivo de atender à restrição de desempenho do treinamento.

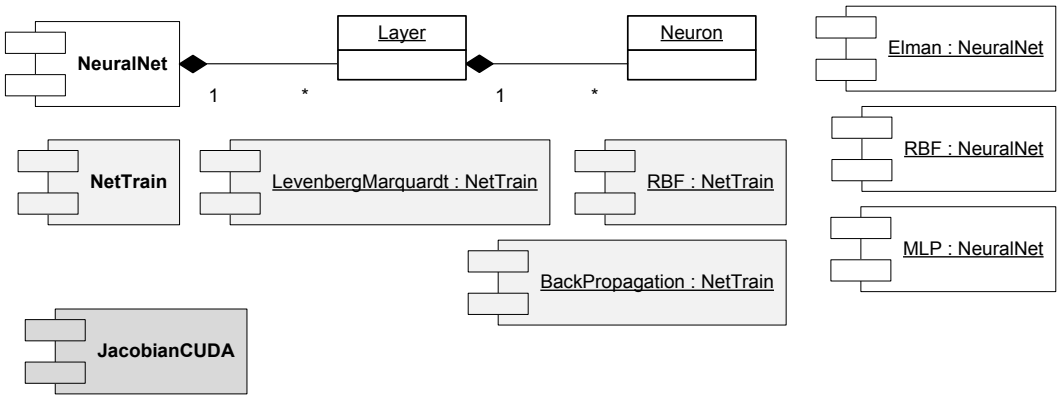


Figura 18 – Visão geral dos componentes principais da ANNCOM.

A arquitetura foi decomposta em componentes orientados a objetos de forma a suportar novas funcionalidades, como um novo algoritmo de treinamento, um novo tipo de rede neural, um comportamento diferente para os neurônios, etc. Essa característica não foi encontrada em nenhuma das ferramentas avaliadas Por exemplo, nelas não é possível propagar um sinal por uma rede MLP utilizando uma função de ativação diferente das já existentes, ou ajustar individualmente todos os parâmetros das gaussianas, inclusive o desvio padrão, dos neurônios no treinamento de redes radiais (o Matlab, por exemplo, utiliza um único valor de

desvio padrão para todos os neurônios da camada escondida). Na ANNCOM, para adicionar alguma funcionalidade nova, basta criar um novo componente derivado de um mais genérico (*NeuralNet*, *NetTrain*, etc.) e implementar a funcionalidade nova. Ao se criar um componente derivado, herda-se todas as funcionalidades e comportamento do componente pai, o que facilita e torna mais produtivo o desenvolvimento, uma vez que só é necessário implementar a parte adicional. No caso do Matlab, vale lembrar que é possível desenvolver funções novas, mas sem a facilidade de se herdar funcionalidades previamente desenvolvidas.

Outro detalhe da arquitetura é o suporte a conexões com vários tipos de gerenciadores de bancos de dados (SGBD – Sistema Gerenciador de Banco de Dados). Essa funcionalidade, apoiada por classes dedicadas à interface com bancos de dados disponíveis no arcabouço da Microsoft, permite que as redes neurais se conectem com diversos tipos de SGBD diferentes, facilitando a portabilidade e a implantação dos sistemas que utilizam a biblioteca, conforme mostra a Figura 19. Além disso, através de uma conexão direta entre a biblioteca e o SGBD, pode-se passar todo o processamento dos dados para o SGBD, de forma a respeitar políticas mais rígidas de manipulação de dados, muitas vezes adotadas pelos setores de TI (Tecnologia da Informação). Nesse caso, a TI fica responsável pela gerência dos procedimentos dentro do SGBD, enquanto a biblioteca apenas faz a leitura, sem ter a necessidade de processos para exportação e importação dos dados.

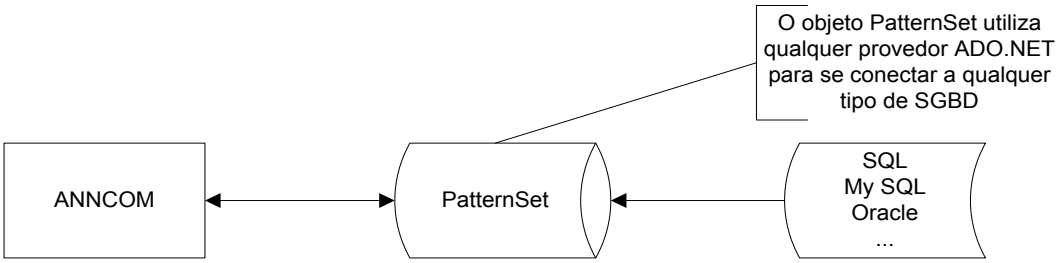


Figura 19 – Suporte a conexão com vários tipos de SGBD.

Para facilitar o desenvolvimento de novos sistemas com a ANNCOM, os componentes principais podem ser adicionados à caixa de ferramentas do Visual Studio (Microsoft Corporation, 2010). Com isso, no desenvolvimento de aplicações novas, o usuário só precisa clicar e arrastar novas redes ou treinamentos para o sistema que está sendo desenvolvido, e editá-los de forma visual e intuitiva. A Figura 20 ilustra a utilização dessa funcionalidade, apresentando, no lado direito,

a janela de ferramentas (*Toolbox*) com três componentes (*NeuralNet*, *GradientDescendent* e *LevenbergMarquardt*). Na parte de baixo da figura, encontram-se duas instâncias de componentes já adicionadas ao projeto (*neuralNet1* e *levenbergMarquardt1*); o componente de treinamento aparece selecionado. Na lateral esquerda da figura, estão listadas as propriedades do componente selecionado. Essas propriedades podem ser alteradas e verificadas em tempo real e, dessa forma, o desenvolvimento de um novo sistema se torna mais intuitivo, já que fica disponível na tela, toda a informação sobre o componente.

Como as redes neurais artificiais possuem uma estrutura com muitas coleções (uma rede possui um coleção de camadas, as camadas possuem uma coleção de neurônios e estes possuem uma coleção de sinapses), a ANNCOM também oferece um editor com janelas para facilitar a manipulação de tais estruturas, como mostra a Figura 21.

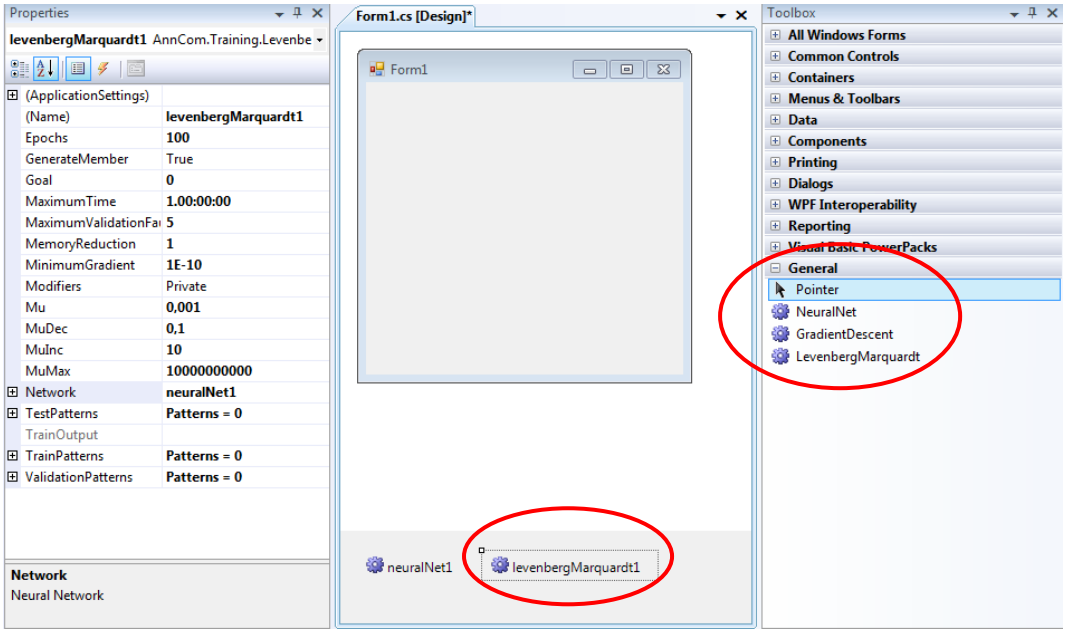


Figura 20 – Suporte da ANNCOM para tempo de execução no Visual Studio.

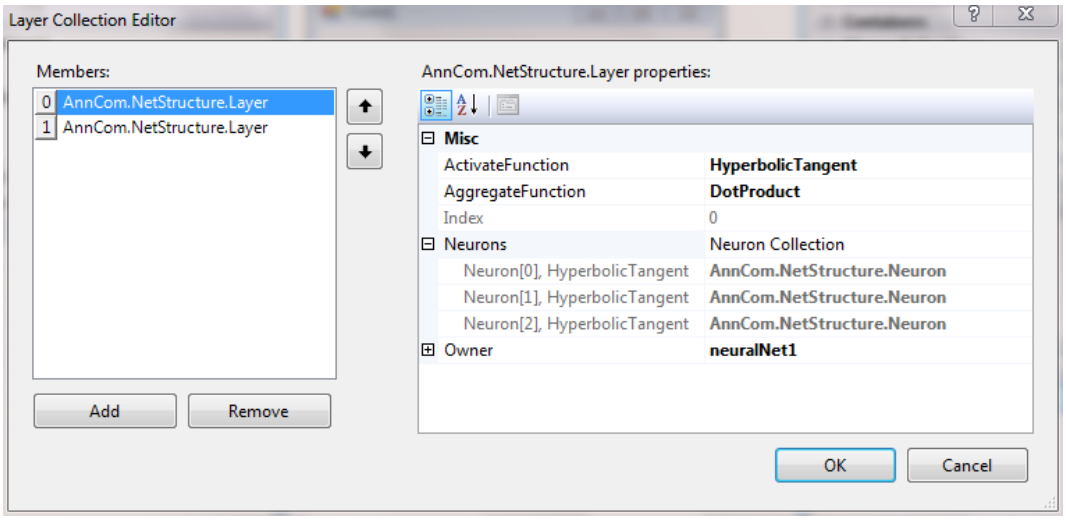


Figura 21 – Adição e edição de coleções na ANNCOM.

O requisito que trata do desempenho da biblioteca exigiu uma completa mudança de paradigma na implementação habitual de treinamento de uma rede neural. Neste trabalho, uma parte do treinamento foi modelada para ser executada em uma GPU com suporte a CUDA (Figura 22), levando-se em consideração a Lei de Amdahl (seção 3.1.1). Outras partes da biblioteca também incorporaram inovações para que o tempo de execução fosse reduzido. Essas inovações incluem o uso de *threads*, sistemas com vários processadores e vários computadores. A seguir, serão descritos em maiores detalhes os dois componentes principais da biblioteca: o *NeuralNet* e o *NetTrain*. Além disso, será apresentada uma ferramenta gráfica para auxiliar a criação de soluções envolvendo redes neurais.

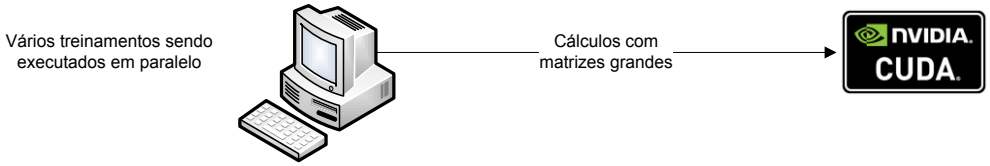


Figura 22 – Visão geral do desvio de parte do treinamento para GPU.

4.2. Estrutura Básica

Para facilitar a leitura e a organização do código, a biblioteca foi dividida em partes que foram definidas como Espaços de Nomes, conforme mostra a Figura 23. Além dos espaços principais, de Redes Neurais (*NetStructure*), de Calculadores de Erros (*NetError*) e de Treinamento (*Training*), que serão descritos nas

seções seguintes, existem o Espaço de Modelos (*Templates*), onde ficam os modelos de redes neurais implementados, o Espaço de Funções de Ativação (*ActivateFunctions*), cujo conteúdo são as funções de ativação das redes neurais e o Espaço de Saídas (*NetOutput*). Esse último também será descrito em uma próxima seção.

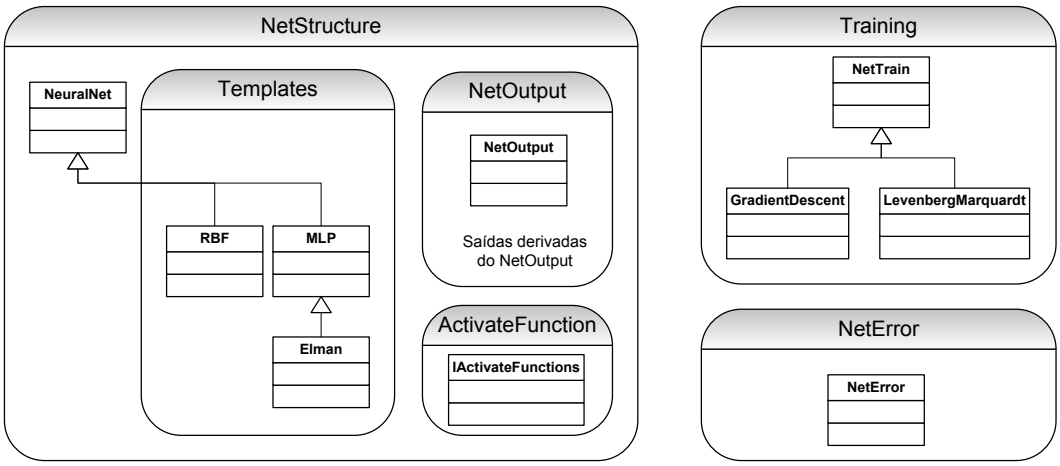


Figura 23 – Esquema com todos os espaços de nomes da biblioteca ANNCOM.

Além dessas partes, foram desenvolvidas outras secundárias que dão suporte às conexões com bancos de dados e cálculos de erro. Para trabalhar com bancos de dados e arquivos de texto ou XML, a ANNCOM possui uma estrutura chamada *PatternSet*, que armazena e manipula as entradas e alvos para treinamento, embaralha-os, transforma-os em matrizes ou divide-os em subgrupos para treinamentos com validação cruzada.

4.2.1. Componente NeuralNet

Com o objetivo de facilitar a explicação sobre o componente *NeuralNet*, a Figura 24 ilustra uma rede neural *Multi Layer Perceptron* (MLP). Nela, as camadas destacadas em azul e cinza possuem coleções de neurônios, os quais, possuem uma coleção de sinapses própria. O processamento dos padrões de entrada, oriundos, no caso da Figura 24, do banco de dados (BD), é feito propagando-os através da rede, que coloca o resultado no neurônio, ou nos neurônios, da última camada.

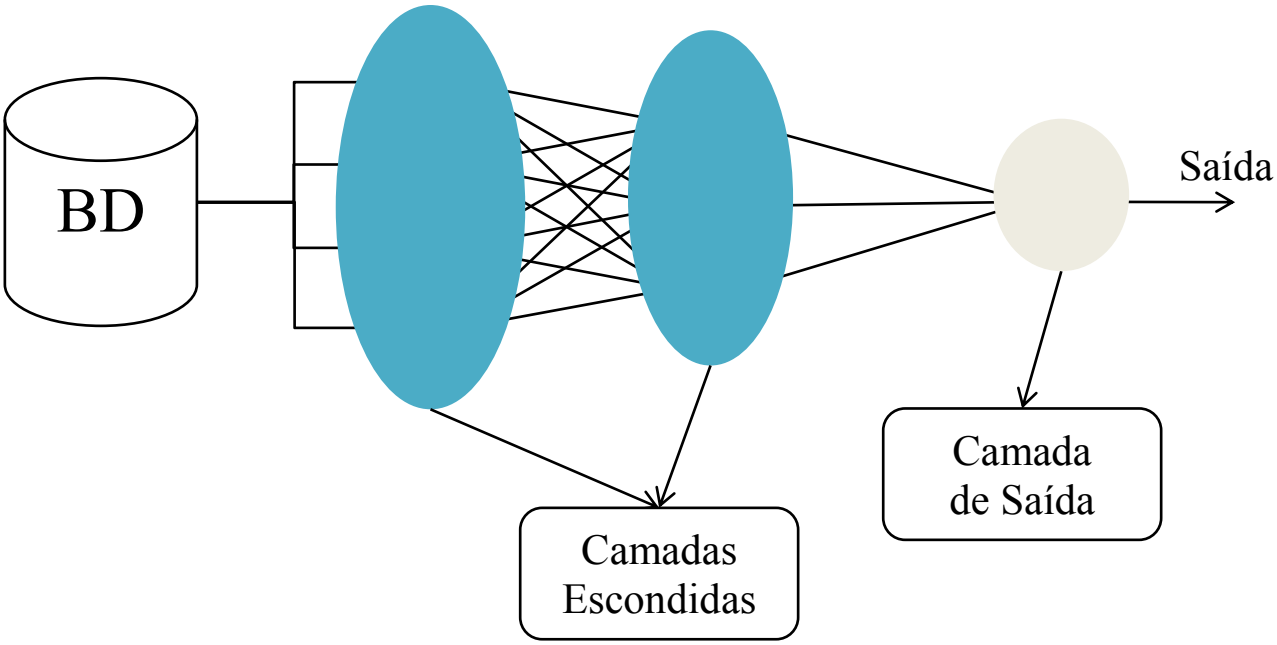


Figura 24 – Representação de uma rede neural MLP.

O componente *NeuralNet* representa uma rede neural artificial completa, incluindo todas as camadas com os neurônios e sendo o responsável pela propagação dos sinais de entrada pela rede. Fica também a cargo desse componente o dever de manter um estado consistente em toda a estrutura da rede neural. Neste trabalho, uma rede neural é considerada consistente quando todos os seus neurônios possuem um valor de saída referente ao último sinal propagado por ele. Como o *NeuralNet* é uma representação genérica de uma rede, todos os modelos desenvolvidos são uma especialização dele e, assim como o *NetError*, que será descrito mais adiante, podem-se criar novos modelos simplesmente herdando do componente pai, nesse caso, o *NeuralNet*.

A primeira inovação da ANNCOM é a estrutura do *NeuralNet* que é composta por vários objetos menores, conforme apresentado no padrão de projeto Composição (Figura 3). Nessa estrutura, a rede neural é formada por uma coleção de camadas que, por sua vez, é formada por uma coleção de neurônios, e estes possuem uma coleção de sinapses. Assim sendo, os componentes da biblioteca se apresentam de forma mais intuitiva, pois as partes do modelo teórico se transformaram em objetos ou componentes independentes, que podem ser especializados, modificados ou atualizados, sem afetar o resto da estrutura.

Além de uma estrutura mais intuitiva, o modelo estrutural da biblioteca permite uma gerência de memória melhor, mesmo custando mais espaço, pois as

informações que a rede possui ficam distribuídas em vários objetos que são gerenciados pelo componente *NeuralNet*. Normalmente, uma rede neural é tratada como uma cadeia de vetores ou várias matrizes, o que pode ser mais simples de implementar, mas que pode gerar blocos contínuos suficientemente grandes de memória a ponto de serem paginados para o disco rígido, implicando em um desempenho menor no treinamento.

4.2.2. Modelos de Redes Neurais Artificiais

Para facilitar o uso da biblioteca, alguns dos modelos mais utilizados de redes neurais foram implementados e são indicados na Figura 25. Esses objetos são especializações do componente mais genérico e possuem algumas restrições de funcionalidades ou comportamentos para que o modelo teórico da rede não seja desrespeitado. Por exemplo, no modelo *Elman*, Figura 26, onde $x(k)$ = entrada da rede, $a_n(k)$ = saída dos neurônios, $z^{-1}(k)$ = vetor de estados e $y(k)$ = saída da rede, os neurônios da camada escondida estão ligados aos neurônios da camada de entrada e, ao se criar uma rede a partir de uma instância do objeto *Elman* da biblioteca ANNCOM, essa ligação será feita de forma automática.

No caso das redes radiais (RBF), ao se criar uma nova rede, deve-se ajustar os parâmetros das funções de ativação (gaussiana) dos neurônios que são criados a partir dos padrões de treinamento. Esse processo também é executado automaticamente ao se criar um objeto RBF da ANNCOM.

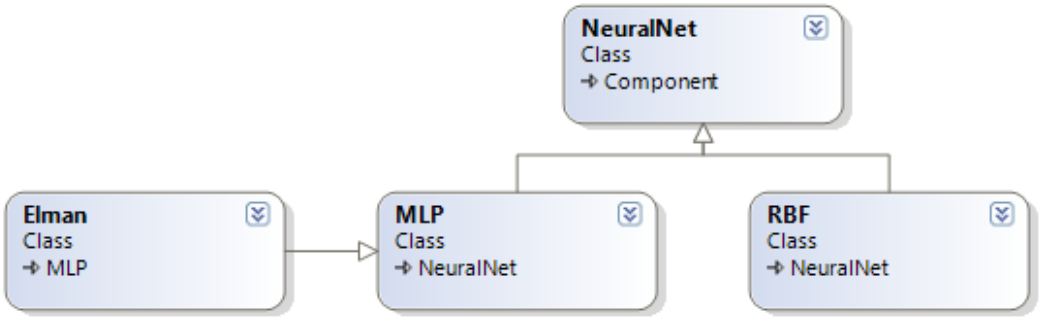


Figura 25 – Diagrama de classes simplificado dos modelos de redes neurais implementados nesse trabalho.

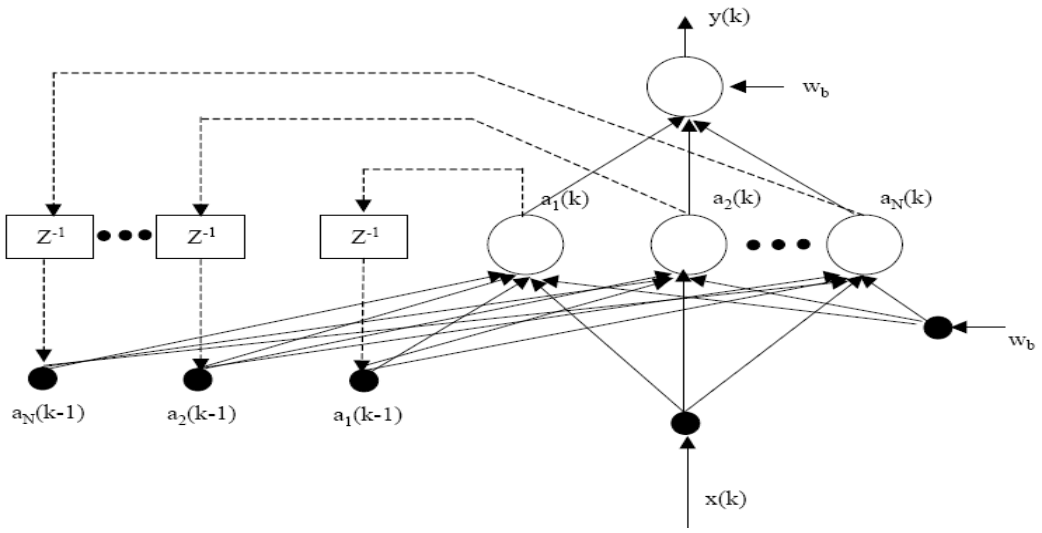


Figura 26 – Modelo de uma rede Elman.

As redes MLP são as mais comuns e sua estrutura já foi apresentada na Figura 24. Para esse tipo de rede, a biblioteca ANNCOM impede o uso de algumas funções de ativação e funções de agregação, e conecta automaticamente os neurônios, de modo que o sinal sempre siga na direção da saída, pois isso é uma característica de tipo de rede, também denominado MLP *Feed-forward Only*.

4.2.3. Estrutura NetOutput

Para decodificação dos resultados, a ANNCOM possui um objeto chamado *NetOutput*. Essa decodificação converte as saídas dos neurônios da última camada da rede para um formato compatível com o problema. Por exemplo, se uma rede neural está sendo usada para avaliar se um cliente pode ou não pegar um empréstimo bancário, o *NetOutput* deve transformar as saídas dos neurônios em "sim" ou "não". Desse objeto derivam vários outros mais específicos, como saídas Numérica, de Classificação e outras, seguindo o padrão de projeto Estratégia. Além disso, como a ANNCOM é orientada a objetos, podem-se criar novos decodificadores herdeiros do *NetOutput*. A Figura 27 apresenta um diagrama de classes resumido com a estrutura já implementada na biblioteca.

O tipo *Approximation* apenas desnormaliza os sinais, caso eles tenham sido normalizados antes de serem apresentados à rede. Esse tipo de saída é normalmente usado em previsões de séries e aproximações de funções. O tipo *Classification* é utilizado em problemas como classificação de clientes, apoio a decisões e etc.

Na ANNCOM, o tipo *Classification* foi derivado em mais três classes, com nomes auto-explicativos para especialistas na área de redes neurais: *One of N*, *Thermometer*, *Binary*. O tipo *Clustering* foi criado para agrupamento de subconjuntos. Esse tipo de saída se adéqua a problemas como mineração de dados ou problemas de classificação onde as classes ainda não são conhecidas.

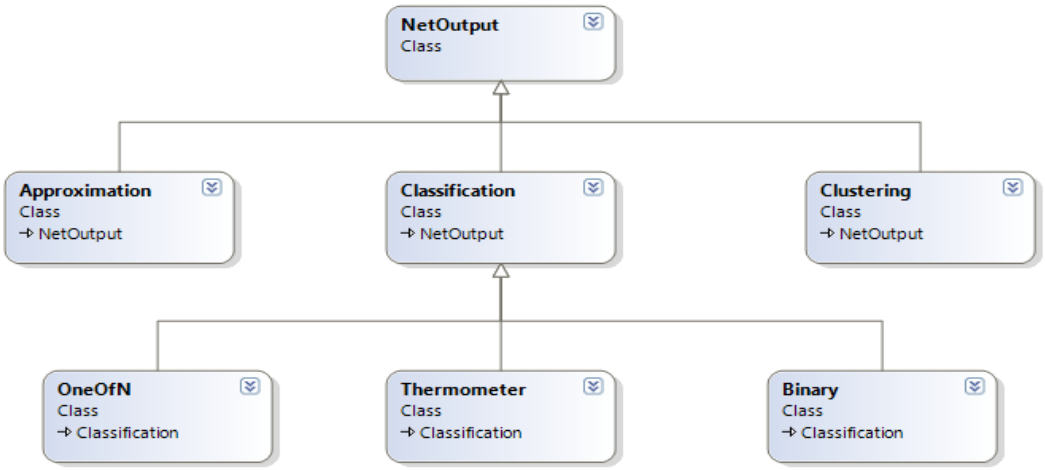


Figura 27 – Diagrama de classes com as estruturas de decodificação das saídas das redes neurais.

4.2.4. Estruturas para Cálculo de Erro

Para fornecer suporte ao cômputo de erros, a biblioteca possui uma estrutura (*NetError*) com diversos tipos de cálculos conhecidos, como Erro Médio Quadrático, Erro Absoluto Médio, U de *Theil*, entre outros. Além disso, é possível criar novos objetos para cálculo de erro, simplesmente herdando do objeto *NetError*. A Figura 28 apresenta um diagrama de classes simplificado, com os objetos para cálculo de erro implementados na ANNCOM.

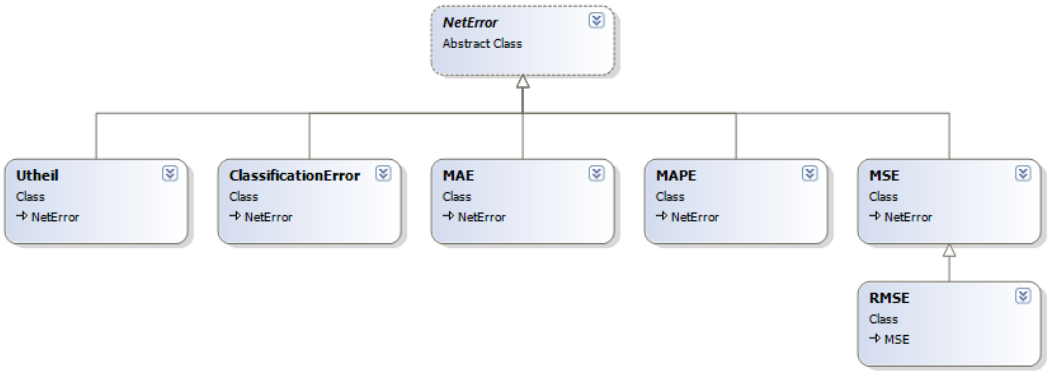


Figura 28 – Diagrama de classes com os objetos para cálculo de erro, implementados na ANNCOM.

A métrica U de Theil serve para determinar a acerácea de uma previsão (P_i) comparando o conjunto previsto com os valores reais (A_i), conforme a Equação 4. Caso a previsão seja perfeita, o resultado será zero, e se um modelo naïve, onde a previsão é igual à última realização da série, for utilizado, o resultado será um. Dessa forma, modelos melhores darão resultados entre zero e um.

$$U = \frac{\sqrt{\sum (P_i - A_i)^2}}{\sqrt{\sum {A_i}^2}} \qquad \text{Equação 4}$$

O objeto chamado *ClassificationError* fornece um resultado percentual dos erros de classificação a partir de uma matriz de confusão como a da Equação 5, onde C_1 é o acerto da rede no grupo um, C_0 é o acerto da rede no grupo zero, F_1 é o erro da rede no grupo um e F_0 é o erro da rede no grupo zero. A partir dessa matriz, pode-se calcular a sensibilidade (S) (Equação 6) e a especificidade (E) (Equação 7).

$$\begin{bmatrix} C_1 & C_0 \\ F_1 & F_0 \end{bmatrix} \qquad \text{Equação 5}$$

$$S = \frac{C_1}{C_0 + F_1} \qquad \text{Equação 6}$$

$$E = \frac{F_1}{C_0 + F_1} \qquad \text{Equação 7}$$

A métrica MAE (*Mean Absolute Error* – Erro Absoluto Médio) é como o nome sugere uma média dos erros absolutos da previsão e é dado pela Equação 8, onde *e* representa o erro da previsão. Essa métrica é usada em previsões de séries e aproximações de funções, assim como a métrica MAPE (*Mean Absolute Percentage Error* – Erro Médio Percentual Absoluto). A MAPE é dada pela Equação 9, onde *A* é a série real e *F* é a série prevista. Essa métrica apresenta alguns problemas práticos como, por exemplo, ocorrências de divisões por zero quando existem zeros na série real.

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |e_i| \qquad \text{Equação 8}$$

$$M = \frac{1}{n}\sum_{t=1}^{n}\left|\frac{A_t - F_t}{A_t}\right| \qquad \text{Equação 9}$$

As métricas MSE (*Mean Squared Error* – Erro Médio Quadrático) e sua derivação chamada RMSE (*Root Mean Squared Error* – Raiz do Erro Médio Quadrático) são definidas, respectivamente, na Equação 10 e na Equação 11. Como o MSE é o momento de segunda ordem do erro, incorpora tanto a variância do estimador quanto a sua parcialidade (BERGER e CASELLA, 2001). Para um estimador não viesado, isto é, quando a diferença entre o valor esperado e o valor estimado é zero, o MSE é a variância e ambos têm a mesma unidade de medida. Em uma analogia ao desvio-padrão, tomando a raiz quadrada do MSE produz-se o RMSE, que tem as mesmas unidades que a quantidade estimada por um estimador imparcial. O RMSE é a raiz quadrada da variância, conhecida como o erro padrão (BERGER e CASELLA, 2001).

$$MSE = \frac{1}{n}\sum_{i=1}^{n} {e_i}^2 \qquad \text{Equação 10}$$

$$RMSE = \sqrt{MSE} \qquad \text{Equação 11}$$

4.3. Modelo de Treinamento

Antes de apresentar o modelo de treinamento da biblioteca ANNCOM, serão descritos os dois principais algoritmos de treinamento de redes neurais, ambos envolvendo o cálculo do gradiente. O primeiro deles é chamado Gradiente Decrescente, e é um algoritmo de otimização de primeira ordem, que utiliza o negativo do gradiente para encontrar o mínimo local de uma função. Conforme mostra a Figura 29, cada passo do algoritmo tem a direção do vetor gradiente.

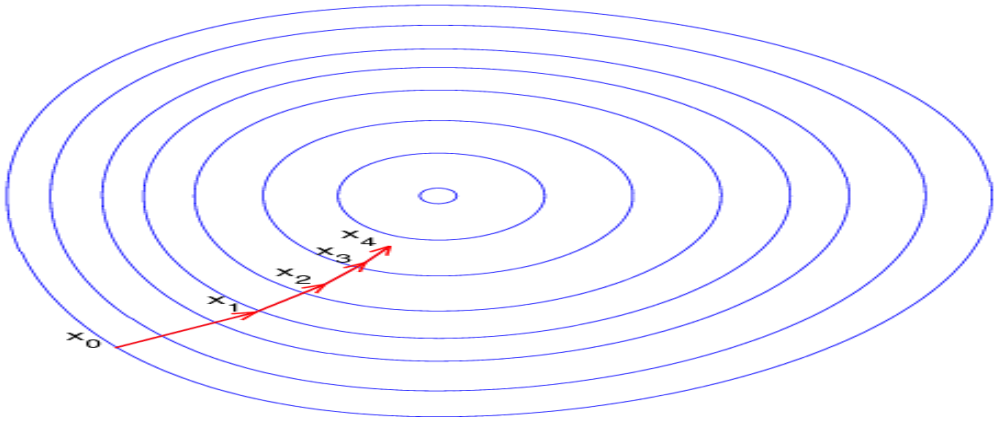


Figura 29 – Ilustração de uma descida por gradiente (WIKIPEDIA, 2010).

Para treinar as redes neurais, o objeto *GradientDescent* propaga, a cada iteração ou passo, todos os padrões de entrada da rede (modo *batch*), calculando o erro dos neurônios da última camada para cada padrão (Equação 12, onde t é o valor real, y é a saída da rede, p é o índice do padrão e i é o índice dos neurônios da última camada da rede).

$$E_i^p = t_i^p - y_i^p \qquad \text{Equação 12}$$

Com isso encontra-se o erro total (Equação 13) que será usado para encontrar o MSE da rede neural (Equação 14, onde n é o número de padrões e k é o número de neurônios na última camada da rede).

$$E_{SSE} = \sum_p \sum_i \frac{1}{2}\left(t_i - y_i\right)^2 . \qquad \text{Equação 13}$$

$$e_k = \frac{1}{n} E_{SSE} \qquad \text{Equação 14}$$

Esse erro é propagado para as camadas anteriores levando-se em conta os valores dos pesos (w) das sinapses (Equação 15, onde w_{ij} é o peso que liga o neurônio j ao neurônio i).

$$\frac{\partial e}{\partial w_{ij}} = \sum_p \frac{\partial e_p}{\partial w_{ij}} = \sum_k \frac{\partial e_p}{\partial a_k} \frac{\partial a_k}{\partial w_{ij}} \qquad \text{Equação 15}$$

A partir daí, mede-se a contribuição delta (δ_i) de cada padrão ao somatório a_i (Equação 16, onde a_i é a soma ponderada das entradas para o neurônio i).

$$\delta_i = \frac{\partial e_p}{\partial a_i} = \frac{\partial e_p}{\partial y_i} \frac{\partial y_i}{\partial a_i} \qquad \text{Equação 16}$$

O primeiro termo da Equação 16 é obtido da Equação 18 (válido apenas para SSE – *Sum Squared Error*) e o segundo termo da Equação 19, resultando na Equação 17.

$$\delta_{i_saída} = -(t_{pi} - y_{pi})f' \quad \text{Equação 17}$$

$$\frac{\partial e_p}{\partial y_i} = -(t_{pi} - y_{pi}) \quad \text{Equação 18}$$

$$\frac{\partial y_i}{\partial a_i} = f'(a_i) \quad \text{Equação 19}$$

Na Equação 19, a função *f'* é a derivada da função de ativação de cada neurônio. Considerando que os neurônios influenciam o erro pelo efeito deles sobre os neurônios das camadas subseqüentes, o cálculo do delta (Equação 20, onde *w* é o peso de uma sinapse) para os neurônios nas camadas escondidas é feito de forma indireta (REED e MARKS, 1999).

$$\delta_i = \frac{\partial e_p}{\partial a_i} = \sum_k \frac{\partial e_p}{\partial a_k}\frac{\partial a_k}{\partial a_i} = \sum_k \delta_k \frac{\partial a_k}{\partial a_i} = -f'_i \sum_k \delta_k w_{ki} \quad \text{Equação 20}$$

Após calcular os deltas de todos os neurônios, a atualização dos pesos é feita a partir da Equação 21, não utilizando momento (HAYKIN, 2007), ou da Equação 22, utilizando momento (HAYKIN, 2007). Nessas equações, o η representa a taxa de aprendizado (HAYKIN, 2007).

$$\Delta w_{ij} = -\eta \frac{\partial e_i}{\partial w_{ij}} \quad \text{Equação 21}$$

$$\Delta {w_{ij}}^t = -\eta \frac{\partial e_i}{\partial w_{ij}} + \alpha \Delta {w_{ij}}^{t-1} \quad \text{Equação 22}$$

O segundo algoritmo de treinamento usado na ANNCOM é baseado no algoritmo LMA – *Levenberg-Marquardt Algorithm* (MARQUARDT, 1963) para mínimos quadráticos não lineares, que foi incorporado à retro-propagação para treinamento de redes neurais em (HAGAN e MENHAJ, 1994). O método é uma aproximação do método de Newton, que é descrito na Equação 23 e tem o objetivo de minimizar o erro $E_{SSE} = E(\underline{x})$ da Equação 13, com relação ao vetor $\underline{x}$, que representa o vetor de pesos da rede neural.

$$\Delta x = -[\nabla^2 \mathrm{E}(\underline{\mathrm{x}})]^{-1} \nabla E(\underline{\mathrm{x}}) \quad \text{Equação 23}$$

Na Equação 23, $\nabla^2 \mathrm{E}(\underline{\mathrm{x}})$ é a chamada matriz Hessiana (Equação 24) e $\nabla E(\underline{\mathrm{x}})$ é o gradiente.

$$\mathrm{H}(\underline{\mathrm{x}}) = \nabla^2\mathrm{E}(\underline{\mathrm{x}}) = \begin{bmatrix} \frac{\partial^2 E(\underline{x})}{\partial x_1^2} & \cdots & \frac{\partial^2 E(\underline{x})}{\partial x_1 x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 E(\underline{x})}{\partial x_n x_1} & \cdots & \frac{\partial^2 E(\underline{x})}{\partial x_n^2} \end{bmatrix} \quad \text{Equação 24}$$

Entretanto, o custo computacional do método de Newton pode ser muito alto, por causa da complexidade de se calcular a matriz Hessiana e sua inversa, pelo fato que essa matriz pode ser singular. De forma a contornar esses dois problemas, o LMA, mesmo não possuindo uma convergência tão rápida quanto o método de Newton, possui iterações computacionalmente mais baratas e um ajuste para minimizar a ocorrência de matrizes singulares.

Para simplificar o método de Newton, o LMA assume que $E(\underline{x})$ é uma soma de quadrados na forma da Equação 25, o que possibilita utilizar o método de Gauss-Newton (MARQUARDT, 1963) e, com isso, reescrever as duas componentes da Equação 23 na forma da Equação 26, onde *J* é a matriz Jacobiana (Equação 27) e *S* é dado pela Equação 28.

$$E(x) = \sum_{i=1}^{N} e_i^2\,(\underline{x}) \quad \text{Equação 25}$$

$$\begin{gathered} \nabla E(\underline{\mathrm{x}}) = J^T(\underline{x})\underline{e}(\underline{x}) \\ \nabla^2\mathrm{E}(\underline{\mathrm{x}}) = J(\underline{x})J^T(\underline{x}) + S(\underline{x}) \end{gathered} \quad \text{Equação 26}$$

$$J(x) = \begin{bmatrix} \frac{\partial e_1(\underline{x})}{\partial x_1} & \cdots & \frac{\partial e_1(\underline{x})}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial e_N(\underline{x})}{\partial x_1} & \cdots & \frac{\partial e_N(\underline{x})}{\partial x_n} \end{bmatrix} \quad \text{Equação 27}$$

$$S(\underline{x}) = \sum_{i=1}^{N} e_i\,(\underline{x})\nabla^2 e_i(\underline{x}) \quad \text{Equação 28}$$

Como o método de Gauss-Newton assume que $S(\underline{x}) \approx 0$, a Equação 23 pode ser reescrita na forma da Equação 29.

$$\Delta x = -\left[J(\underline{x})J^T(\underline{x})\right]^{-1}J^T(\underline{x})\underline{e}(\underline{x}) \quad \text{Equação 29}$$

Para minimizar a ocorrência de matrizes singulares, o algoritmo LM modifica a Equação 29 obtendo a Equação 30 (MARQUARDT, 1963), onde *I* é a matriz identidade e μ é um parâmetro que é multiplicado, a cada iteração, por um fator β enquanto a função $E(\underline{x})$ é minimizada.

$$\Delta x = -\left[J(\underline{x})J^T(\underline{x}) + \mu I\right]^{-1}J^T(\underline{x})\underline{e}(\underline{x}) \quad \text{Equação 30}$$

Caso o valor da função de erro aumente, o parâmetro μ é dividido pelo fator β. É importante perceber que, quando μ é muito grande, tem-se uma maior convergência com um passo de $1/\mu$, e quando μ é muito pequeno, o algoritmo tende ao método de Gauss-Newton.

O ponto mais importante para incorporação desse algoritmo ao treinamento é o cálculo da matriz Jacobiana a partir dos pesos da rede, por meio de pequenas modificações no algoritmo do gradiente decrescente já apresentado. Lembrando que a função de erro é dada pela Equação 31 (onde $\underline{a}_p^n$ é saída da rede neural para o padrão p), no caso da retro-propagação padrão, a derivada é dada pela Equação 32, onde k é o índice dos neurônios da última camada da rede, que possui um total de K neurônios, e l é o índice da camada.

$$E = \frac{1}{2}\sum_{p=1}^{P}\left(\underline{t}_p - \underline{a}_p^n\right)^T\left(\underline{t}_p - \underline{a}_p^n\right) = \frac{1}{2}\sum_{p=1}^{P}\underline{e}_p^T\underline{e}_p \qquad \text{Equação 31}$$

$$\frac{\partial \hat{E}}{\partial w^l(i,j)} = \frac{\partial \sum_{k=1}^{K} e_{p(k)}^2}{\partial w^l(i,j)} \qquad \text{Equação 32}$$

Para os elementos da matriz Jacobiana, é necessário calcular a Equação 33. Lembrando que o cálculo da Equação 33 para a última camada no processo de retro-propagação do gradiente decrescente é dado pela Equação 17, esses elementos são calculados com a modificação dada pela Equação 34 na última camada.

$$\frac{\partial e_p(k)}{\partial w^l(i,j)} \qquad \text{Equação 33}$$

$$\Delta^k = -f^k \qquad \text{Equação 34}$$

Na ANNCOM, utiliza-se uma abordagem inovadora para se treinar redes neurais. Na maioria dos programas avaliados, as redes neurais apontam para o algoritmo de treinamento que será usado para treiná-la. Na biblioteca deste trabalho, concebeu-se um modelo de treinador e aluno. A principal diferença entre as duas abordagens se evidencia ao se modelar um sistema envolvendo muitas redes neurais: a nova abordagem torna a gerência dos treinamentos mais intuitiva, já que podem ser criadas "salas de aula" com redes neurais, como mostra a Figura 30. Nessa figura, as salas de aula representam treinadores com diferentes configurações, que são responsáveis por treinar todos os seus alunos, no caso, as redes neurais. Por exemplo, considere que a sala A representa um treinamento usando o LMA e as redes 1, 2 e 3 são redes MLP com diferentes números de camadas e neurônios. Nesse caso, o treinador A deve treinar, validar e testar as redes 1, 2 e 3

usando suas próprias configurações. O treinador B, por sua vez, treina, valida e testa as redes 4 e 5, usando outras configurações e até outro algoritmo de treinamento.

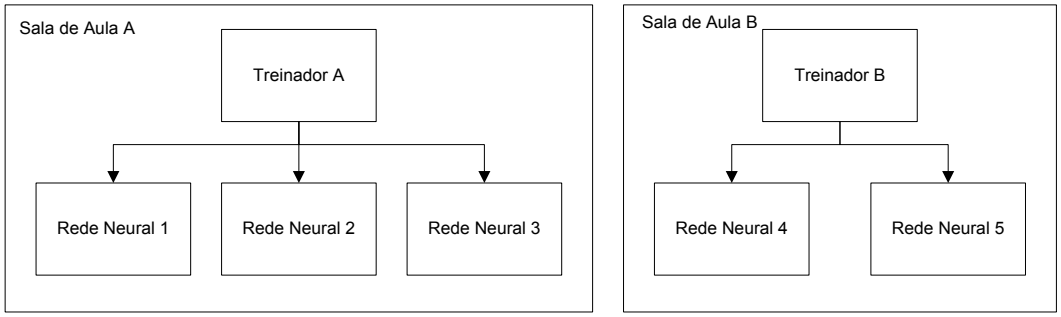


Figura 30 – Exemplo de duas "salas de aula" onde várias redes são treinadas.

Além desse novo modelo para treinamento, foram adicionadas algumas funções para aumentar a produtividade dos programadores que fizerem uso da biblioteca, como por exemplo, o treinamento com validação cruzada automática, que faz várias divisões na base de entrada, embaralhando os padrões e selecionando a rede que melhor atende aos critérios de avaliação do sistema em questão – **Error! Reference source not found.**. Nessa figura, os dados da base de entrada são lidos pelo treinador, que os embaralha de modo que, ao dividir a base misturada, gere novos grupos de treinamento, validação e teste. Depois de embaralhar e dividir a base de entrada, o treinador inicia o treinamento nas suas redes alunas. Esse processo pode ser repetido diversas vezes e serve para maximizar as chances de sucesso no treinamento (HAYKIN, 2007).

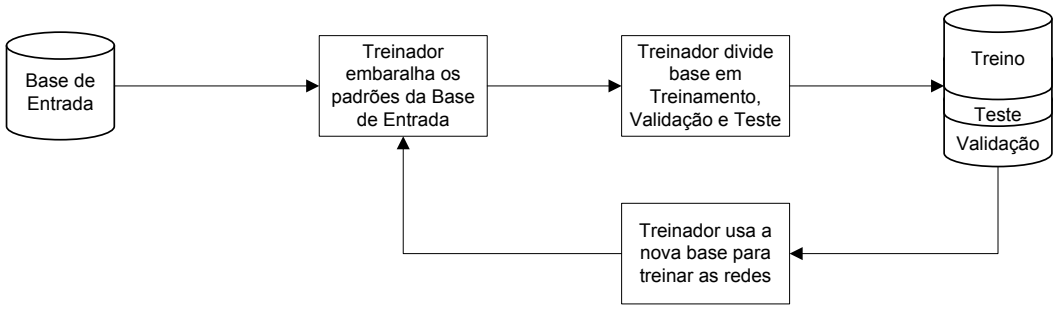


Figura 31 – Validação cruzada automática da ANNCOM.

Para avaliar os resultados com diferentes critérios de erro, o *NetTrain* possui um gerenciador para cálculos de erro, que é responsável por calcular os erros refe-

rentes aos padrões da base de validação ou teste – Figura 32. Nessa figura, o retângulo cinza é o início do processo com os padrões sendo propagados através da rede neural. Após a propagação, o treinador invoca o gerenciador de erro para executar os cálculos necessários. Por sua vez, o gerenciador de erro, executa os cálculos, salva os resultados e devolve o controle ao treinador. Caso não tenha que executar nenhum cálculo, informa ao treinador para continuar. Por exemplo, se um sistema treina uma rede neural utilizando, para teste, as métricas Erro Percentual Absoluto Quadrático e U de Theil, a saída desse treinamento terá dois vetores com os resultados dos cálculos de erro de teste de cada época, além das informações padrões, como tempo decorrido, erro do treinamento e status final.

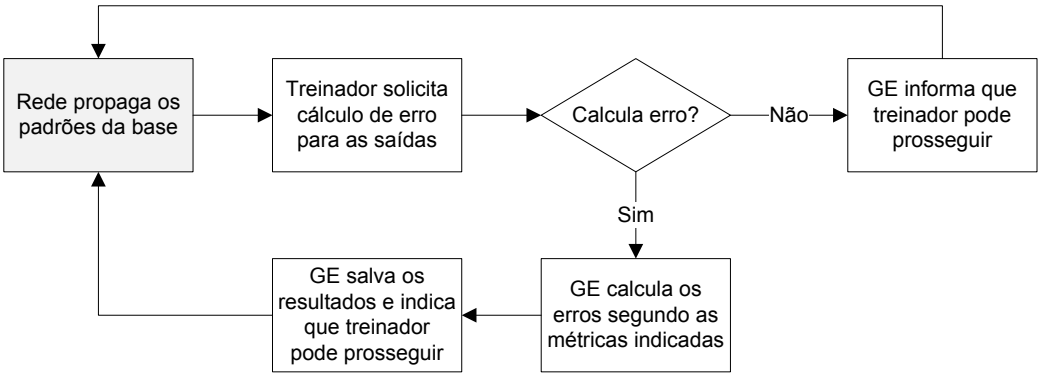


Figura 32 – Exemplo de funcionamento do Gerenciador de Erro (GE) para cálculo de erros de validação e teste durante o treinamento.

Além das funcionalidades já apresentadas, o *NetTrain* oferece dois tipos de inicialização de pesos para as redes neurais: uma aleatória e baseada em (REED e MARKS, 1999) e outra baseada em (NGUYEN e WIDROW, 1990) usada no Matlab. Com o objetivo de melhorar o desempenho da biblioteca, o treinamento pode ser feito utilizando vários processadores, através do recurso de *threads*. A parte do código paralelizada é responsável pelo cálculo da matriz Jacobiana quadrada e da variação dos pesos a partir dessa matriz, conforme Equação 30 (HAGAN e MENHAJ, 1994). Tal cálculo representa mais de 70% do tempo de processamento para uma base de tamanho pequeno e pode chegar a quase 90% em bases maiores.

4.3.1.
O Treinamento em GPGPU

Considerando que a Equação 30 é executada uma vez a cada época do treinamento e que nela a matriz *J* possui tamanho (w, p), onde *w* é o número de pesos da rede e *p* é o número de padrões, mesmo utilizando vários núcleos, o tempo total de um treinamento pode não ser satisfatório, uma vez que a matriz, pode conter milhões de elementos. Na verdade, esse tempo pode chegar a 90% do tempo total, conforme apresentado anteriormente.

A utilização de placas gráficas se mostrou promissora, pois o número de núcleos de processamento em uma única placa pode chegar a centenas. Além disso, o tempo para se gerenciar *threads* na parte C# da ANNCOM pode custar mais do que 10% do tempo total de treinamento, enquanto que, conforme indicado no capítulo 3, a arquitetura CUDA pode tratar *threads* sem custo adicional. Outro ponto positivo, percebido ao se utilizar a tecnologia da NVIDIA, foi a possibilidade de se trabalhar com várias placas gráficas, permitindo a execução de vários treinamentos simultâneos.

A Figura 33 apresenta uma visão geral do treinamento e o fluxo do programa através do *host* e do *device*. Nessa figura, a Equação 30 é enviada para a GPU, de forma que o vetor com as variações dos pesos da rede fosse obtido diretamente na placa gráfica e a partir da matriz Jacobiana, dessa forma, a placa de vídeo é responsável por calcular toda a Equação 30, transferindo para a memória RAM da CPU apenas o resultado de Δx..

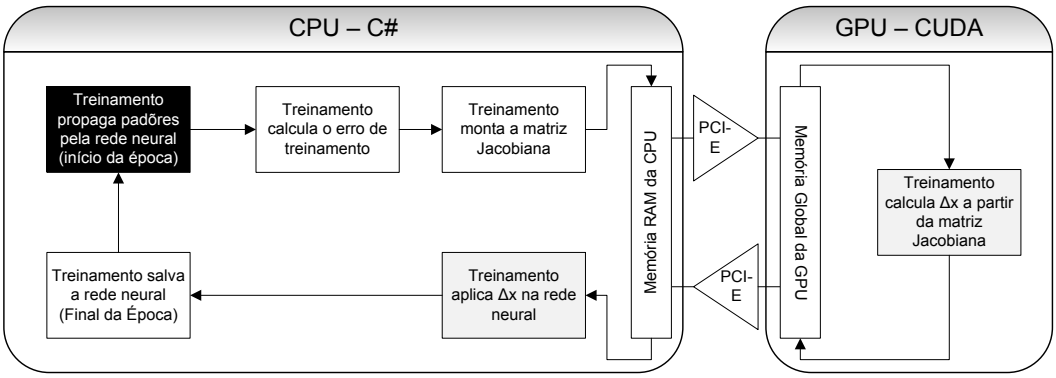


Figura 33 – Segundo modelo (resumido) proposto, que executa uma porção maior do código na placa gráfica.

4.3.2.
Implementação na GPU

A implementação da Equação 30 na GPU foi dividida em duas partes (*kernels*) principais: a primeira parte é responsável pela multiplicação matricial das matrizes e a segunda responsável por calcular a inversão matricial de um dos resultados da primeira parte. A Figura 34 mostra como foi dividido o processamento em *kernels*.

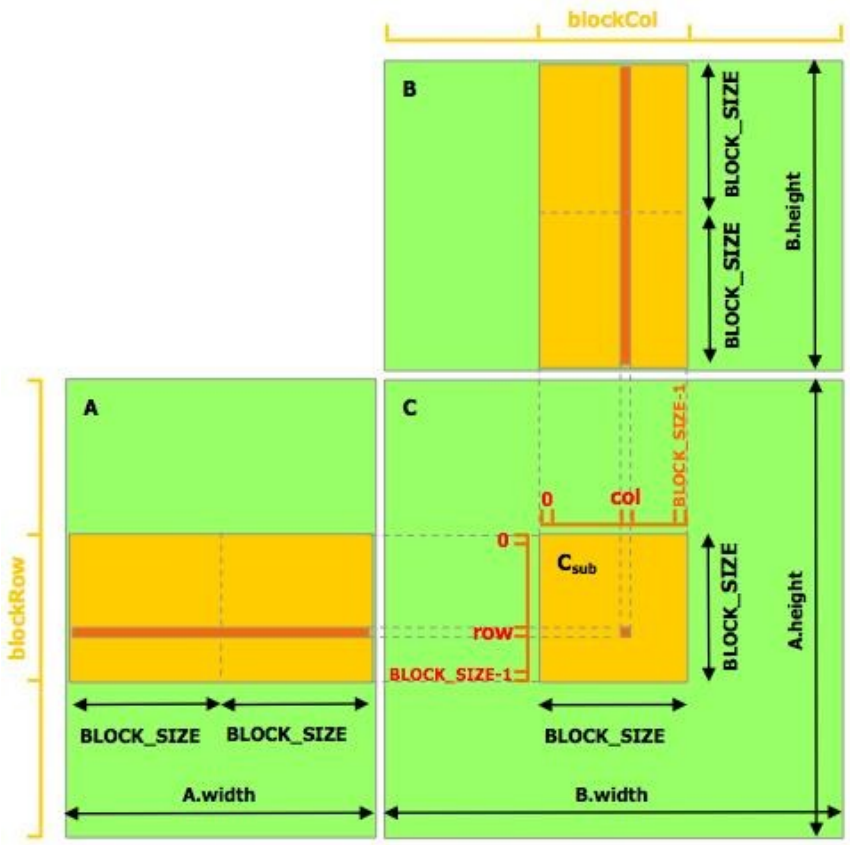


Figura 34 - Divisão do processamento em kernels.

4.3.3.
Segunda Parte: Inversão Matricial

O cálculo da matriz inversa, apesar de não representar a parte mais custosa da Equação 30, é a parte mais complexa do desenvolvimento. Para aproveitar velocidade da memória compartilhada da placa gráfica, cujos acesos pode chegar a ser 300 vezes mais rápido do que o acesso à memória global, o primeiro passo foi dividir a matriz em blocos conforme a Figura 35, da mesma forma como foi feito na multiplicação matricial. A vantagem do tratamento baseado em blocos é

dividir a matriz em submatrizes e transferi-las para memória compartilhada. Dessa forma, os acessos à memória global são limitados a uma única leitura e escrita, melhorando o desempenho. A partir daí, aplica-se a eliminação gaussiana no bloco, onde os pivôs são calculados.

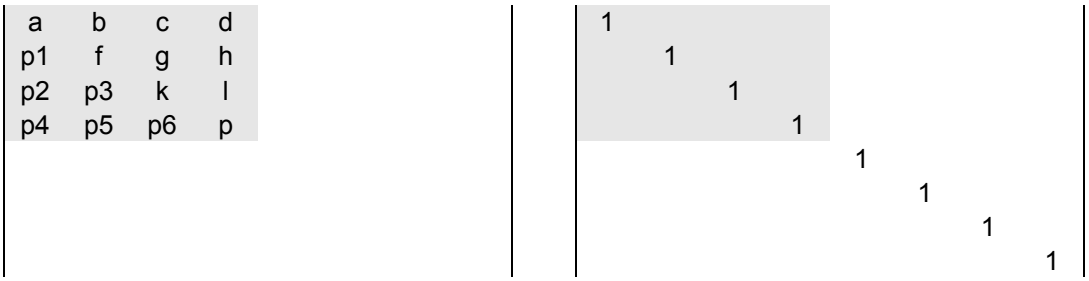


Figura 35 – Divisão da matriz em blocos de tamanho *n* (nesse caso, *n* é 4).

A partir dos pivôs calculados, atualizam-se as n linhas adjacentes das matrizes, conforme a Figura 36. Essa operação é feita nos diversos blocos que, nesse exemplo, estão marcados com cores diferentes. Esta operação é executada linha a linha de forma independente, mas os pivôs são modificados antes da atualização da próxima coluna, o que exige uma sincronização entre os blocos.

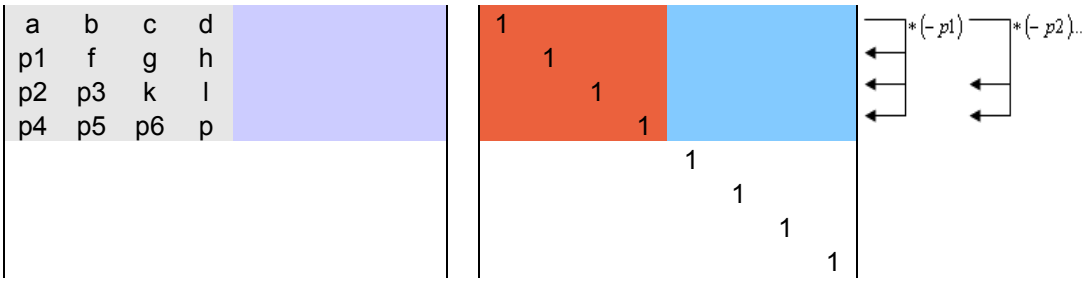


Figura 36 – Atualização das linhas adjacentes.

Os elementos abaixo do pivô do bloco da etapa 1 são necessário para produzir o bloco seguinte. Depois disso, as colunas são calculadas novamente a partir dos pivôs gerados nos passos anteriores, e, em seguida, coluna por coluna do bloco, propaga o resultado para as colunas do bloco 0 salvando os pivôs gerados em cada linha. A Figura 37 mostra como fica o bloco depois dos passos já apresentados.

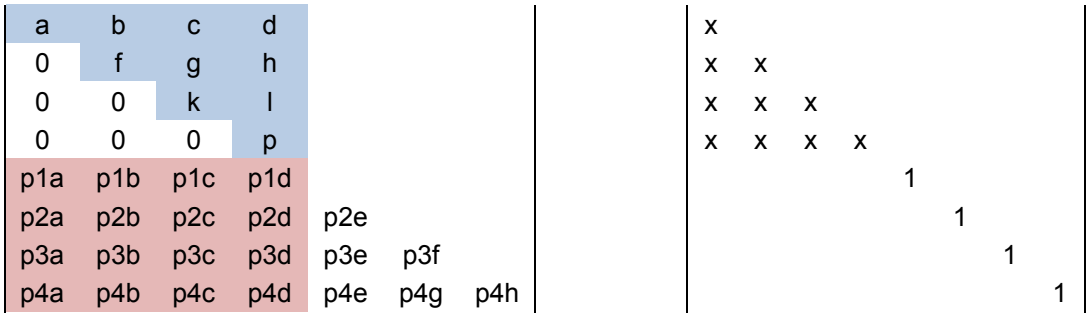


Figura 37 – Blocos atualizados.

A partir dos pivôs calculados na etapa 3, são calculadas as linhas remanescentes da matriz e da unidade matriz original novamente. Esta é a parte mais custosa do algoritmo e que requer para os cálculos os seguintes elementos:

- Pivôs do respectivo bloco (bloco azul escuro na Figura 38);
- Linhas de fatores para a multiplicação dos pivôs (destacado em amarelo na Figura 38);
- O bloco a ser ajustado (de cinza, vermelha, azul claro na Figura 38)

Os cálculos dentro de um bloco são independentes uns dos outros, mas as operações nas coluna devem ser sincronizadas.

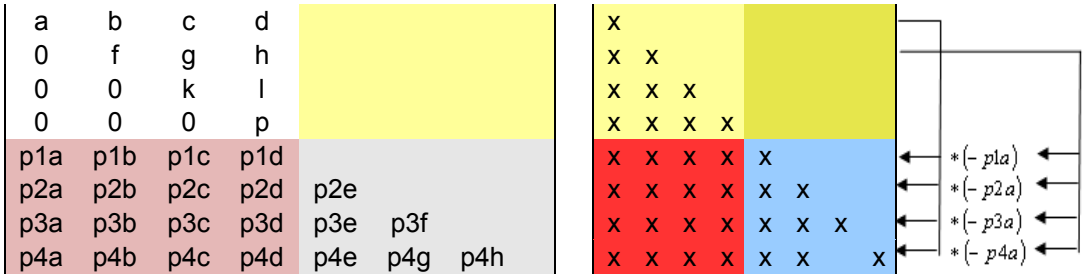


Figura 38 – Linha de fatores para a multiplicação dos pivôs.

Finalmente, todos os elementos abaixo da diagonal principal são iguais a zero. A partir desse ponto, basta repetir os passo 1 ao 4 usando o sub-bloco do passo 1 ao longo da diagonal. Para o restante do processamento, a transformação do resultado a ser observado que a matriz triangular inferior não é preenchido com 0, mas com valores numéricos.

4.4. Ferramenta Gráfica para Criação de Soluções Utilizando Redes Neurais – Clinn

Nesta seção é apresentada a ferramenta denominada Clinn, que significa *Client of Neural Network*, que utilizou, como biblioteca de redes neurais, a ANN-COM, já apresentada anteriormente neste capítulo. O Clinn, cuja tela inicial é apresentada pela Figura 39, também foi desenvolvidos sobre a plataforma .NET da Microsoft, pelos motivos já citados anteriormente.

O Clinn oferece diversos modos de trabalho e visualização para projetos envolvendo uma ou várias redes neurais. Além disso, existem vários modelos "passo-a-passo" que auxiliam na construção das soluções ou projetos e em estratégias para criar e treinar redes genéricas ou baseadas em algum modelo (ex: MLP, Elman).

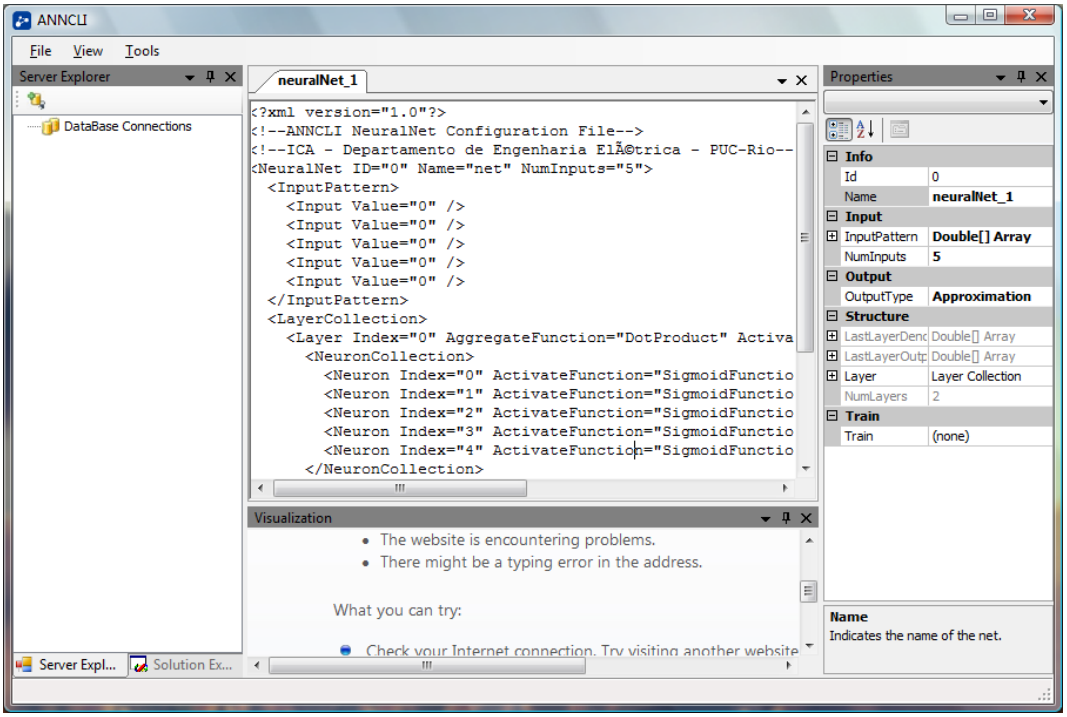


Figura 39 – Tela inicial do Clinn.

4.4.1. Interface utilizando Docas Flutuantes

As interfaces gráficas que utilizam docas flutuantes são muito comuns nos programas atuais, pois apresentaram diversas informações na tela de forma organizada. Na Figura 39 é possível verificar diversas funcionalidades disponíveis em

forma de docas flutuantes. Essas “janelas internas” possuem diversas funcionalidades que facilitam o trabalho do cliente. A Figura 40 mostra a possibilidade de se criar um ambiente completamente personalizado e que pode ser salvo, tornando mais fácil a adaptação ao programa.

Além das docas de funcionalidades (propriedades, explorador de solução e explorador de banco de dados), existe a doca de documentos, onde são editadas as redes neurais por meio de arquivos em formato XML. No futuro, o editor de texto será melhorado de forma a interpretar em tempo real o texto digitado, além de colori-lo, a fim de auxiliar na edição dos arquivos XML. Entretanto, já existe suporte para validação, que é feita cada vez que os documentos são salvos. Nos parágrafos seguintes, cada uma das docas de funcionalidades disponíveis no Clinn será descrita com maiores detalhes.

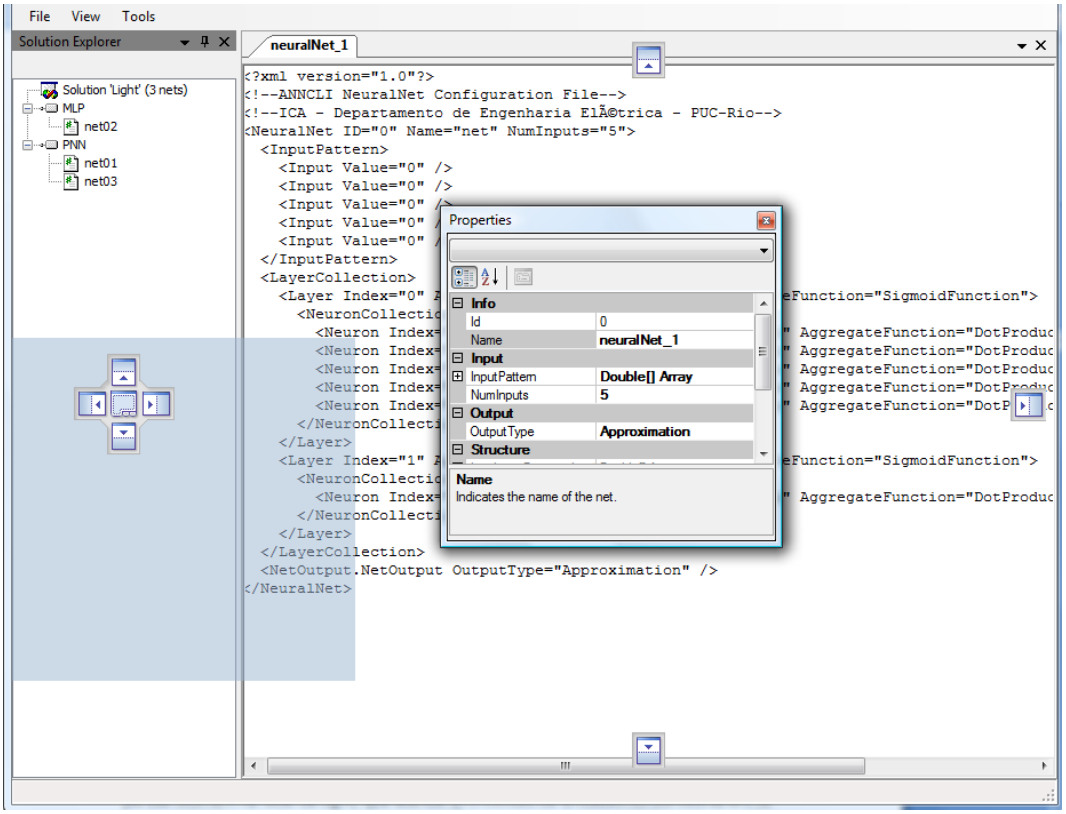


Figura 40 – Movimentação das docas pelo programa.

A doca Propriedades auxilia na edição das redes de uma forma simples e rápida. A Figura 41 mostra maiores detalhes da doca e como é simples ler qualquer informação contida no componente. Por essa doca é possível inserir um padrão

manualmente na camada de entrada da rede neural e propagá-lo através dessa rede, através da propriedade *InputPattern*.

As Figura 42, Figura 43 e Figura 44 mostram a possibilidade de editar as estruturas internas da rede como, respectivamente: camadas, neurônios e sinapses. Na Figura 42, são mostradas as camadas contidas nesta coleção (à direita). Nela, as camadas podem ser removidas e adicionadas, além de poderem ter sua ordem alterada. Neste modo é possível alterar qualquer propriedade das camadas individualmente. Na Figura 43, todas as propriedades dos neurônios são mostradas. Nessa figura, é possível observar que algumas propriedades são somente-leitura para preservar a consistência da rede neural. Na Figura 44, destaca-se o vetor de pesos, que permite visualizar ou editar os mesmos. Nesta tela, todas as propriedades das sinapses podem visualizadas, e algumas, alteradas pelo mesmo motivo indicado anteriormente.

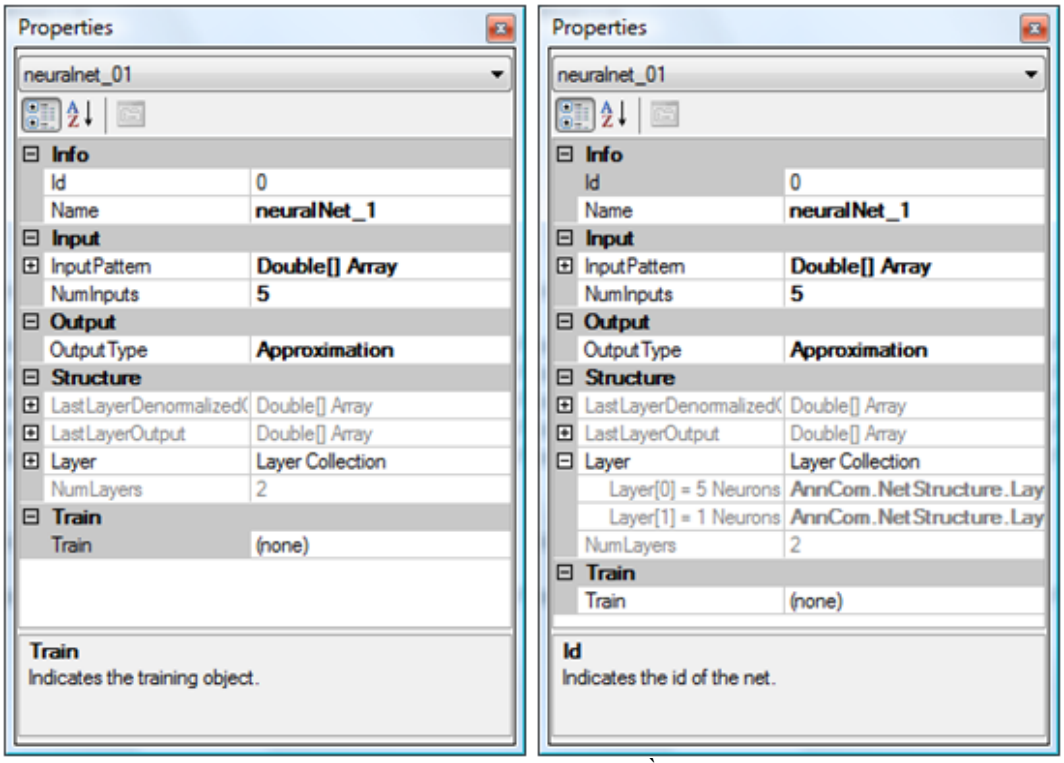


Figura 41 – A doca de propriedades em detalhes. À direita, a figura mostra a facilidade de navegação pelo componente.

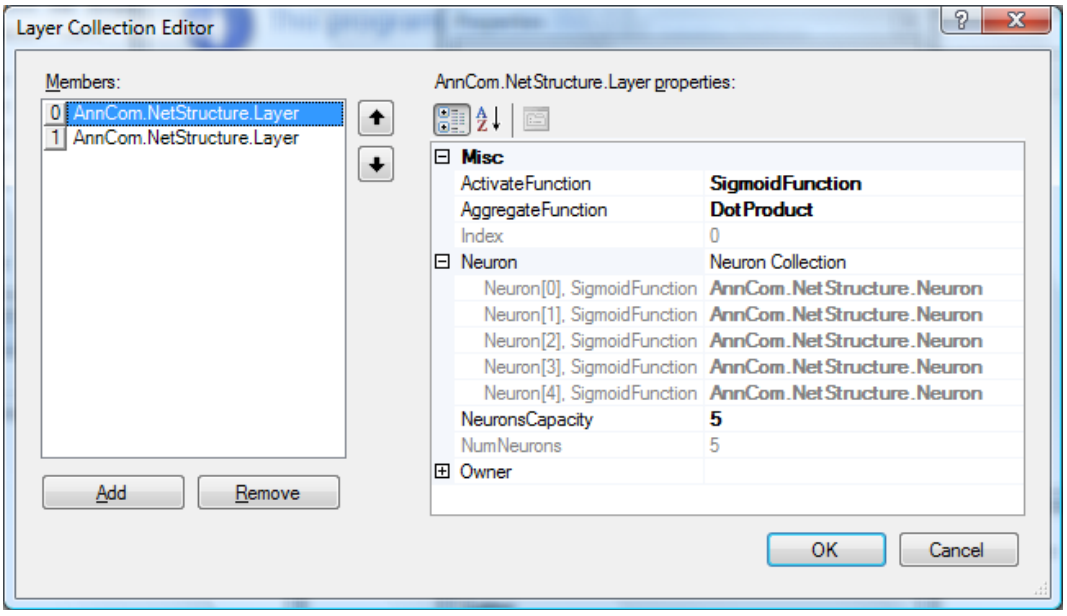


Figura 42 – Editor de coleção de camadas.

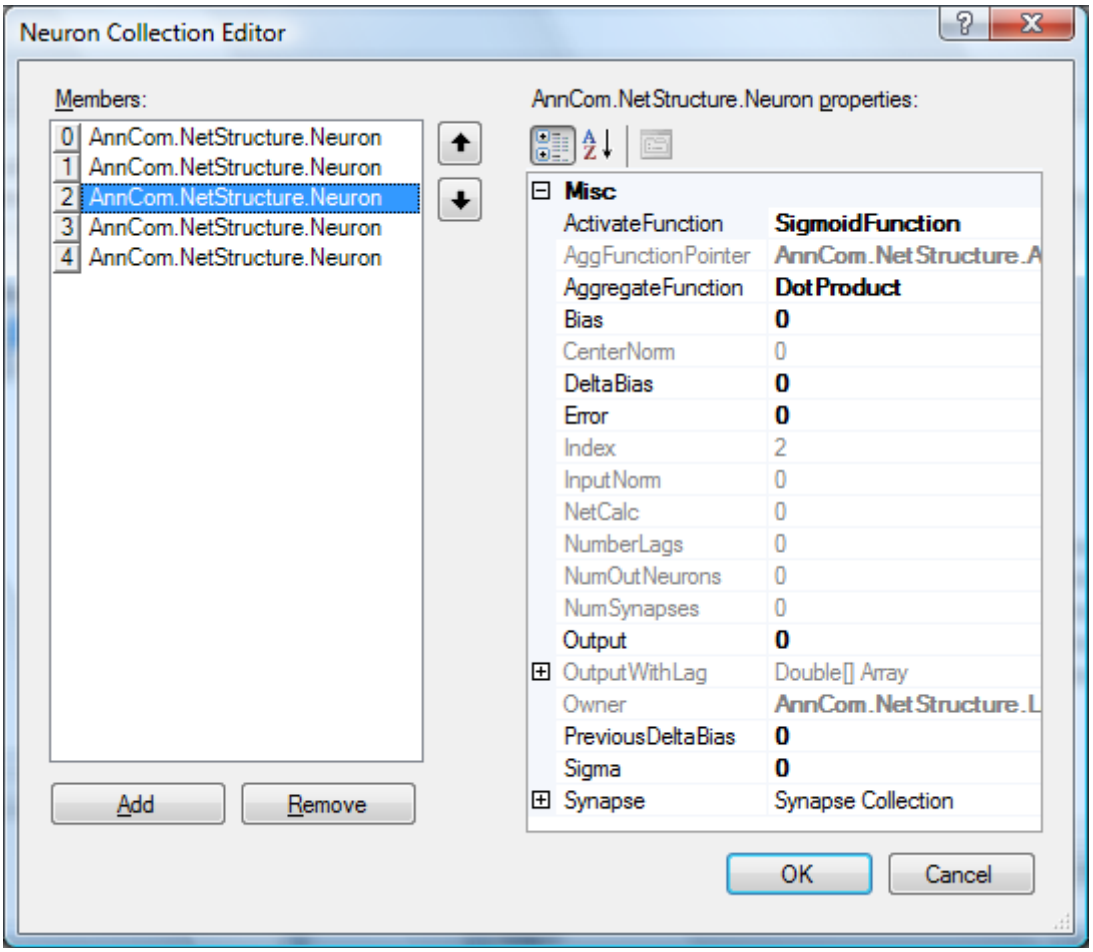


Figura 43 – Editor de coleção de neurônios.

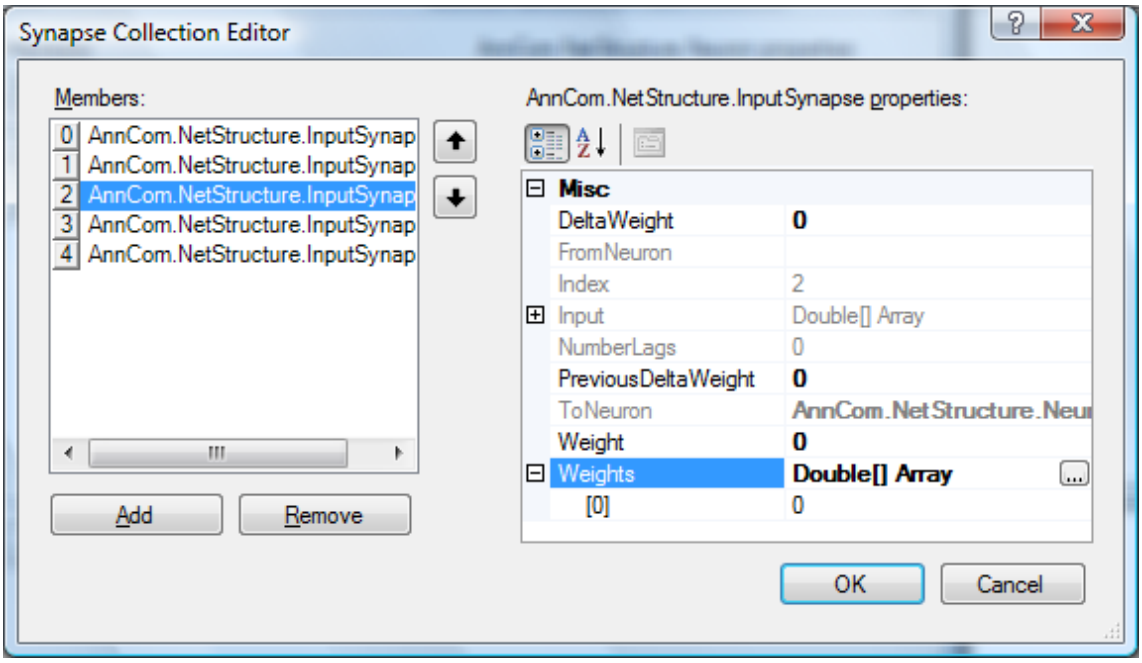


Figura 44 – Editor de coleção de sinapses.

A doca “Explorador de Solução” organiza as redes já criadas na solução corrente, uma vez que, para se encontrar uma boa rede para um problema, é necessário se testar várias redes com diferentes arquiteturas. Por exemplo, se uma estratégia de treinamento é configurada para que se crie redes MLP variando a quantidade de neurônios em uma determinada camada, podem ser geradas muitas redes. Com o objetivo de organizar essas redes, o Explorador de Solução as agrupa como mostra a Figura 45. Nessa figura, pode-se ver uma solução já criada com algumas redes neurais. Para editar qualquer rede contida na árvore, basta clicar duas vezes sobre a mesma e esta será aberta pelo editor de documentos que será apresentado em breve. Além disso, esta doca permite apagar os arquivos, adicionar novos e alterar seus nomes.

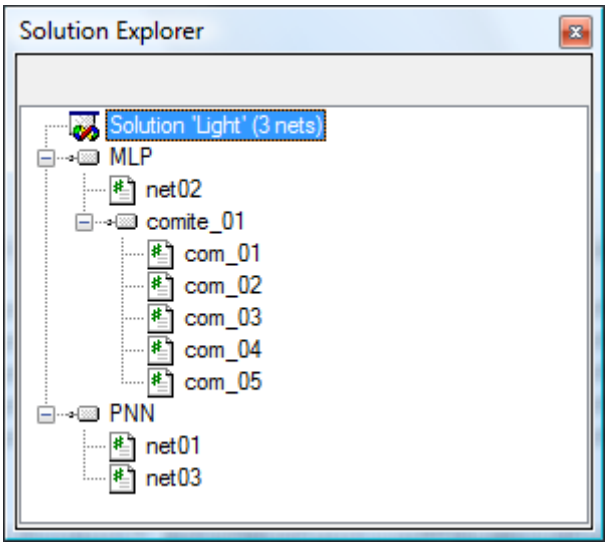


Figura 45 – Doca exploradora de solução.

A doca "Explorador de Bases de Dados" (Figura 46) permite a conexão com bases simultâneas e a visualização de tabelas ou visualizações dessas tabelas de forma simples e rápida. Na Figura 46, o explorador está conectado a uma base de dados SQL e mostra o conteúdo da mesma. Essa doca foi criada, pois a manipulação dos dados é uma parte fundamental na solução de um problema utilizando redes neurais. A rápida visualização e manipulação desses dados podem economizar algumas horas no final de um projeto. A Figura 47 mostra a visualização e a edição de tabelas ou visualizações dessas tabelas.

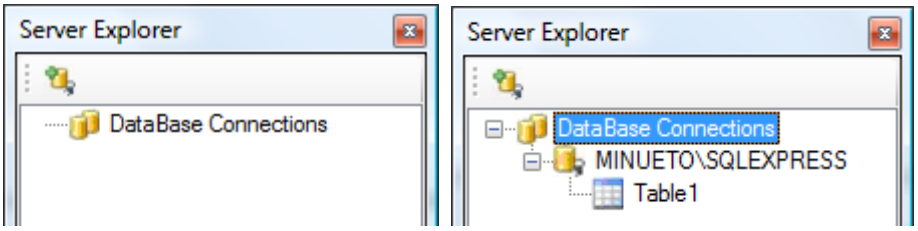


Figura 46 – Explorador de bases de dados.

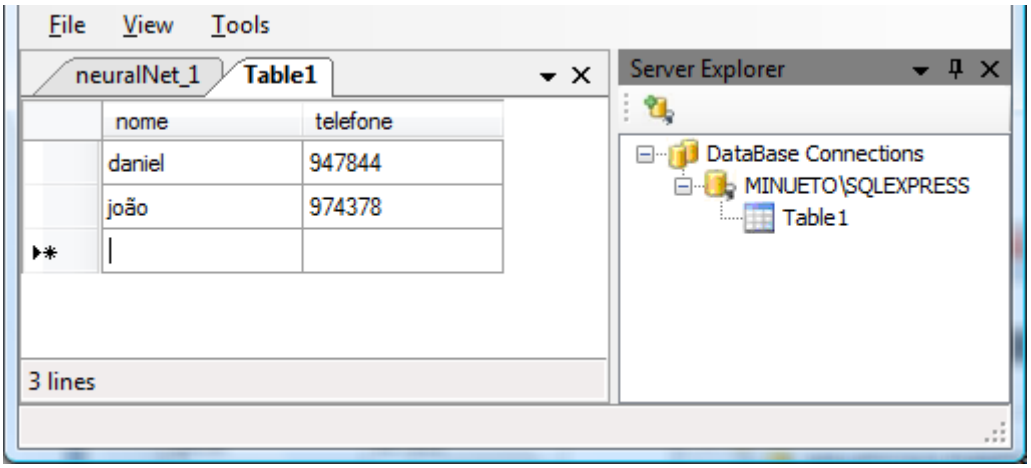


Figura 47 – Possibilidade de visualizar e editar as informações das tabelas.

A doca de treinamento oferece algumas funções que automatizam o processo de criação de uma rede. Nela é possível realizar operações para adicionar treinamento, interconectar a rede, desconectar a rede e inicializar os pesos. A operação de adicionar treinamento cria um componente de treinamento e o liga à rede neural. Interconectar a rede significa interligar todos os neurônios da rede através de sinapses. Os neurônios da camada n são ligados aos da camada n-1 até a primeira camada escondida, pois esta é ligada ao vetor de entrada através da *InputSynapses*. A operação de desconectar toda a rede remove todas as conexões da rede neural em questão. Quando esse comando é acionado, uma janela de confirmação é exibida, pois todo um treinamento pode ser perdido com este comando. A inicialização dos pesos da rede neural pode ser feita de três formas: (1) o usuário escolhe um valor fixo que será copiado para todos os pesos. Isto pode ser útil quando se quer experimentar diferentes tipos de treinamento; (2) o usuário fornece uma semente inicial para geração de pesos pseudo-randômicos. Este método gera pesos próximos de zero para evitar o problema de paralisia da rede; e (3) o usuário deixa o programa gerar pesos pseudo-aleatórios por conta própria. Isto pode ser útil quando se quer experimentar diversas vezes um determinado treinamento.

Futuramente, esta doca suportará mais opções além de uma integração maior com a doca exploradora de base de dados a fim de tornar mais dinâmico o processo.

As docas apresentadas acima são utilizadas para manipular a solução e, de forma gráfica, as propriedades dos componentes utilizados na solução. Para se editar as tabelas, visualizações e redes neurais, a doca de documentos. Ela tem um comportamento de lista, onde as abas representam os documentos abertos que

podem ser fechados ou reordenados. Isto é bastante útil quando se trabalha com muitos documentos abertos ao mesmo tempo. A Figura 48 mostra alguns documentos abertos nessa doca. É importante salientar que os arquivos de configuração de rede neural são validados cada vez que os mesmos são salvos, com o objetivo de mantê-los sempre válidos. Outra característica interessante é o back-up automático dos arquivos, que salva de tempos em tempos os arquivos para protegê-los de falhas no sistema. Mesmo se um arquivo nunca foi salvo, uma cópia temporária deste fica salva no disco rígido. Além das funcionalidades apresentadas, a doca de documentos fornece suporte para o passo a passo que será detalhado a seguir.

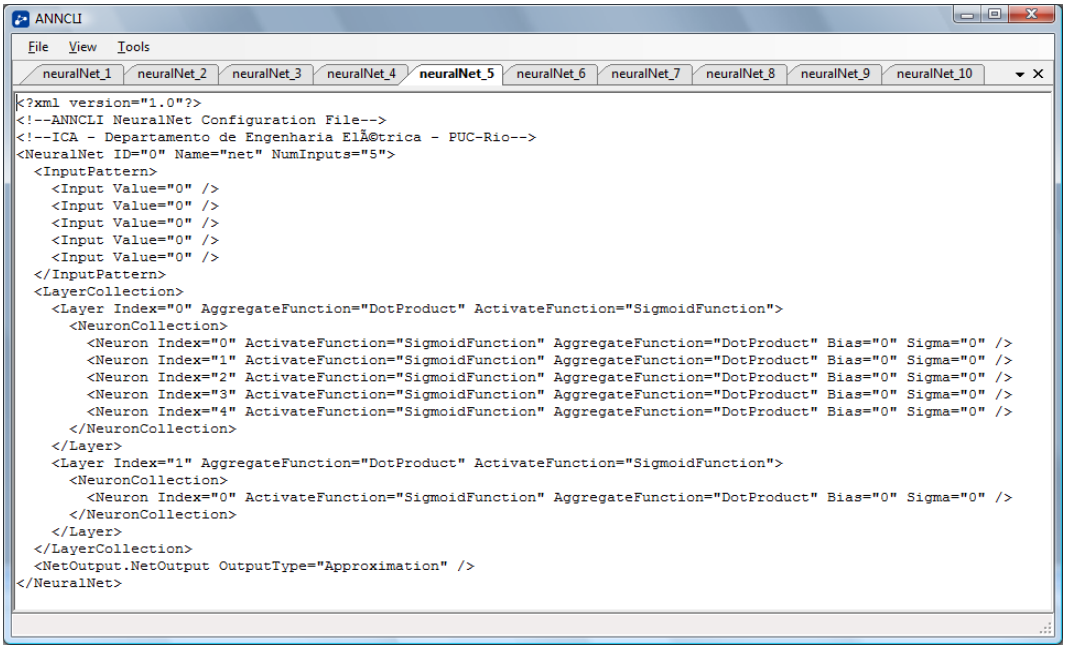


Figura 48 – Doca de documentos com uma lista de arquivos abertos.

4.4.2. Processo de Criação Automatizada

Uma maneira mais fácil de criar novas redes é usando o modelo passo a passo que pode ser visto na Figura 49. Neste modo, as configurações das redes são selecionadas de maneira intuitiva e fácil. Além disso, todos os detalhes da rede vão sendo aprofundados a cada passo. Na primeira parte, características como nome, tipo de estrutura, número de entradas, tipo de normalização, funções de treinamento, tipo de cálculo de erro são apresentados apenas como uma seleção simples.

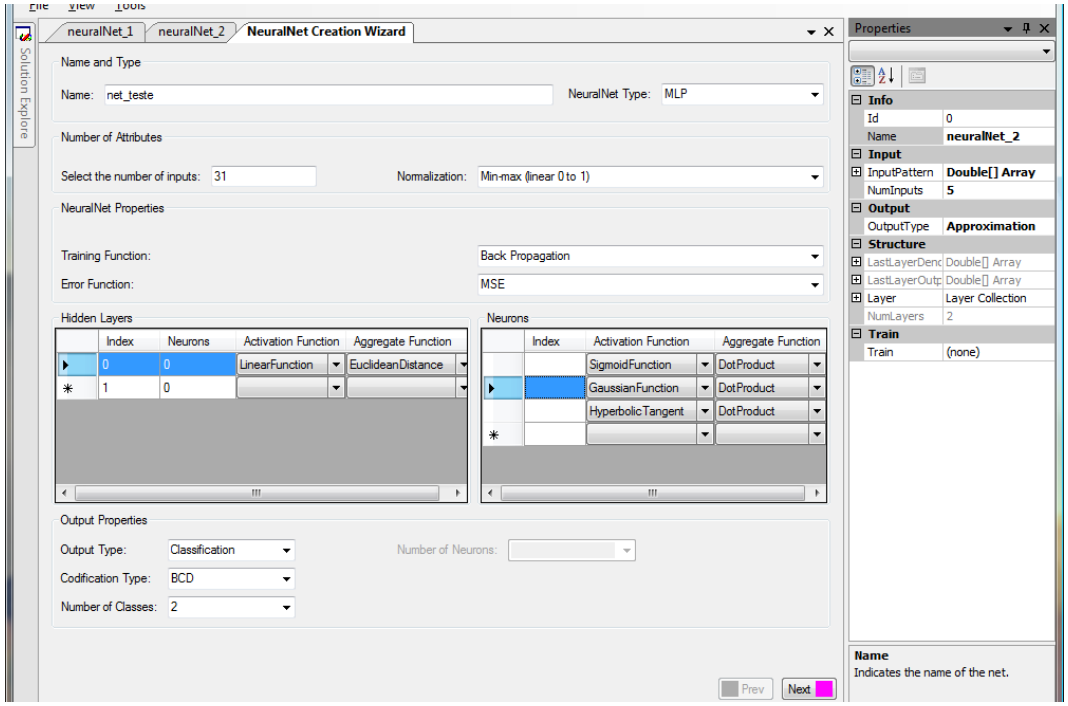


Figura 49 – Modelo passo a passo para se criar uma nova rede neural.

Depois de escolher essas primeiras características, o usuário pode montar a estrutura de camadas das redes adicionando os neurônios e selecionando, para cada neurônio, as funções de ativação e agregação, o que possibilita montar estruturas com várias funções distintas. Na terceira e última parte desta primeira tela é possível escolher o formato de saída da rede entre classificação e aproximação (para as redes de agrupamento, essas opções ficam desabilitadas). Como são muitas telas para descrever os diferentes caminhos possíveis no modo passo a passo, apenas a primeira tela é aqui apresentada.

4.4.3. Processo de Treinamento

Para treinar as redes, o Clinn oferece outro modelo passo a passo que cria estratégias de treinamento. Nessas estratégias, é possível escolher limites para variar o número de neurônios nas camadas, programar alterações nas funções de ativação e agregação, o número de épocas, o valor do momento e até ativar uma "sintonização" automática para os limites superiores e limites inferiores nas saídas dos neurônios da última camada em rede neurais de classificação. Por exemplo, o gráfico da Figura 50 mostra um limite superior (*upper threshold*) em 0,8 e um

limite inferior (*lower threshold*) em 0,1. Isto quer dizer que todos os valores maiores ou iguais a 0,8 serão interpretados como um e todos os valores menores que 0,1, como zero. A região entre esses dois valores é chamada de região de dúvida. Neste caso a rede neural não sabe informar a que grupo ou classe o padrão apresentado pertence. Nesta última opção, futuramente será possível executar um algoritmo genético para mapear os melhores valores.

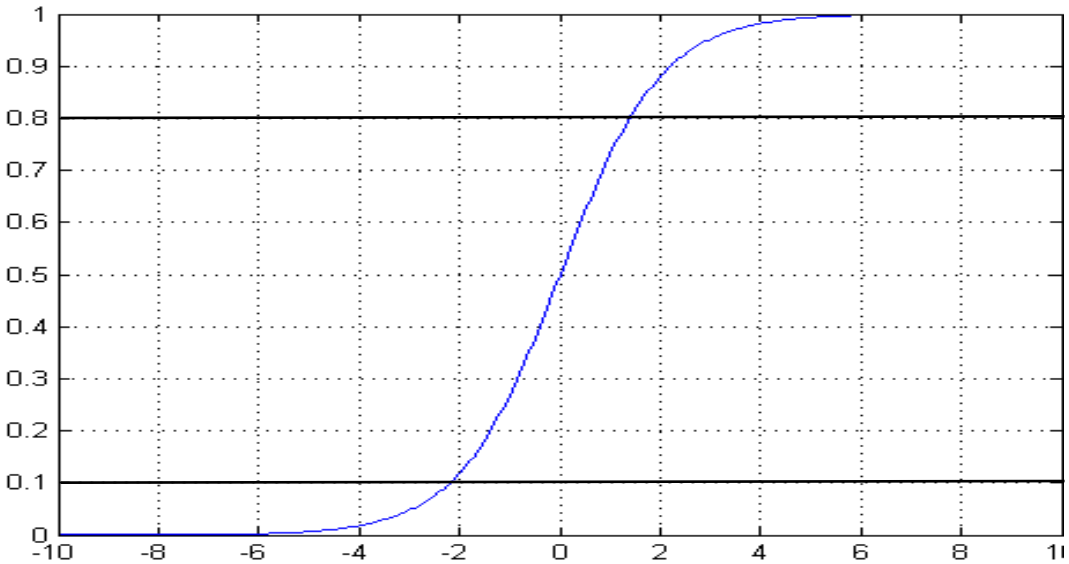


Figura 50 – Função sigmoid de um neurônio da última camada da rede neural.

5
Estudo de Casos

Neste capítulo serão apresentados sete experimentos onde foi utilizada a biblioteca ANNCOM para o treinamento das redes e de comitês de redes. Nesses experimentos, foram utilizados comitês de redes neurais para classificar clientes de uma distribuidora de energia elétrica como regulares ou irregulares. Em todos os experimentos, foram utilizadas quatro modelos de placas da NVIDIA e *driver* versão 190, conforme Tabela 3, onde GFLOP significa quantas operações de ponto flutuante são executadas por segundo em bilhões, e a Capacidade CUDA significa o quanto da arquitetura é suportada pela placa (min. 1.1 e Max. 1.3). Maiores detalhes em (NVIDIA, 2009).

Modelo	Número SP	Memória Global	Pico Medido	Capacidade CUDA
(I) Athlon 4400/1GB GeForce 8800 GT	112	512 MB	194 GFLPO/s	1.1
(II) Athlon 6000/2GB GeForce 260 GTX	224	896 MB	310 GFLOP/s	1.3
(III) Phenon II/16GB Tesla c1060	4×240	4×4096MB	4×340 GFLOP/s	1.3

Tabela 3 – Modelos de placas usadas no estudo de casos.

5.1.
Caso 1 – Treinamento dos Comitês para Classificação do Cliente Fraudador da Light

Perdas não-técnicas são responsáveis por um prejuízo anual de R$ 5 bilhões, correspondentes a 15.298 GW/h ou 5% de toda a energia distribuída no país. Uma das maiores perdas ocorre na rede da Light, que atende 3,79 milhões de unidades consumidoras em 31 municípios do Estado do Rio de Janeiro, incluindo a capital e a Região Metropolitana. A concessionária perde R$ 720 milhões anuais com esses desvios. Os números fazem parte de levantamento preparado pela Agência Nacional de Energia Elétrica (Aneel), com base nas informações enviadas ao órgão regulador pelas próprias distribuidoras, no momento em que elas se submetem ao

processo revisão tarifária, a cada quatro anos (MUNIZ, FIGUEIREDO, *et al.*, 2008).

O diretor-geral da Aneel, Jerson Kelman, ressalta que o furto de energia tem impacto direto na tarifa paga pelos consumidores honestos. No cálculo do reajuste tarifário, feito anualmente, a Aneel autoriza o repasse de parte das perdas com furtos para as contas de todos os consumidores. No entanto, a Aneel não possui um levantamento completo de quanto o consumidor de cada companhia paga a mais de tarifas devido aos furtos. Os cálculos com regras idênticas são feitos separadamente para cada distribuidora. No Rio de Janeiro, por exemplo, que é apontado como um dos estados campeões do furto de energia, os consumidores da Light (cerca de 10 milhões no Estado, incluindo a capital e algumas cidades do interior) pagam R$ 51 a mais, por ano (MUNIZ, FIGUEIREDO, *et al.*, 2008). De acordo com Vidinich, a prática vai desde o "furto rústico", como a instalação de um fio que puxa a energia de baixa tensão dos postes, até alterações "mais sofisticadas" nas instalações elétricas internas de casas e empresas, em que o relógio só registra parte dos gastos. De acordo com a Aneel, uma das ações mais "criativas" já detectadas foi o uso de açúcar no interior do relógio para atrair formigas e com isso retardar o funcionamento do medidor (MUNIZ, FIGUEIREDO, *et al.*, 2008).

Estes números denotam o quão importante e fundamental é o investimento de recursos nesta área. Ciente deste fato, a concessionária investiu milhões e dois anos de trabalho na implantação de um novo sistema de gestão comercial que vai beneficiar seus clientes com um atendimento diferenciado. Entre os benefícios do novo sistema está o livre acesso ao histórico de relacionamento com os clientes, o que torna o atendimento personalizado e ágil. A empresa muda a filosofia de gestão por unidade consumidora para gestão por cliente. Entre outros aspectos, este novo sistema proporcionará maior precisão nas inspeções de fraudes que são custosas para a companhia, afirma Mauro Andrade, Diretor de Projeto da Light (MUNIZ, FIGUEIREDO, *et al.*, 2008).

Além disso, com o objetivo de pressionar as concessionárias a ajustarem suas tarifas, através da redução de perdas, a Agência Nacional de Energia Elétrica adotará, a partir do próximo ciclo de revisão tarifária 2007-2010, novas regras para reconhecimento das perdas não-técnicas, fraudes e furtos. As empresas terão que apresentar metas de redução por subáreas dentro da área de concessão, além de uma meta geral. A ANEEL está tratando o tema como prioridade e pretende

adotar uma política de tolerância zero, diz Ricardo Vidinich, superintendente de regulação da comercialização (MUNIZ, FIGUEIREDO, *et al.*, 2008).

Na metodologia definida para o segundo ciclo de revisões tarifárias, a partir de 2007, foram introduzidas metas variáveis (levando em conta características de cada região) para redução de perdas não-técnicas. As empresas terão que apresentar metas de redução por subáreas dentro de sua área de concessão. Cada uma destas subáreas será dividida a partir de um índice de complexidade social. "É óbvio que as perdas provocadas por fraudes e furtos em Ipanema, não são as mesmas das verificadas na Favela do Jacarezinho, por exemplo, apesar de esta modalidade de crime não ser prática única de comunidades de baixa renda", comentou. A Aneel, então, fará uma comparação dos índices com os de outras distribuidoras para saber se as metas são factíveis e estão em linha com o setor. A Aneel pretende com essa medida estimular as empresas a combater furtos e fraudes. “Essa é uma abordagem que reconhece as especificidades de cada região do país”, comentou. Kelman (diretor-geral da Aneel) também afirmou que a resolução 456/2000, que regulamenta a relação distribuidora/consumidor, está sendo revista para adaptá-la à realidade atual do problema. A Aneel também finaliza a reformulação da resolução que estabelece relações entre clientes e concessionárias, de forma a fechar o cerco aos atos ilícitos (MUNIZ, FIGUEIREDO, *et al.*, 2008).

Atualmente a Light utiliza um conjunto de metodologias associadas a um serviço de denúncias que visam identificar clientes de baixa tensão suspeitos de estarem cometendo algum tipo de fraude. Estes clientes, classificados como suspeitos por estas metodologias, são indicados para um especialista da empresa que, com base em sua experiência, seleciona um determinado conjunto de clientes para serem inspecionados. Através deste processo, a Light tem obtido uma média de 25% de acertos na comprovação de clientes fraudadores (MUNIZ, FIGUEIREDO, *et al.*, 2008).

5.1.1. Estrutura do Sistema

A estrutura do sistema de treinamento das redes de classificação é dividida em três módulos: pré-processamento, Filtragem e classificação. O Pré-processamento inclui limpeza de dados, onde os dados duplicados ou corrompi-

dos, valores faltantes e *outliers* são removidos. A Codificação e Normalização de atributos categóricos transformam os dados de modo que eles tenham a mesma influência sobre processamento dos dados e a seleção das variáveis. Para maiores informações sobre os passos anteriores, verificar (HAYKIN, 2007). Os módulos de Filtragem e Classificação usam comitês com cinco redes neurais MLP e 28 atributos de entrada, uma camada escondida com diferentes números de neurônios e um neurônio na camada de saída. Uma visão geral do processo é mostrada na Figura 51, onde os atributos dos dados de entrada são descritos na Tabela 4. O motivo de se ter dois comitês, um de filtragem e outro de classificação, é porque os dados são muito ruidosos com muitos falsos verdadeiros e vice-versa. Utilizando dois comitês, o primeiro monta uma base de dados com apenas os registros mais claros. Essa nova base será usada para treinar os comitês de redes finais e que serão usados na classificação (MUNIZ, FIGUEIREDO, *et al.*, 2008).

Atributo	Descrição	Índice
TIPO_CLIENTE	Residencial, Comercial, Industrial e Rural	0..2
CODIGO_ANOMALIA_1	Código de irregularidade do leiturista.	3..5
CODIGO_ANOMALIA_2	Código de irregularidade do sistema.	6..8
CODIGO_ANOMALIA_3	Código de irregularidade de consumo	9..11
Indica_Irregularidade_Cliente	Indica se o cliente já teve irregularidade ou não	12..13
CON_VAL	Valor do consumo do local no mês em referência.	14
Média trimestral	Valores mensais definido por localidade e cliente.	15
Média semestral	Valores mensais definido por localidade e cliente.	16
Média anual	Valores mensais definido por localidade e cliente.	17
Indicador Trimestral 1	Indica variação do consumo nos três últimos meses.	18
Indicador Trimestral 2	Indica variação do consumo nos três últimos meses.	19
Indicador Trimestral 3	Indica variação do consumo nos três últimos meses.	20
Indicador Anual	Indica variação do consumo médio no ano.	21
Indicador de Ajuste	Indica variação do consumo nos 2 últimos meses.	22
Indicador de Tendência	Calculado pelo fator de tendência da série.	23
Temperatura Máxima	Temperatura máxima referente à região do cliente.	24
Temperatura Mínima	Temperatura mínima referente à região do cliente.	25
Indicador climático 1	Indicador climático de sensação térmica	26
Indicador climático 2	Indicador climático de El Niño e La Niña	27

Tabela 4 – Descrição dos atributos de entrada da base de clientes da Light.

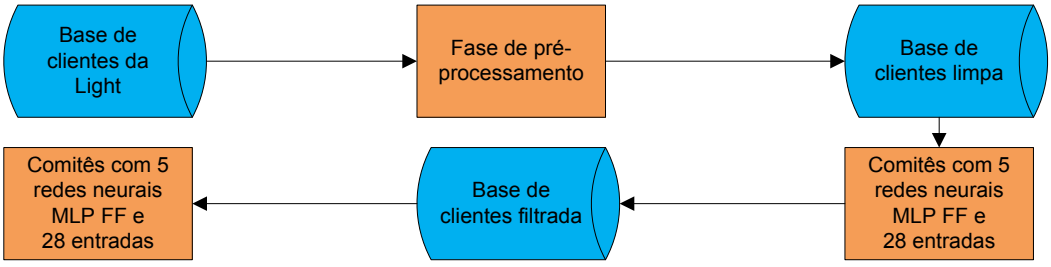


Figura 51 – Visão geral do processo de classificação dos clientes de baixa tensão.

5.1.2.
Resultados do Treinamento e Testes de Desempenho

Os resultados deste trabalho referem-se apenas ao desempenho do cálculo da Equação 29 das duas arquiteturas descritas anteriormente, sendo que um treinamento é realizado inteiramente na CPU fazendo o uso da biblioteca MKL, e outro onde a parte mais cara do treinamento é realizada pela placa de vídeo. Os tempos indicados nas tabelas abaixo se referem ao tempo de treinamento de uma época de um comitê com cinco redes neurais e estão em milissegundos.

Com relação à base de dados, vale ressaltar que a área de concessão da Light é dividida em cinco regiões: Leste, Oeste, Litorânea, Interior e Baixada; onde cada uma dessas regiões possui clientes residenciais e comerciais. Os seis experimentos abaixo se referem, respectivamente, aos clientes comerciais de todas as regiões, e os clientes residenciais das regiões indicas. Essa divisão foi feita, porque os clientes comerciais possuem um comportamento bem diferente dos clientes residenciais, mas homogêneo entre as regiões.

Como os clientes comerciais de todas as regiões mostram um mesmo comportamento, um comitê foi criado para classificar todos eles. Para estes clientes, a rede e os parâmetros de treinamento utilizados estão listados abaixo:

- 53.307 clientes;
- Tamanho da matriz Jacobiana: 348×53.307;
- 12 neurônios na camada oculta;
- Treinamento com 50 épocas

Configuração	Tempo MKL (ms)	Tempo CUDA (ms)	Aceleração
I	3.525	843	4,18
II	3.525	670	5,26
III	1.906	213	8,95

Tabela 5 – Tempos referentes à base de clientes comerciais.

Os resultados na Tabela 5 mostram os tempos de treinamento utilizando o MKL Intel e os tempos usando uma arquitetura CUDA. Os resultados do treinamento da rede com a base de dados de clientes comerciais já mostram uma diferença considerável entre CPU e GPU, mesmo a base não sendo muito grande.

Para os clientes da região Leste, os parâmetros utilizados estão listados abaixo:

- 98.983 clientes;
- Tamanho da matriz Jacobiana: 406×98.983;
- 14 neurônios na camada oculta;
- Treinamento com 50 épocas

Configuração	Tempo MKL (ms)	Tempo CUDA (ms)	Aceleração
I	7.878	953	8,27
II	7.878	889	8,86
III	4.427	232	19,08

Tabela 6 – Tempos referentes à base de clientes da região Leste.

Os resultados na Tabela 6 mostram os tempos de treinamento utilizando o MKL Intel e os tempos usando uma arquitetura CUDA. Nesse experimento, pode ser notado nos resultados, o aumento da diferença entre o tempo de treinamento na CPU e na GPU. Um dos fatores que podem explicar esse resultado é que, ao fazer o cálculo de Δx na GPU, a CPU precisa transferir para a memória todas as matrizes globais em todas as épocas da placa gráfica. Também é importante notar a diferença entre a freqüência da CPU e da GPU, algumas CPU têm uma freqüência quase cinco vezes maior do que algumas GPU. Nos experimentos realizados neste trabalho, essa diferença chegou a três vezes.

Para os clientes da região Oeste, os parâmetros utilizados estão listados abaixo:

- 123.272 clientes;
- Tamanho da matriz Jacobiana: 435×123.272;
- 15 neurônios na camada oculta;
- Treinamento com 50 épocas

Configuração	Tempo MKL (ms)	Tempo CUDA (ms)	Aceleração
I	11.372	1.140	9,98
II	11.372	1.138	9,99
III	6.203	274	22,64

Tabela 7 – Tempos referentes à base de clientes da região Oeste.

Os resultados na Tabela 7 mostram os tempos de treinamento utilizando o MKL Intel e os tempos usando uma arquitetura CUDA. O número de clientes da base de dados da região Oeste é ainda maior do que a base de clientes da região Leste. Como era de esperar, a diferença dos tempos de formação entre CPU e

GPU aumentou, uma vez que a placa gráfica ganha no paralelismo; isso significa que quanto mais dado, mais pode ser paralelizado.

Para os clientes da região Litorânea, a configuração é listada abaixo:

- 52.217 clientes;
- Tamanho da matriz Jacobiana: 290×52.217;
- 10 neurônios na camada oculta;
- Treinamento com 50 épocas

Configuração	Tempo MKL (ms)	Tempo CUDA (ms)	Aceleração
I	2.542	608	4,18
II	2.542	390	6,52
III	1.475	186	7,93

Tabela 8 – Tempos referentes à base de clientes da região Litorânea.

Os resultados na Tabela 8 mostram os tempos de treinamento utilizando o MKL Intel e os tempos usando uma arquitetura CUDA. O treinamento com a base de clientes da região Litorânea apresentou um resultado importante com relação à aceleração na primeira configuração. O tempo de treinamento na CPU e na GPU estão virtualmente empatados. As causas disso já foram descritas anteriormente, mas este fato indica que uma avaliação sobre a complexidade do problema deve ser feita para verificar se uma arquitetura em GPU é recomendada.

Para os clientes da região Interior, a configuração é listada abaixo:

- 27.353 clientes;
- Tamanho da matriz Jacobiana: 348×27.353;
- 12 neurônios na camada oculta;
- Treinamento com 50 épocas

Configuração	Tempo MKL (ms)	Tempo CUDA (ms)	Aceleração
I	1.700	499	3,53
II	1.700	312	5,45
III	992	202	4,91

Tabela 9 – Tempos referentes à base de clientes da região Interior.

Os resultados na Tabela 9 mostram os tempos de treinamento utilizando o MKL Intel e os tempos usando uma arquitetura CUDA. Ao treinar a base de clientes da região Interior, mesmo a primeira configuração mostra um ganho de desempenho de quase 400% maior em comparação com a CPU. Além disso, o custo da placa gráfica utilizada nessa configuração vale, aproximadamente, US$ 30, enquanto que o computador pode custar mais de US$ 300 nos Estados Unidos.

Para os clientes da região Baixada, a configuração é listada abaixo:

- 203.704 clientes;
- Tamanho da matriz Jacobiana: 406×203.704;
- 14 neurônios na camada oculta;
- Treinamento com 50 épocas

Configuração	Tempo MKL (ms)	Tempo CUDA (ms)	Aceleração
I	16.255	-	-
II	16.255	1.482	10,97
III	9.363	333	28,12

Tabela 10 – Tempos referentes à base de clientes da região Baixada.

Os resultados na Tabela 10 mostram os tempos de treinamento utilizando o MKL Intel e os tempos usando uma arquitetura CUDA. A maior base de clientes é a da região Baixada, com mais de 200.000 clientes. Nesse último experimento, pode ser observada a grande diferença entre a segunda e a terceira configurações, uma vez que a primeira configuração não pôde ser avaliada devido ao tamanho da memória. O tempo de treinamento da terceira configuração é quase três vezes mais rápido do que o tempo da terceira configuração, e razão para esta diferença é o número de processadores dentro destas duas placas gráficas, além do número de registros processados.

Para observarmos o desempenho da aplicação CUDA em diferentes tamanho de entrada, foi realizado um último teste onde foram colocados dados aleatórios e de tamanho crescente. A partir dos tempos gastos para os diferentes tamanhos de dados, foi gerado o gráfico da Figura 52.

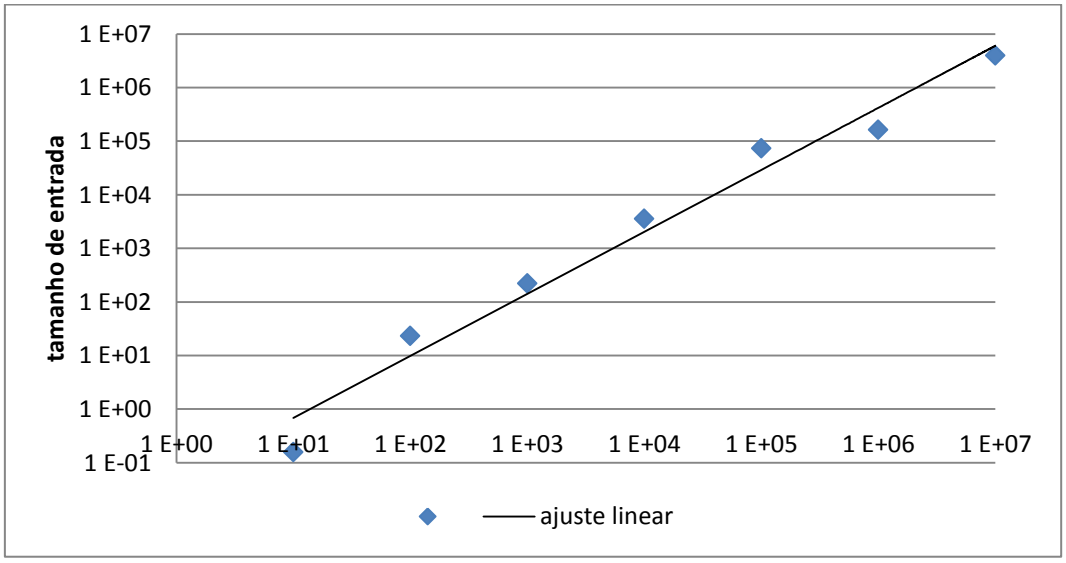


Figura 52 – Tempo (em segundos) vs. Tamanho (em número de elementos) da matriz de entada.

Os resultados obtidos nos experimentos acima mostram o poder de processamento das placas gráficas e como a programação paralela pode melhorar o desempenho do treinamento de redes neurais, bem como de várias aplicações. Nos experimentos realizados, para se calcular a Equação 29, os fatores determinantes para o tempo gasto de processamento eram o número de sinapses da rede neural e a quantidade de registros na base de dados, por causa do algoritmo utilizado. Um detalhe importante que pôde ser observado nos experimentos é a relação tamanho da matriz Hessiana e Aceleração. Ficou claro que se deve fazer uma avaliação quanto à necessidade de se utilizar uma arquitetura de placas gráficas de acordo com o tamanho do problema. Em alguns casos, o tamanho do problema não é grande o suficiente para se justificar o custo de implementação e, em outros, é tão pequeno que não compensa nem o tempo da transferência de dados entre GPU e CPU.

6
Conclusões e Trabalhos Futuros

Este trabalho surgiu da necessidade de se ter uma ferramenta que pudesse facilitar a criação de projetos usando redes neurais artificiais. Esta ferramenta deveria ser simples, mas robusta o suficiente para trabalhar com grandes bases de dados. Além disso, os processos de treinamento deveriam ser suficientemente rápidos para poderem realizar o treinamento de diversas redes rapidamente.

O primeiro passo foi o desenvolvimento de uma biblioteca para modelar e treinar redes neurais, a qual foi finalizada com a construção da ANNCOM que utiliza a tecnologia CUDA para melhorar o desempenho dos treinamentos. Vale salientar que a utilização do LMA em placas gráficas é nova e apresentou excelentes resultados, mesmo com apenas uma parte do processo de treinamento em execução na GPU. Após essa primeira fase, foi necessário desenvolver um ambiente gráfico que utilizasse completamente a biblioteca e fornecesse ao usuário final todas as opções oferecidas pela ANNCOM. Após um estudo das aplicações oferecidas pelo mercado, deu-se início à construção do Clinn. Um software que oferecia todas as possibilidades da biblioteca de componentes de redes neurais. É importante salientar que, além do objetivo primário de ser ter uma ferramenta, a interface gráfica oferecida pelo Clinn partiu da vontade de fazer um software que se apresentasse de maneira fácil e intuitiva para vários tipos de usuários com diferentes culturas de utilização de um computador.

Para realizar uma boa análise de desempenho nos experimentos comparativos realizados, todas as medidas de tempo tomadas sobre a CPU foram realizadas utilizando a biblioteca MKL com um programa escrito em linguagem C. Esta biblioteca tem um desempenho excepcional e algumas comparações feitas mostraram ganhos na ordem das centenas de vezes. Por isso, pode-se concluir que os resultados alcançados nesse trabalho foram muito bons, uma vez que o modelo proposto foi quase 30 vezes mais rápido que o modelo seqüencial usando MKL.

Em trabalhos futuros, a biblioteca ANNCOM contará com outros tipos de treinamento, como o para redes do tipo *Kohonen*, que já estão em fase de desen-

volvimento. Além disso, para obter melhor desempenho, todo o processo de formação será transferido para a placa de vídeo e as transferências de dados feitas a cada época serão substituídas por uma única transferência inicial das bases de treinamento. Dessa forma, talvez seja possível executar o treinamento em tempo real, que é exigido em vários tipos de problemas, por exemplo, um sistema de *up-scaling* de vídeo que precisa treinar uma rede a cada quadro.

Com relação ao Clinn, já existem muitas idéias, em forma de sugestões e críticas, para melhorar a funcionalidade. O processo de treinamento ainda está minimamente implementado e oferecerá suporte a mais tipos de bancos de dados diferentes, além de receber uma ferramenta bem completa para mineração e seleção de dados. Ainda com relação ao Clinn, já estão sendo estudados outros modelos que poderão auxiliar, em vários níveis da ferramenta, como um processo inteligente de escolha de variáveis, que descobrirá sozinho quais atributos são mais ou menos importantes para determinado problema.

7
Referências Bibliográficas

ADVE, S. V. et al. **Parallel Computing Research at Illinois:** The UPCRC Agenda. UPCRC. Redwood City: University of Illinois. Novembro 2008.

ALMASI, G. S.; GOTTLIEB, A. Highly parallel computing. **IBM Systems Journal**, Redwood City, CA, 1990. 165-166.

AMDAHL, G. M. **The validity of the single processor approach to achieving large-scale computing capabilities**. AFIPS Spring Joint Computer Conference. Atlantic City: AFIPS Press. 1967. p. 483–485.

BASS, L.; CLEMENTS, P. C.; KAZMAN, R. **Software Architecture in Practice**. 2nd Edition. ed. Canada: Addison-Wesley Professional, 2003.

BERGER, R. L.; CASELLA, G. **Statistical Inference**. 2 edition. ed. New York: Duxbury Press, 2001.

BERNSTEIN, A. J. Program Analysis for Parallel Processing. **IEEE Trans. em Electronic Computers**, n. 15, p. 757–762, Outubro 1966.

CULLER, D.; SINGH, J. P.; GUPTA, A. **Parallel Computer Architecture:** A Hardware/Software Approach. 1a Edição. ed. San Francisco: Morgan Kaufmann, 1998.

DAVIS, A. M. **Software Requirements:** Objects, Functions and States. 2nd Edition. ed. NJ: Prentice Hall, 1993.

ELETRICA, ENERGIA. Energia Eletrica. **Energia Eletrica**, 2006. Disponivel em: <http://www.eletrica.com.br/noticias/noticias.asp?Id_Noticias=27654>. Acesso em: 09 nov. 2006.

FLYNN, L. J. Intel Halts Development of 2 New Microprocessors. **The New York Times**, New York, Maio 2004.

GAMMA, E. et al. **Design Patterns:** Elements of Reusable Object-Oriented Software. Illustrated Edition. ed. New York: Addison-Wesley Professional, 1994.

GUSTAFSON, J. L. Reevaluating Amdahl's Law. **Communications of the ACM**, v. 31, p. 532-533, 1988.

HAGAN, M. T.; MENHAJ, M. B. Training feedforward networks with the Marquardt algorithm. **Neural Networks, IEEE Transactions on**, v. 5, n. 6, p. 989-993, Novembro 1994. ISSN 10.1109/72.329697.

HAYKIN, S. **Redes Neurais**. 2ª Edição. ed. Porto Alegre: Bookman, 2007.

ICA. **Relatório de Estudo do Contexto do Problema, Projeto 24: O Uso da Inteligência Artificial na Melhoria da Acertividade**. PUC-Rio. Rio de Janeiro, p. 30. 2007.

INTEL CORPORATION. Intel® MKL. **Intel® Software Network**, 2010. Disponivel em: <http://software.intel.com/en-us/intel-mkl/>. Acesso em: Janeiro 2010.

JANG, H.; PARK, A.; JUNG, K. **Neural Network Implementation Using CUDA and OpenMP**. Proceedings of the 2008 Digital Image Computing: Techniques and Applications. Washington: IEEE Computer Society. 2008. p. 155-161.

JOHNSON, R. C. EETimes.com - IBM Fellow: Moore's Law defunct. **EETimes.com**, 2009. Disponivel em: <http://eetimes.com/news/latest/showArticle.jhtml;jsessionid=CQ2BCKNVLPM5OQSNDLSCKHA?articleID=216403284>. Acesso em: 27 jan. 2010.

LAHABAR, S.; AGRAWAL, P.; NARAYANAN, P. J. **High Performance Pattern Recognition on GPU**. Hyderabad. 2009.

LAMPORT, L. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Progranm. **IEEE Transactions on Computers**, Washington, v. 28, n. 9, p. 690-691, Setembro 1979. ISSN 0018-9340.

LAMPORT, L. **Specifying Systems:** The TLA+ Language and Tools for Hardware and Software Engineers. San Francisco: Addison-Wesley Professional, 2002.

MARQUARDT, D. W. An Algorithm For Least-Square Estimation of Nonlinear Parameters. **Society for Industrial and Applied Mathematics**, v. 11, n. 2, p. 431-441, June 1963. ISSN 0036-1399.

MICROSOFT..NET Framework Conceptual Overview. **MSDN**, 2009. Disponivel em: <http://msdn.microsoft.com>. Acesso em: 27 Novembro 2009.

MICROSOFT CORPORATION. Visual Studio Developer Center. **MSDN**, 2010. Disponivel em: <http://msdn.microsoft.com/en-us/library/c5z9s1h4.aspx>. Acesso em: Janeiro 2010.

MUNIZ, C. et al. Indicação de Suspeitos de Irregularidade em Instalações Elétricas de Baixa Tensão. **Learning and Nonlinear Models**, Rio de Janeiro, v. 6, n. 1, p. 16-28, 2008. ISSN ISSN 1676-2789.

MUNIZ, C. et al. **Irregularity Detection on Low Tension Electric Installations by Neural Network Ensembles**. Neural Network Alliances with Cognitive Computing and Intelligent Machine Embodiments. Atlanta: IEEE. 2009. p. 2176-2182.

NGUYEN, D.; WIDROW, B. **Improving the Learning Speed of 2-Layer Neural Networks by Choosing Initial Values of the Adaptive Weights**. International Joint Conference on Neural Networks. Stanford: Stanford University. 1990. p. 21-26.

NVIDIA. **CUDA Programming Guide**. 2.3.1. ed. Santa Clara: NVIDIA, 2009.

NVIDIA. NVIDIA. **CUDA Zone**, 2010. Disponivel em: <http://www.nvidia.com/cuda>. Acesso em: jan. 2010.

REED, R. D.; MARKS, R. J. **Neural Smithing**. 1ª Edição. ed. Massachusetts: The MIT Press, 1999.

ROOSTA, S. H. **Parallel Processing and Parallel Algorithms:** Theory and Computation. 1a Edição. ed. New York: Springer, 1999.

SOMMERVILLE, I. **Engenharia de Software**. 6ª Edição. ed. São Paulo: Pearson, 2004.

THE MATHWORKS, INC. The MathWorks. **The MathWorks**, 2010. Disponivel em: <http://www.mathworks.com/>. Acesso em: Janeiro 2010.

TOMASULO, R. M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. **IBM Journal of Research and Development**, v. 11, n. 1, p. 25-33, Janeiro 1967.

WIKIPEDIA. Field-programmable gate array. **Wikipedia**, 2010. Disponivel em: <http://en.wikipedia.org/wiki/Field-programmable_gate_array>. Acesso em: 01 mar. 2010.

WIKIPEDIA. Quantum computer. **Wikipedia**, 2010. Disponivel em: <http://en.wikipedia.org/wiki/Quantum_computer>. Acesso em: 01 mar. 2010.

WIKIPEDIA. Wikipedia. **Wikipedia**, 2010. Disponivel em: <http://en.wikipedia.org>. Acesso em: Janeiro 2010.

YOURDON, E. **Modern Structured Analysis**. United States ed edition. ed. US: Prentice Hall PTR, 1988.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Administração
Baixar livros de Agronomia
Baixar livros de Arquitetura
Baixar livros de Artes
Baixar livros de Astronomia
Baixar livros de Biologia Geral
Baixar livros de Ciência da Computação
Baixar livros de Ciência da Informação
Baixar livros de Ciência Política
Baixar livros de Ciências da Saúde
Baixar livros de Comunicação
Baixar livros do Conselho Nacional de Educação - CNE
Baixar livros de Defesa civil
Baixar livros de Direito
Baixar livros de Direitos humanos
Baixar livros de Economia
Baixar livros de Economia Doméstica
Baixar livros de Educação
Baixar livros de Educação - Trânsito
Baixar livros de Educação Física
Baixar livros de Engenharia Aeroespacial
Baixar livros de Farmácia
Baixar livros de Filosofia
Baixar livros de Física
Baixar livros de Geociências
Baixar livros de Geografia
Baixar livros de História
Baixar livros de Línguas

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo