Petrônio Cândido de Lima e Silva

Paralelização do Algoritmo de Backpropagation para Clusters Beowulf

Montes Claros

2005

INSTITUTO EDUCACIONAL SANTO AGOSTINHO
FACULDADE DE CIÊNCIAS EXATAS E TECNOLÓGICAS
CURSO DE SISTEMAS DE INFORMAÇÃO

Paralelização do Algoritmo de Backpropagation para Clusters Beowulf

Monografia apresentada à Disciplina de POCC II como requisito parcial à obtenção do grau de Bacharel em Sistemas de Informação da Faculdade de Ciências Exatas e Tecnológicas das Faculdades Santo Agostinho, sob a orientação do Prof. Msc. Clarimundo Machado Moraes Júnior

Montes Claros, 23 de novembro de 2005

Monografia apresentada à Disciplina de POCC II como requisito parcial à obtenção do grau de Bacharel em Sistemas de Informação da Faculdade de Ciências Exatas e Tecnológicas das Faculdades Santo Agostinho, sob o título *"Paralelização do Algoritmo de Backpropagation para Clusters Beowulf"*, defendida por Petrônio Cândido de Lima e Silva e aprovada pela banca examinadorea constituída dos professores:

Prof. Msc. Clarimundo Machado Moraes Júnior
Orientador

Profa. Msc. Marilée Patta e Silva
FACET

Prof. Msc. Nilton Alves Maia
FACET

Dedicatória

À toda minha família, e em especial meus pais, Joaquim Cândido da Silva e Édina Lúcia de Lima e Silva, minha avô, Maria Salomé de Almeida (Dona Nina), e meus padrinhos João Domingos e Vita.

Vocês são meus espelhos e maiores mestres !!!

Agradecimentos

À Deus, pelo dom da vida e pela sua onipresença;

Ao Prof. Msc. Clarimundo Machado Moraes Júnior, pela motivação e dedicação como orientador;

À Profa. Msc. Marilee Patta, pela paciência e dedicação;

Aos meus grandes amigos de todas as horas: Bisa, Gefter e Richardson;

Aos meus colegas de Faculdade, em especial Lince Ierres, Luis Guisso, Maurício (Supermauz), Wellington e Fernando (Tim ou Claro ou sei lá....);

Aos meus colegas de trabalho da Santo Agostinho, em especial, os colegas da Divisão de Tecnologia de Informação, DCRA, Comunicação, Financeiro e Biblioteca;

Aos meus amores eternos: Maria Alice (Loli) e Maria Flávia (Papaty) e sua maravilhosa família.

Sem vocês, jamais teria chegado até aqui.

Resumo

As redes neurais artificiais têm se infiltrado em muitos setores como ferramenta de automação e controle. Todavia, o processamento numérico necessário para a obtenção de resultados da rede neural pode se tornar dispendioso dependendo do número de dados a serem tratados pela rede. O tempo de resposta da rede nesses casos é pouco interessante para aplicações *on-line*. Este trabalho dedicou-se ao estudo das formas de paralelização do algoritmo *Bacpropagation*, utilizado no aprendizado das redes neurais, em *clusters* multicomputadores da classe *Beowulf*. Foram estudadas arquiteturas paralelas e bibliotecas de programação paralela além dos fundamentos das redes neurais artificiais. Foi proposto um algoritmo paralelo para o algoritmo seqüencial e uma topologia da rede neural mais apropriada para o processamento paralelo.

Abstract

The artificial neural networks has infiltered in many sectors as an automation and control tool. But the numeric processing necessary for obtaining the net results can become huge depending on the input data size to be treated by the net. The net's response time in these cases isn't interesting for on-line aplications. These paper was dedicated in the study of the parallelizing of the Backpropagation Algorithm, utilized in the neural nets learning, in cluster of multicomputers of Beowulf class. Was studied parallel arquitectures and parallel programing libraries, and the artificial neural networks fundamentals. Was proposed an parallel algorithm for the sequetial algorithm and the neural net topology apropriated for the parallel processing.

Sumário

Lista de Tabelas p. ix

Lista de Figuras p. x

1 Introdução p. 2

1.1 Objetivos do Trabalho . . . . . p. 4

1.1.1 Geral . . . . . p. 4

1.1.2 Específicos . . . . . p. 4

1.2 Justificativas . . . . . p. 4

1.3 Metodologia . . . . . p. 5

1.3.1 Revisão Teórica . . . . . p. 5

1.3.2 Paralelização do Algoritmo *backpropagation* . . . . . p. 5

1.3.3 Implementação e Testes . . . . . p. 5

1.4 Organização do Trabalho . . . . . p. 6

2 Redes Neurais Artificiais p. 7

2.1 A Inteligência Artificial . . . . . p. 7

2.2 Histórico das Redes Neurais Artificiais . . . . . p. 8

2.3 O *Perceptron* . . . . . p. 9

2.3.1 O Neurônio Biológico . . . . . p. 9

2.3.2 O Neurônio Artificial . . . . . p. 11

2.4 Arquiteturas de Redes Neurais Artificiais . . . . . p. 15

2.5 Métodos de Aprendizado . . . . . p. 17

2.6 O Algoritmo *Backpropagation* . . . . . p. 19

2.7 Considerações sobre o projeto de RNA . . . . . p. 24

3 Programação Paralela p. 26

3.1 Introdução . . . . . p. 26

3.2 Granularidade . . . . . p. 27

3.3 Taxonomias . . . . . p. 28

3.3.1 Classificação de Flynn . . . . . p. 28

3.3.2 Classificação de Duncan . . . . . p. 29

3.4 Multicomputadores e Clusters . . . . . p. 30

3.5 Bibliotecas de Programação Paralela . . . . . p. 31

3.5.1 PVM . . . . . p. 33

3.5.2 MPI . . . . . p. 34

3.6 A biblioteca MPI . . . . . p. 34

3.6.1 A implementações MPI . . . . . p. 37

3.6.2 A biblioteca Java MPI . . . . . p. 37

3.7 Medidas de Desempenho . . . . . p. 37

3.7.1 Speed Up e Eficiência . . . . . p. 38

3.7.2 Escalabilidade . . . . . p. 39

4 Paralelização do Algoritmo Backpropagation p. 40

4.1 Implementação do *Cluster* . . . . . p. 40

4.2 Paralelização do Algoritmo . . . . . p. 43

5 Testes p. 48

5.1 Parâmetros da RNA . . . . . p. 48

5.2 O conjunto de testes . . . . . p. 52

5.3 Testes . . . . . p. 52

6 Conclusão p. 58

6.1 Considerações iniciais . . . . . p. 58

6.2 Contribuições . . . . . p. 58

6.3 Trabalhos Futuros . . . . . p. 59

Anexo A – Arquivos de Configuração do Cluster p. 60

A.1 Arquivo /etc/xinet.d/rsh . . . . . p. 60

A.2 Arquivo /etc/xinet.d/rlogin . . . . . p. 60

A.3 Arquivo /etc/pam.d/rsh . . . . . p. 61

A.4 Arquivo /etc/pam.d/rlogin . . . . . p. 61

A.5 Arquivo /etc/hosts.allow . . . . . p. 61

A.6 Arquivo /home/usuario/.rhosts . . . . . p. 61

Anexo B – Tabela de Processos Dos nós do cluster p. 62

Anexo C – Código Fonte da Rede Neural p. 63

C.1 Arquivo ativacao.java . . . . . p. 63

C.2 Arquivo neuron.java . . . . . p. 63

C.3 Arquivo layer.java . . . . . p. 65

C.4 Arquivo A.java . . . . . p. 69

C.5 Arquivo AA.java . . . . . p. 70

C.6 Arquivo B.java . . . . . p. 71

C.7 Arquivo BB.java . . . . . p. 71

C.8 Arquivo ConjuntoTreinamento.java . . . . . p. 72

C.9 Arquivo rede.java . . . . . p. 73

C.10 Arquivo RedeCluster.java . . . . . p. 79

Referências p. 88

Lista de Tabelas

1 Funções Básicas MPI . . . . . p. 35

2 Funções MPI de Comunicação Ponto a Ponto . . . . . p. 36

3 Funções MPI de Comunicação Coletiva . . . . . p. 37

4 Descricação do *hardware* do *cluster*. . . . . p. 40

5 Descrição dos Nós do *Cluster* . . . . . p. 41

6 Particionamento de disco dos Nós do *Cluster* . . . . . p. 41

7 Resultados aferidos com a função de ativação bipolar . . . . . p. 54

8 Resultados aferidos com a função de ativação binaria . . . . . p. 54

Lista de Figuras

1 Neuronio biológico (KANDEL; SCHWARTZ, 1991). . . . . p. 10

2 Representação do neurônio artificial (HAYKIN, 2001). . . . . p. 11

3 Representação matricial de X . . . . . p. 12

4 Representação vetorial de X . . . . . p. 12

5 Gráfico da Função Degrau . . . . . p. 13

6 Gráfico da Função Sigmóide . . . . . p. 14

7 Gráfico da Função Sigmóide Bipolar . . . . . p. 14

8 Redes recorrentes.(HAYKIN, 2001, p. 49) . . . . . p. 16

9 Uma RNA de camada única.(HAYKIN, 2001, p. 47) . . . . . p. 16

10 Uma RNA de múltiplas camadas.(HAYKIN, 2001, p. 48) . . . . . p. 17

11 Algoritmo *Backpropagation* no modo *On-line*(PIMENTEL; FIALLOS, 1999) p. 23

12 Algoritmo *Backpropagation* no modo *Batch*(PIMENTEL; FIALLOS, 1999) . p. 24

13 Algoritmo *Backpropagation* no modo *Block* (PIMENTEL; FIALLOS, 1999) . p. 25

14 Classificação de Flynn (MENDONÇA; ZELENOVSKY, 2005) . . . . . p. 28

15 Classificação de Duncan (SENGER, 1997) . . . . . p. 30

16 Fluxo de execução em um ambiente paralelo (DASGUPTA; KEDEM; RABIN, 1995) . . . . . p. 32

17 Classes da API JavaMPI (BAKER et al., 1999) . . . . . p. 38

18 Arquitetura do *Cluster* . . . . . p. 40

19 Arquivo /etc/hosts . . . . . p. 41

20 Arquivo /etc/lam/lam-bhost.def . . . . . p. 42

21 Compilação da Biblioteca mpiJava . . . . . p. 42

22 Criação de processo no *cluster* para java usando o script prunjava . . . . . p. 43

23 Criação de processo no *cluster* para java usando o comando mpirun . . . . p. 43

24 RNA seqüencial - Fase Forward . . . . . p. 43

25 RNA Paralela - Fase *Forward* . . . . . p. 46

26 RNA Paralela - Fase *Backward* . . . . . p. 46

27 Diagrama de Classes . . . . . p. 47

28 Parâmetro *a* na função sigmóide . . . . . p. 48

29 Parâmetro *a* na função sigmóide bipolar . . . . . p. 49

30 Parâmetro *a* na derivada da função sigmóide . . . . . p. 49

31 Parâmetro *a* na derivada da função sigmóide bipolar . . . . . p. 50

32 Impacto da taxa de aprendizado no número de iterações . . . . . p. 51

33 Matriz de entrada bipolar para o caractere A . . . . . p. 52

34 Matriz de entrada binária para o caractere A . . . . . p. 53

35 Vetor de saída bipolar para o caractere A . . . . . p. 53

36 Vetor de saída binária para o caractere A . . . . . p. 53

37 Variação do tempo . . . . . p. 55

38 Variação do Speed Up . . . . . p. 55

39 Variação da eficiência . . . . . p. 56

40 Variação do tempo e *speed-up* pelo número de processadores na função sigmóide . . . . . p. 56

41 Variação do tempo e *speed-up* pelo número de processadores na função sigmóide bipolar . . . . . p. 57

1 Introdução

Nas últimas décadas, duas interessantes vertentes da ciência da computação vêm tomando frente nos meios afins: as Redes Neurais Artificiais - RNA e os *clusters* multicomputadores. Extrapolando os meios científicos, essas inovações têm entrado nas indústrias, seja na melhoria dos processos industriais ou para baratear os custos de parque tecnológico, aliando a economia à potência computacional.

Desde o advento das redes de computadores, o paradigma computacional mudou de forma drástica. Era o fim dos *mainframes* e início da utilização da tecnologia Cliente/Servidor. Neste mesmo período ocorreu a popularização dos computadores pessoais, tornando-os acessíveis a empresas de médio e pequeno porte, e posteriormente, aos usuários domésticos.

Iniciava-se aí a corrida tecnológica por processadores mais velozes e por maior e melhor capacidade de armazenamento. Segundo PIROPO (2004), a Lei de Moore[1] previa que, a cada dezoito meses, a velocidade dos processadores dobraria, mas como lembra TANENBAUM (2003, p. 379), "No passado, a solução sempre foi deixar o relógio mais rápido. Infelizmente, estamos começando a atingir alguns limites fundamentais na velocidade do relógio."

O custo de grandes processadores para empreendimentos científicos é proibitivo até mesmo para as maiores universidades e para os centros de pesquisa, cuja necessidade de processamento é fator decisivo.

Forçados a aliar o poder da supercomputação com as restrições de orçamento, os cientistas da NASA criaram o primeiro multicomputador baseado em troca de mensagens, o Projeto *Beowulf*[2]. Esse projeto utilizava uma rede comercial interligando uma série de computadores pessoais, que compartilhavam entre si recursos computacionais.

De outro lado, os estudos do cérebro humano e seu funcionamento, levaram à criação

[1] Gordon Moore, funcionário da Intel, em 1956 previu o comportamento evolucionário dos processadores. Apesar de não ser uma lei científica, a previsão de Moore se concretizou.

[2] Conforme explicado no site do projeto, http://www.*Beowulf*.org, o nome *Beowulf* não tem nenhum significado técnico, sendo adotado pelos criadores da tecnologia como alusão a um personagem da mitologia nórdica

de modelos teóricos do pensamento e do aprendizado. A partir da década de 40, estudiosos de todo o mundo se voltaram para os então recentes *Perceptrons*, modelos matemáticos propostos por McCULLOCH e PITTS (apud HAYKIN, 2001). Após a formulação desse modelo, surgiram diversos outros estudos acerca dos neurônios artificiais e nasciam as RNA, a vertente conexionista da Inteligência Artificial. Na década de 80, os estudos se intensificaram e começaram a surgir os primeiros indícios práticos dessa tecnologia no mercado.

As redes neurais artificiais são uma realidade nos processos industriais, nos quais a automação industrial as emprega de forma ostensiva. Nesses meios, as redes neurais chegam a ter entre dezenas e centenas de nós, e seu processamento é centralizado. Nesse trabalho realizou-se uma pesquisa sobre as Redes Neurais Artificiais Paralelas a partir da paralelização do algoritmo de aprendizado das RNA sequenciais, propondo uma arquitetura adaptada aos *clusters Beowulf*.

A implementação das RNA em que se fazem necessários dezenas ou até centenas de sensores produzindo estímulos, isto é, entradas para a rede neural, e há, conseqüentemente, dezenas ou centenas de neurônios artificiais, gera, alta carga de processamento além de consumir muita memória, especialmente para as RNA, cuja arquitetura engloba várias camadas de neurônios. Mas, em uma aplicação na qual há milhares de entradas a serem processadas ou o aprendizado da rede envolve um universo muito extenso de padrões de treinamento a serem apresentados, o processamento gasto para produzir as saídas da rede gasta todos os recursos da máquina, mesmo possantes estações de trabalho utilizadas nos grandes centros de pesquisa.

O campo de aplicação das redes neurais artifícias é de fato extenso. Segundo BRAGA, CARVALHO e LUDERMIR (2000, p. 217), "Pelo fato de as RNA serem aptas a resolver problemas de cunho geral, tais como aproximação, classificação, categorização, predição, etc., a gama de áreas onde estas podem ser aplicadas é bastante extensa."

As aplicações de RNA, geralmente, otimizam e automatizam processos, dando-lhes flexibilidade e adaptibilidade a estímulos externos. Com isso, as indústrias as empregam em larga escala, bem como institutos de pesquisa científica, as empregam no aperfeiçoamento das técnicas e ferramentas diversas.

O problema da alta carga computacional envolve, necessariamente, o uso de computadores paralelos. Grandes empresas e indústrias tendem a investir em supercomputadores proprietários baseados em multiprocessadores, um investimento caro para as instituições de pesquisa e universidades. Constata-se, a partir da Resenha Estatística do CNPq[3], que o investimento federal em fomento à pesquisa, mesmo nas maiores instituições de pesquisa brasileiras, não

[3]http:// ftp.cnpq.br/pub/doc/aei/resenha.pdf, pág. 21.

permite o custeio de supercomputadores, cujo valor excede a casa dos milhões de dólares. Na lista *TOP 500*[4], que elabora o *ranking* dos 500 supercomputadores mais rápidos do mundo, empresas privadas do Brasil ocupam o 53°, 129° e o 453° lugares, mas as instituições de pesquisa públicas sequer são citadas na lista. A TOP 500 é mantida por *University of Tenesse*[5], *National Energy Research Scientific Computing Center*[6] e pela *University of Mannhein* [7]

1.1 Objetivos do Trabalho

1.1.1 Geral

Paralelizar o algoritmo de *backpropagation* para redes neurais *Multi-Layer Perceptrons* em *cluster* da classe *Beowulf*.

1.1.2 Específicos

1. Avaliar o tempo de aprendizado e resposta da rede;
2. Sugerir uma topologia de distribuição dos nós da rede neural entre os processadores do *cluster*, de modo que possibilite maximizar o processamento e diminuir a comunicação;

1.2 Justificativas

No contexto das grandes aplicações industriais e científicas das RNA, justificou-se a necessidade do estudo sobre a Redes Neurais Artificiais Paralelas que são a otimização dos algoritmos de treinamento das redes neurais, visando redução do tempo de resposta das redes neurais que envolvem um universo extenso de padrões ou entradas da rede. A paralelização do algoritmo *backpropagation*, responsável pelo aprendizado nas redes neurais, em *clusters Beowulf*, visa eliminar gargalos de processamento com uma alternativa que alia poder de processamento e baixo custo financeiro, ideal para os centros de pesquisa científica e instituições de médio e pequeno porte.

[4] http://www.top500.org/
[5] http://icl.cs.utk.edu/
[6] http://www.nersc.gov/
[7] http://www.uni-mannheim.de/english/

1.3 Metodologia

O desenvolvimento deste trabalho deu-se em três fases distintas:

1. revisão teórica;
2. paralelização do algoritmo *backpropagation*;
3. implementação e testes.

1.3.1 Revisão Teórica

Primeiramente, fez-se necessária revisão bibliográfica e teórica das tecnologias envolvidas, a fim de melhor compreender e apronfundar os conceitos necessários e rever os experimentos já existentes no mesmo campo de pesquisa. Para esse fim, uma pesquisa bibliográfica que abranja a literatura escrita e a internet foi realizada.

1.3.2 Paralelização do Algoritmo *backpropagation*

Esta fase foi divida em três etapas: estudo analítico do Algoritmo; paralelização do algoritmo; definição de arquitetura de RNA. O estudo analítico do algoritmo visou levantar os pontos paralelizáveis e os pontos seqüenciais do algoritmo, bem como regiões concorrentes. A paralelização do algoritmo utilizou essas informações para definir o algoritmo paralelo ótimo e escalável para o algoritmo seqüencial.

Há diversas metodologias para treinar o *perceptron*, mas esse trabalho tratará somente do aprendizado supervisionado por correção de erro.

Em seguida, foi estudada a melhor arquitetura de Rede Neural para *clusters*, de modo a resolver o problema da granularidade, minimizando a comunicação e maximizando o processanto entre os nós do *cluster*.

1.3.3 Implementação e Testes

Esta fase foi dividida em quatro etapas: implementação do *cluster*; codificação do algoritmo; definição do conjunto de aprendizado e os testes de desempenho.

A implementação do *cluster Beowulf* utilizou quatro CPUs, sendo uma mestre e quatro escravos. Esta implementação utilizou o sistema operacional de código aberto Linux, e a

implementação LAM do padrão MPI. Após implementado o *cluster*, foi feita a codificação do algoritmo paralelo utilizando a linguagem JAVA e a biblioteca MPI.

Para treinar a rede e fazer os testes de desempenho, definiu-se um modelo de problema do qual foi retirado o conjunto de dados utilizado para treinar e testar a rede. Com o modelo pronto, foram efetuados os testes de desempenho, definindo as métricas ideais para a rede.

1.4 Organização do Trabalho

A monografia foi estruturada em seis capítulos: introdução; redes neurais artificiais; arquiteturas paralelas; programação paralela; paralelização do algoritmo de *backpropagation*; conclusão e trabalhos futuros.

Na introdução apresentou-se os objetivos que se pretendem alcançar com a pesquisa, as motivações e a justificativa da relevância deste para as áreas afins. Na metodologia foi exposto todo o processo de execução da trabalho e na organização explana-se de como foi organizada a estrutura de capítulos do trabalho.

Em seguida, discutiu-se as Redes Neurais Artificiais. Pretendeu-se, com esse capítulo, uma revisão teórica embasadora de todo o restante do trabalho, no que tange às redes neurais. Primeiro foi dada uma breve visão histórica do desenvolvimento das redes neurais e surgimento das principais teorias. As arquiteturas de redes neurais, algoritmos e técnicas de aprendizado foram discutidas a seguir.

Após a discussão teórica das redes neurais, abordou-se as arquiteturas paralelas com ênfase na computação em *Cluster*. Aqui discutem-se as principais técnologias de máquinas paralelas e seu desenvolvimento histórico. Na Programação Paralela discutiu-se as metodologias de paralelização de algoritmos e bibliotecas próprias. Um estudo sobre a eficiência dos algoritmos paralelos e suas métricas foi também definido.

Após os capítulos teóricos, que embasaram todos os temas necessários, o capítulo Paralelização do Algoritmo *Backpropagation* descreve em pormenores a implementação do *cluster Beowulf* e a paralelização do algoritmo de *backpropagation*, bem como a arquitetura de RNA utilizada.

Finalizando-se o trabalho, o capítulo Conclusão e Trabalhos Futuros descreve os resultados práticos da pesquisa e as perspectivas de novos trabalhos na área.

2 Redes Neurais Artificiais

2.1 A Inteligência Artificial

O termo Inteligência Artificial - I.A., foi primeiro cunhado por John McCarthy, em 1956, conforme mostra TEIXEIRA (1998), em uma conferência em Dartmouth nos Estados Unidos. Nessa conferência estavam reunidos os maiores nomes da Ciência da Computação à época, para discutir as bases para o desenvolvimento de uma ciência da mente, a qual deveria tomar como modelo, o computador digital. Havia, nessa época, uma discussão de qual campo pertenceriam os estudos nessa área, se à matemática, à computação, à teoria de controle, à pesquisa operacional ou à teoria da decisão. Em todas essas áreas é reflexos da inteligência artificial mas atualmente seu estudo é um sub-campo da Engenharia e da Ciência da Computação.

A I.A., é formada do diálogo interdisciplinar entre uma gama de outras áreas do saber, que vão desde as ciências biológicas, passando pelas ciências humanas e sociais aplicadas até as exatas. Desde suas origens, pesquisadores de diversos campos como psicólogos, lingüistas, matemáticos, cientistas da computação e economistas têm somado seus saberes para construir "máquinas que funcionarão de forma autônoma em ambientes complexos e mutáveis", como mostra RUSSELL e RUSSELL (2004, p. 19).

Existem diversos paradigmas dentro da I.A que procuram organizar o aprendizado da máquina segundo alguma teoria ou área de conhecimento, como a psicologia ou a estatística, e propondo uma série de técnicas e algoritmos estruturados apartir dessas teorias.

A tendência Simbolista que estuda os sistemas especialistas, busca organizar o conhecimento humano sob a forma de símbolos. Dessa forma, o aprendizado de um determinado conceito se dá "através da análise de exemplos e contra exemplos desse conceito", como apresentado por REZENDE (2003, p. 93). Desse estudo surgiram as linguagens Lisp e outras linguagens orientadas a objeto, bem como os Sistemas Baseados em Conhecimento e os sistemas de Raciocínio Baseado em Casos. Os Sistemas Baseados em Conhecimento, também conhecidos como Sistemas Especialistas, possuem um banco de dados com informações es-

pecíficas de uma determinada área de conhecimento e se baseiam em mecanismos de busca nesses bancos de dados com a aplicação de regras determinísticas de decisão. Esses sistemas foram primeiramente desenvolvidos para substituir especialistas em determinadas áreas, como médicos e advogados. Os sistemas de Raciocínio Baseado em Casos são um sub-campo dos Sistemas Baseados em Conhecimento que ganharam um destaque recente pela sua capacidade de agregar novos conhecimentos à medida que novos problemas ainda não conhecidos surgem.

REZENDE (2003) mostra que outra tendência, a Estaticista, procura utilizar modelos estatísticos para encontrar uma boa aproximação do conceito induzido. As Redes Bayesianas foram criadas para permitir a representação eficiente do conhecimento incerto e raciocínio rigoroso. Baseiam-se na Teoria das Probabilidades, a partir do conhecimento prévio do problema, e das probabilidades de ocorrência dos eventos dentro do problema, podem-se combinar as probabilidades para determinar a probabilidade final de uma sequência de eventos. Conhecendo o evento mais provável, torna-se fácil estipular a melhor ação a ser tomada.

Já a tendência Evolucionista, estuda os algoritmos genéticos, baseados na teoria evolucionária de Charles Darwin. Um algoritmo genético consiste em uma população de elementos que competem para solucionar um determindado problema. Elementos que possuem uma performance fraca são descartados, enquanto os elementos mais fortes proliferam, produzindo variações de si mesmos.

A tendência conexionista também conhecida como *ciência cognitiva*, que será o foco deste trabalho, lida com as Redes Neurais e será abordada em profundidade adiante. Há outras técnicas dentro da I.A., como a Lógica *Fuzzy*, mas o seu estudo não fazem parte do escopo desta pesquisa.

2.2 Histórico das Redes Neurais Artificiais

A ciência cognitiva é um campo interdisciplinar que se vale das idéias e métodos da psicologia cognitiva, psicobiologia, da inteligência artificial, da filosofia, linguística e da antropologia. Destes campos, a inteligência artificial, já tratada na seção 2.1, e a psicologia cognitiva são os que mais influenciaram as redes neurais artificiais (RNA). Segundo STERNBERG (2000, p. 22), "a psicologia cognitiva trata do modo como as pessoas percebem, aprendem, recordam e e pensam sobre a informação".

A psicologia cognitiva se baseia, historicamente, em duas diferentes abordagens de compreensão da mente humana: a da filosofia, que se baseia na instrospecção, e a fisiologia, que se baseia em métodos empíricos. Os estudos filosóficos investigam as estruturas simbólicas do

raciocício, como os elementos da lógica e as representações mentais. A fisiologia, e em especial a neuroanatomia, busca os fatores e estruturas biológicas que fundamentam o raciocínio.

As pesquisas da neuroanatomia deram origem aos primeiros modelos empíricos sobre os neurônios biológicos. Na década de 40, o psiquiatra e neuroanatomista Warren McCulloch apresenta um estudo dos eventos no sistema nervoso, que é conhecido hoje como o primeiro trabalho na área de I.A. A ele se juntou o matemático Pitts em 1942, que formulou os modelos matemáticos do comportamento neuronal, como mostra McCULLOCH e PITTS (apud HAYKIN, 2001). O aprendizado nas redes biológicas foi o objeto de Donald Hebb, em um trabalho de 1949, conforme HEBB (apud HAYKIN, 2001). Nesse trabalho, Hebb expõe sua teoria de que o aprendizado é baseado no reforço das sinapses entre dois neurônios ativados, que em termos matemáticos se representaria pela variação dos pesos de entrada dos neurônios.

Na década de 50, ROSENBLATT (apud HAYKIN, 2001) demonstrou o novo modelo do perceptron com sinapses ajustáveis e um algoritmo de treinamento. Em 1969, MINSKY e PAPERT (apud HAYKIN, 2001) publicaram um artigo que chamava a atenção para as limitações dos perceptrons, demonstrando como os perceptrons podiam tratar apenas de problemas linearmente separáveis. Na década de 70, o estudo das redes neurais caiu no ostracismo, com a grande repercursão do trabalho de Misky e Papert.

A grande retomada do estudo das redes neurais aconteceu na década de 80, com a publicação dos trabalhos de HOPFIELD (apud HAYKIN, 2001) e da apresentação do algoritmo Backpropagation por RUMELHART, HINTON e WILLIAMS (apud BRAGA; CARVALHO; LUDERMIR, 2000). Após essa fase o estudo das redes neurais artificiais cresceu exponencialmente, chegando aos dias atuais com inúmeras aplicações teóricas e implementações.

2.3 O *Perceptron*

2.3.1 O Neurônio Biológico

Conforme BRAGA, CARVALHO e LUDERMIR (2000), o sistema nervoso humano, que tem no cérebro seu principal órgão, é responsável pela emoção, raciocínio e percepção, bem como pela execução de funções sensoriomotoras e autônomas.

O sistema nervoso é divido em duas partes principais: o Sistema Nervoso Central - SNC e o Sistema Nervoso Periférico - SNP. O SNC compõe-se do cérebro e da medula espinhal, enquanto o SNP se compõe de todas as demais células nervosas, cuja função é transmitir a informação entre o sistema nervoso central e os nervos que se localizam nos orgãos, como

apresentado por STERNBERG (2000).

O cérebro é formado por células fundamentais chamadas neurônios, cuja estrutura é ilustrada na figura 1. Cada um desses neurônios processa e se comunica com milhares de outros, continuamente, e em paralelo. A rede desses neurônios é que dá, ao cérebro, a capacidade de reconhecer padrões e relacioná-los e de armazenar conhecimento, segundo mostra BRAGA, CARVALHO e LUDERMIR (2000). A medula espinhal é responsável por conduzir informação entre o SNP e o cérebro.

Um neurônio é divido em três seções: o corpo, os dentritos e o axônio. Os axônios, as linhas de transmissão e os dentritos, as zonas de recepção são dois filamentos celulares morfologicamente distintos.

A conexão entre um axônio de um neurônio com o dentrito de outro neurônio é conhecida como sinapse.

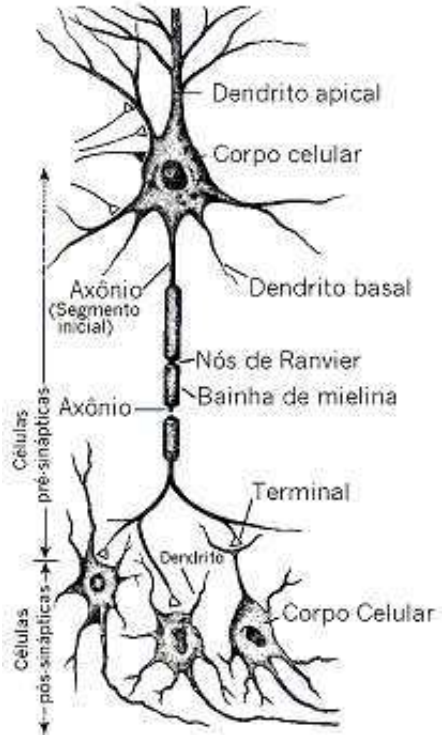

Figura 1: Neuronio biológico (KANDEL; SCHWARTZ, 1991).

Os neurônios são, entre cinco e seis vezes mais lentos, que os transístores de silício em um processador atual. O tempo de resposta de um transístor está na casa dos nanosegundos (10^{-9} segundos). Já nos neurônios, esse tempo é de milissegundos (10^{-3} segundos). Essa lentidão dos neurônios é compensada pela quantidade de neurônios, cujo número estimado é de 10 bilhões, e pelo número de conexões entre eles, algo em torno de 60 trilhões, conforme HAYKIN (2001).

Outra importante característica das redes neurais biológicas é a *tolerância à falhas*. O ser humano é capaz de tolerar danos não generalizados no sistema neural, especialmente no cérebro onde estão concentrados os mais de 100 bilhões de neurônios do nosso sistema nervoso. No caso de danificações e perdas de neurônios, outros neurônios podem ser treinados e assumir as funções das células danificadas. A despeito das perdas contínuas de neurônios devido à idade e outras causas, o homem jamais perde a propriedade de aprender, como mostra FAUSETT (1994).

2.3.2 O Neurônio Artificial

As redes neurais artificiais são algoritmos numéricos que simulam o comportamento dos neurônio biológicos, baseados no modelo proposto em 1943 por McCULLOCH e PITTS (apud HAYKIN, 2001). A formalização do *perceptron*, se deu por Rosenblatt em 1958, enriquecido, ainda, pelos trablhos de HEBB (apud HAYKIN, 2001) em 1949.

O *perceptron*, ou neurônio artificial, é um modelo matemático do comportamento neuronal e unidade fundamental de processamento da rede neural. Como ilustrado na figura 2, o *perceptron* é formado por:

1. um vetor de entrada $X = \{x_1, x_2, x_3, \ldots, x_n\}$;
2. um vetor de pesos $W = \{w_1, w_2, w_3, \ldots, w_n\}$;
3. um valor de *bias* representado por b;
4. um combinador linear net;
5. uma função de ativação φ;

O padrão de entrada X pode ser representado, como mostra a figura 3, na forma matricial.

Computacionalemte a matriz de entrada pode ser representada como um vetor, conforme figura 4.

Cada peso w_i está relacionado a uma das entradas x_i do neurônio e representa o estado de ativação ou força daquela entrada no perceptron. O combinador linear net é a função do produto interno do vetor de entradas do neurônio com seu vetor de pesos mais o *bias* b_j, na forma:

$$net_{kj} = \sum_{i=1}^{n} w_{ji} x_{ki} + b_j \tag{2.1}$$

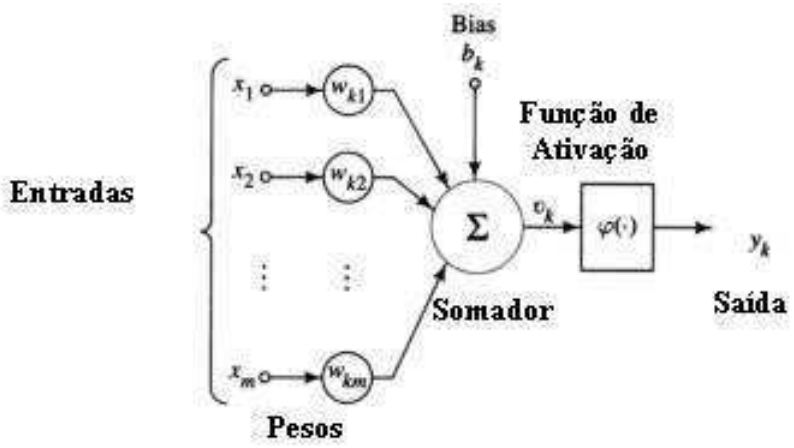

Figura 2: Representação do neurônio artificial (HAYKIN, 2001).

```
0 0 1 0 0
0 1 0 1 0
1 0 0 0 1
1 1 1 1 1
1 0 0 0 1
```

Figura 3: Representação matricial de X

```
0 0 1 0 0 0 1 0 1 0 1 0 0 0 1 1 1 1 1 1 1 0 0 0 1
```

Figura 4: Representação vetorial de X

Onde:

- k é o índice do padrão de entrada;
- j é o índice do neurônio na rede;
- n é o número de entradas do padrão k para o *perceptron* j;
- i é o índice de entrada x no padrão k, e do peso w correspondente no *perceptron* j.

A função de ativação φ que é utilizada para restringir a amplitude de saída de um neurônio, ou seja, é uma função aplicada sobre a saída net_{kj} do combinador linear. O resultado da função de ativação, y_{kj} corresponde à saída do neurônio ao padrão apresentado, na forma:

$$y_{kj} = \varphi(net_{kj}) \tag{2.2}$$

Há vários tipos de funções de ativação que são utilizadas de acordo com a aplicação da rede neural. As funções de ativação clássicas são a *degrau*, *sigmóide* e a *sigmóide bipolar*.

A função degrau, também conhecida como *função de limiar* (*threshold*), ou *função de Heaviside*, é utilizada no neurônio de McCULLOCH e PITTS (1943) e é expressa na forma:

$$y = \begin{cases} 0 & \text{se} \quad x \leq 0 \\ 1 & \text{se} \quad x > 0 \end{cases} \tag{2.3}$$

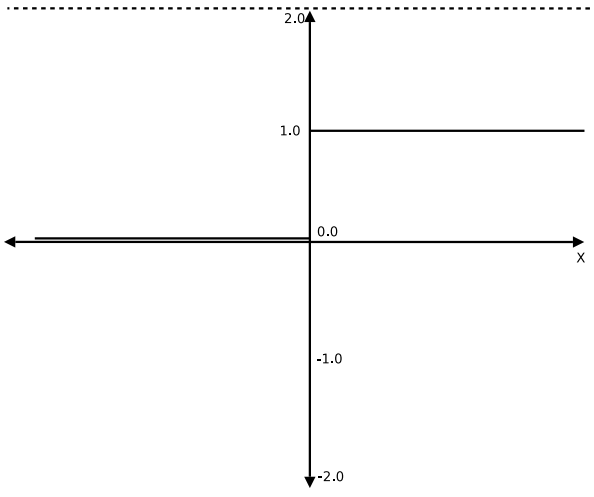

Figura 5: Gráfico da Função Degrau

Nessa função de ativação, o valor de limiar θ delimita a mudança de comportamento da função de ativação, elevando ou diminuindo o valor final de saída da função. Na figura 5, o valor de θ é igual à zero.

A função sigmóide é uma função não linear, crescente, contínua e derivável, que assume valores no intervalo [0,1], sendo expressa como:

$$y = \frac{1}{1 + e^{-ax}} \tag{2.4}$$

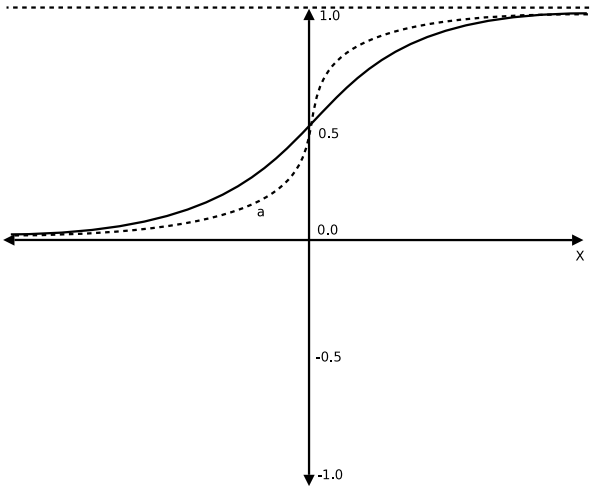

Figura 6: Gráfico da Função Sigmóide

O parâmetro a define a inclinação da curva sigmóide, como pode ser visto na figura 6. Valores altos de a produzem uma inclinação mais acentuada, representada na figura pela curva serrilhada, enquanto valores menores produzem curvas mais suaves, representadas pela linha contínua. A função sigmóide bipolar é uma variação da sigmóide, mas pode assumir valores no intervalo [-1,1], e é definida como:

$$y = \frac{2}{1 + e^{-ax}} - 1 \tag{2.5}$$

Funções sigmoidais têm a vantagem de serem deriváveis e sua derivada é de fácil computação. A simples relação entre o valor da função em um ponto e o valor da derivada no mesmo ponto reduzem custo computacional durante o treinamento, conforme FAUSETT (1994). As derivadas da sigmóide e da sigmóide bipolar são:

$$y = \varphi y[1 - y] \tag{2.6}$$

$$y = \frac{\varphi}{2}[1 + y][1 - y] \tag{2.7}$$

Existem ainda as Funções de Base Radial - RBF[1] que incorporam diferentes técnicas de

[1]Radial Basis Functions

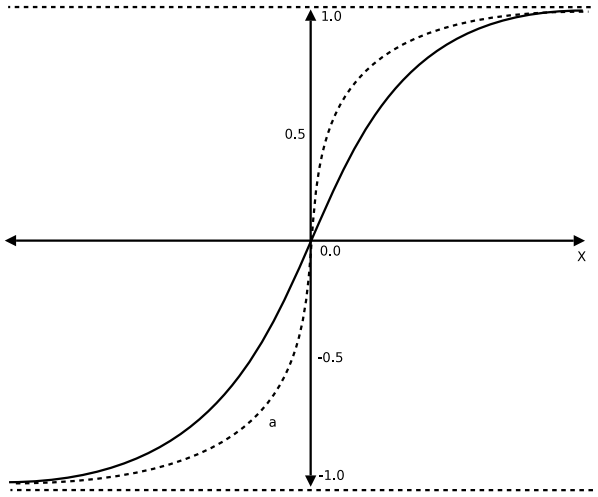

Figura 7: Gráfico da Função Sigmóide Bipolar

treinamento que fogem ao escopo deste trabalho.

2.4 Arquiteturas de Redes Neurais Artificiais

O arranjo dos neurônios em camadas e os padrões de conexão entre as camadas são chamados de arquiteturas de rede. A definição da arquitetura de uma RNA é um parâmetro essencial na sua concepção dado que ela pode restringir o tipo de problema que pode ser tratado pela rede. A arquitetura da rede também influencia no algoritmo de treinamento utilizado. Segundo BRAGA, CARVALHO e LUDERMIR (2000, p. 11), "Fazem parte da definição da arquitetura da rede os seguintes parâmetros: número de camadas da rede, número de nodos em cada camada, tipo de conexão entre os nodos e topologia da rede."

Esses arranjos estruturais formam grafos orientados, onde o fluxo dos sinais depende o número de camadas e do tipo de interligações entre elas. Usualmente, dentro da mesma camada, os neurônios possuem o mesmo tipo de função de ativação e o mesmo padrão de conexões com outros neurônios, conforme FAUSETT (1994).

Quanto ao número de camadas, as redes neurais geralmente são classificadas em redes de camada única, quando existem somente uma camada de entrada e outra de saída. As redes multi-camadas possuem uma camada de entrada, uma camada de saída e uma ou mais camadas escondidas.

As redes podem ainda serem classificadas quanto ao tipo de conexões entre os neurônios. Conexões entre os neurônios que formam grafos direcionados acíclicos são chamadas de redes *feedforward*. Conexões entre os neurônios que formam ciclos, realimentando sua própria camada ou camadas anteriores, são chamadas de redes *feedback* também conhecidas como redes

recorrentes, conforme figura 8. Segundo HAYKIN (2001, p. 49), "a saída de um neurônio para alimentar outro neurônio anterior se dá pelo operador de atraso unitário z^{-1}". Quando todas as ligações da rede formam cíclos então arede é dita auto-associativa, sendo usada para recuperação de um padrão de entrada.

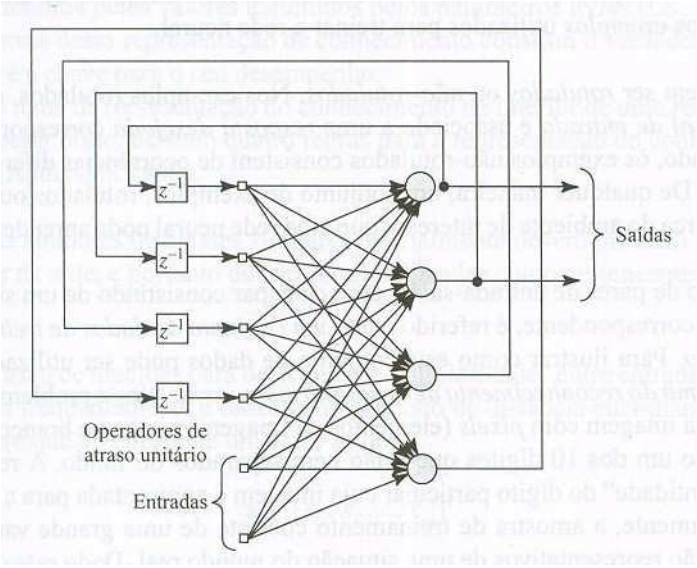

Figura 8: Redes recorrentes.(HAYKIN, 2001, p. 49)

As redes são completamente conectadas quando todos os nós de uma camada da rede estão conectados a todos os nós da camada subjacente. Quando nem todos os nós estiverem conectados entre si a rede é dita parcialmente conectada.

Os perceptrons de camada única são redes neurais compostas de um ou mais neurônios que captam os sinais de entrada e produzem eles próprios as saídas da rede, conforme figura 9. Essas RNA's são capazes de classificar padrões com classes linearmente separáveis.

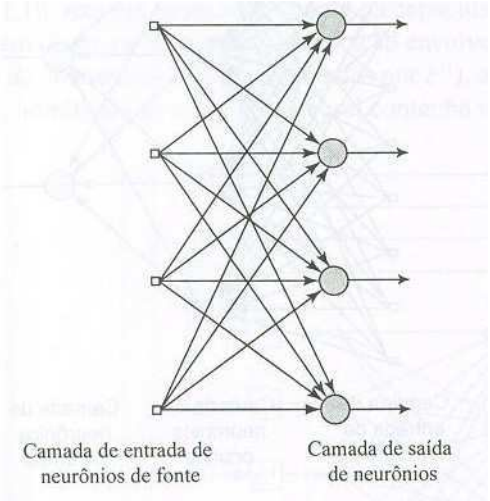

Figura 9: Uma RNA de camada única.(HAYKIN, 2001, p. 47)

As *Multi-Layer Perceptrons* - MLP[2] são redes de múltiplas camadas alimentadas adiante, normalmente acíclicas. Normalmente uma rede MLP é composta de unidades sensoriais que compõem a camada de entrada, uma ou mais camadas de neurônios ocultas e uma camada de neurônios de saída, conforme figura 10. Essas redes são capazes de aproximar funções complexas e não-lineares e classificar padrões não-linearmente separáveis, superando o grande obstáculo das redes de camada única. Contudo o treinamento dessas redes se torna substancialmente mais complexo.

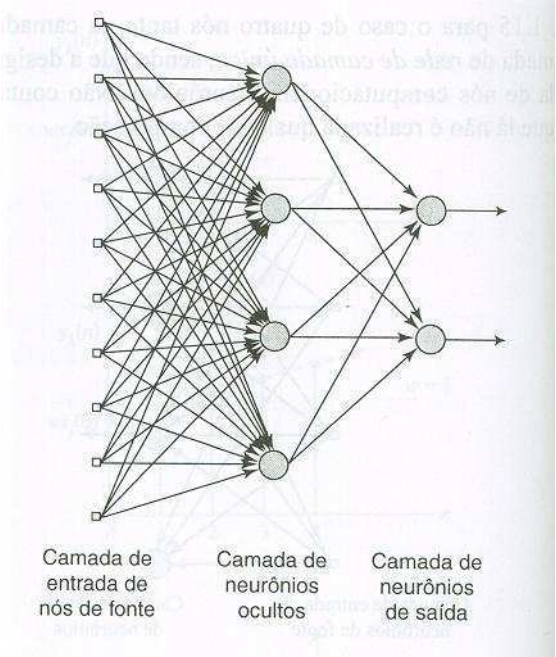

Figura 10: Uma RNA de múltiplas camadas.(HAYKIN, 2001, p. 48)

2.5 Métodos de Aprendizado

Os métodos e técnicas da Inteligência Artificail se destacam dos demais por agregarem conhecimento. Este conhecimento, segundo FISCHLER e FIRSCHEIN (apud HAYKIN, 2001), " se refere à informação armazenada ou a modelos utilizados por uma pessoa ou máquina para interpretar, prever e responder apropriadamente ao mundo exterior".

As RNA's, ao contrário de outras técnicas da Inteligência Artificial, armazenam informação e conhecimento de forma não compreensível ao ser humano. Enquadram-se, segundo REZENDE (2003, p. 92), nos "sistemas do tipo *caixa-preta* onde o sistema desenvolve internamente sua própria representação dos conceitos por ele tratados, não fornecendo um modelo lógico intuitivo ao ser humano". São opostos aos sistemas *orientados a conhecimento* como

[2]Perceptrons de Múltiplas Camadas

os Sistemas Especialistas, que possuem uma base de conhecimento cuja representação é inteligível.

A forma como o conhecimento é agregado ao sistema é conhecido como aprendizagem, desempenhada por um algoritmo de treinamento. Nas RNA's o algoritmo de aprendizado é responsável pela adaptação da rede de forma que, em um número finito de iterações do algoritmo a saída da rede converja para a saída esperada, ou seja, para o resultado desejado. Segundo MENDEL e McLAREM (apud HAYKIN, 2001), "Aprendizagem é um processo pelo qual os parâmetros livres de uma rede neural são adaptados através de um processo de estimulação pelo ambiente no qual a rede está inserida. O tipo de aprendizagem é determinado pela maneira pela qual a modificação dos parâmetros ocorre."

Durante o processo de aprendizado são gerados valores de incremento Δw que serão aplicados no vetor de pesos w. Esse incremento deve alterar o vetor de pesos de forma que ele produza, apartir dos dados de entrada, a solução desejada, de forma que $w_{t+1} = w_t + \Delta w$ esteja mais próxima da solução do que w_t, onde t significa o número da iteração do algoritmo.

Alguns problemas podem ocorrer durante o processo de aprendizado, tais como o sub-ajuste (*underfitting*) e o super-ajuste (*overfitting*). O *underfitting* ocorre quando a rede não converge, ou uma amostra muito pouco representativa é assimilada pela rede em detrimento de amostras mais representativas. O *overfitting* ocorre quando há a consideração excessiva de um ruído na amostra ou de simplesmente uma amostra anômala acarretando que o classificador acaba desviando o classificador de forma que este considere uma extensão maior que a ideal.

Das diversas técnicas de aprendizagem conhecidas, dois tipos são mais relevantes: o aprendizado supervisionado e o aprendizado não supervisionado. As demais técnicas, como o aprendizado competitivo e por esforço, são variações ou casos particulares de cada um destes tipos.

No aprendizado não supervisionado, a rede não dispõe de qualquer informação prévia sobre o domínio dos sinais de entrada da rede. Não há uma entidade externa à rede que forneça informações sobre o que está certo ou errado, a rede se adapta aos dados extraindo deles características relevantes.

Isso é possível devido à redundância dos dados de entrada que formam padrões com regularidade estatística. À rede cabe reconhecer essas regularidades estatísticas e adaptar seus pesos sinápticos para identificá-las e prevê-las. O processo de aprendizado nada mais é do que modificar continuamente os pesos sinápticos em resposta à entrada dos dados na rede, adaptando-os para poderem dividir os sinais de entrada em classes de acordo com suas características intrínsecas.

O aprendizado supervisionado é o foco deste trabalho. No aprendizado supervisionado, o conjunto de dados de treinamento da rede é formado por vetores contendo os dados de entrada e a saída desejada da rede. Existe a figura do supervisor que compara a saída atual da rede frente aos dados de entrada com a saída deseja e fornece um valor de erro. O supervisor calcula o valor do erro para cada padrão apresentado, e reajusta os pesos dos neurônios para minimizar o erro da rede. O erro é então uma função de custo a ser minimizada para que a rede converja para uma solução estável. O algoritmo tradicional para a metodologia de aprendizado supervisionado é o *Backpropagation*, que será discutido em detalhes na seção 2.6.

2.6 O Algoritmo *Backpropagation*

Conforme visto na seção 2.1, a saída de um neurônio pode ser definida como na equação 2.8:

$$y_{jk} = \varphi\left(\sum_{i=1}^{n} w_{ji}x_{ki} + b_j\right) = \varphi(net_{jk}) \tag{2.8}$$

Onde:

- j é índice do neurônio na rede;
- k é índice do padrão;
- i é o índice de entrada x no padrão k, e do peso w correspondente no *perceptron* j.
- n é número de entradas do padrão k;
- w_j é um peso do neurônio j;
- x_k é uma entrada do padrão k
- b_j é o *bias* do neurônio j;
- net_j é o combinador linear do neurônio j ;
- φ é a função de ativação;
- y_{jk} é a saída do neurônio j para o padrão k.

Ao conjunto de entrada $X = \{x_1, x_2, x_3, \ldots, x_k, \ldots, x_n\}$, associa-se o conjunto $D = \{d_1, d_2, d_3, \ldots, d_k, \ldots, d_n\}$ com as saídas desejadas da rede. Após serem apresentados os

padrões à RNA, suas saídas são comparadas aos valores de d_k. O erro e_k é calculado para cada padrão k:

$$e_k = d_k - y_k \tag{2.9}$$

Para corrigir o erro é utilizado o algoritmo *Backpropagation* que propõe uma forma de definir o erro dos neurônios nas camadas intermediárias, possibilitando assim o ajuste dos pesos. Na definição de HAYKIN (2001, p. 183), o algoritmo de *backpropagation* é uma regra matemática de ajuste de erros:

> "Basicamente, a aprendizagem por retropropagação do erro consiste de dois passos através das diferentes camadas da rede: um passo para frente, a *propagação*, e um passo para trás, a *retropropagação*. No passo para frente, um padrão de atividade (vetor de entrada) é aplicado aos nós sensoriais da rede e seu efeito se propaga através da rede, camada por camada. Finalmente, um conjunto de saídas é produzido como a resposta real da rede. Durante o passo de propagação, os pesos sinápticos da rede são todos fixos. Durante o passo para trás, por outro lado, os pesos sinápticos são todos ajustados de acordo com uma regra de correção de erro. Especificamente, a resposta real da rede é subtraida de uma resposta desejada (alvo) para produzir um sinal de erro. Este sinal de erro é então propagado para trás através da rede, contra a direção das conexões sinápticas - vindo daí o nome de "retropropagação de erro" (*error back-propagation*)"

Na fase de propagação, ou *forward*, não existe alteração dos pesos sinápticos dos neurônios, somente na fase de retro-propagação, a fase *backward*. Ocorre então uma iteração das fases forward e backward para cada padrão apresentado à rede, até um determinado limite de iteração ou até que a rede atinja um critério determinidado de parada.

Os critérios de parada do algoritmo podem ser diversos, como uma taxa mínima de erro ou o número máximo de iterações. Os valores desses critérios têm sido estipulados empiricamente, sendo dependentes do tamanho do vetor de entradas e da complexidade da função a ser aproximada pela rede.

O algoritmo *Backpropagation* pode ser definido como na listagem abaixo:

1. Inicializar pesos;
2. Repita enquanto a condição de parada for falsa;
3. Para cada par de treinamento:

   Fase Forward

4. Cada unidade de entrada recebe um sinal de entrada x_i e difunde este sinal para todas as unidades na camada acima (nas camadas escondidas);

5. Cada unidade oculta k computa o $net_k = \sum_{i=1}^{n} w_{ki}x_i + b_k$ e aplica esse valor a função $y = \varphi(net_k)$ para gerar o sinal de saída y;

6. Cada unidade de saída computa o net e aplica esse valor a função φ para gerar o sinal de saída y;

   Fase Backward

7. Cada unidade de saída recebe um valor esperado correspondente ao valor de entrada e calcula o erro:

   $\delta_k = (d_k - y_k)\varphi'(net_k)$

   E calcula o valor de correção dos pesos:

   $\Delta w_k = \eta\delta_k x_i$

   E calcula o valor de correção do *bias*:

   $\Delta b = \eta\delta_k$

8. Cada unidade oculta soma as δ recebidos das camadas anteriores:

   $\delta_{in} = \sum_{k=1}^{m} \delta_k w_{kj}$

   Multiplica esse valor pela derivada da função de ativação:

   $\delta_k = \delta_{in}\varphi'(net)$

   Calcula o valor de valor de correção dos pesos:

   $\Delta w_k = \eta\delta_k x_k$

   E calcula o valor de correção do *bias*:

   $\Delta b = \eta\delta_k$

   Correção dos Pesos

9. Cada unidade corrige o valor dos pesos:

   $w_t = w_{t-1} + \Delta w$

   E o valor do bias:

   $b = b + \Delta b$

10. Testa a condição de parada

O algoritmo de treinamento *Backpropagation* promove a atualização dos pessos por correção de erros, conforme visto na seção 2.5:

$$w_{t+1}^{ij} = w_t^{ij} + \Delta w \tag{2.10}$$

Onde:

- t é o índice da iteração do algoritmo.
- j é o índice do neurônio na rede;
- i é o índice do peso w do neurônio j.

O peso w_i do neurônio j na iteração $t+1$ é igual ào peso atual mais o incremento Δw. O cálculo do valor Δw para todos os pesos de todos os neurônios da rede é o núcleo do algoritmo *Backpropagation*.

O algoritmo de treinamento busca minimizar o erro das saídas em relação aos valores desejados do conjunto de treinamento. A função de custo a ser minimizada é:

$$E = \frac{1}{2}\sum_{i=1}^{n}(d^i - y^i)^2 \tag{2.11}$$

Ao minimizar essa função busca-se obter a direção de ajuste para o valor Δw, através do cálculo do gradiente descendente da função E no peso w_t. O gradiente de uma função está na direção e sentido em que a função tem taxa de variação máxima. Valor Δw é na direção oposta do vetor gradiente ∇E.

$$\Delta w = -\frac{\partial E}{\partial w_t}E \tag{2.12}$$

Pela regra da cadeia, o gradiente pode ser calculado como:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}} \tag{2.13}$$

A derivação das fórmulas depende de onde o neurônio se encontra na rede neural. Para a camada de saída:

$$\delta_j = (d_j - y_j)\varphi'(net_j) \tag{2.14}$$

Onde:

- j é o índice do neurônio;
- δ_j é o valor do gradiente.

Se o neurônio se encontra nas camadas intermediárias:

$$\delta_j = \varphi'(net_j) \sum \partial w_t \tag{2.15}$$

$$\Delta w = \eta \varphi'(net_{kj}) x_i \tag{2.16}$$

Onde:

- η é a taxa de aprendizado;
- t é a iteração, também conhecida como época.

O parâmetro η é reponsável por acelerar ou retardar o treinamento. Para que possa ser calculada, a função φ deve ser contínua e derivável.

$$w_{t+1}^{ij} = w_t^{ij} + \eta \varphi'(net_{kj}) x_i \tag{2.17}$$

Na literatura encontram-se diversas implementações desse algoritmo, algumas delas acrescidas de pequenas variações e otimizações desenvolvidas no decorrer do tempo. Entre elas destacam-se os modos de treinamento do algoritmo que influenciam a freqüencia de cálculo do erro e ajuste dos pesos.

No modo *On-line* os pesos são atualizados após a apresentação de cada padrão, conforme figura 11.

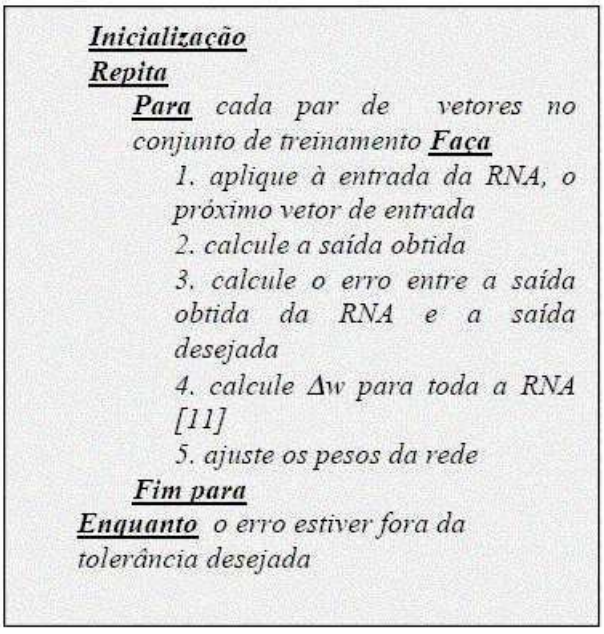

Figura 11: Algoritmo *Backpropagation* no modo *On-line*(PIMENTEL; FIALLOS, 1999)

No modo *Batch* os pesos são atualizados após a apresentação de todos os padrão, conforme figura 12. Esse método acelera o treinamento dado que a atualização é realizada apenas uma vez a cada N iterações, sendo N o número de padrões.

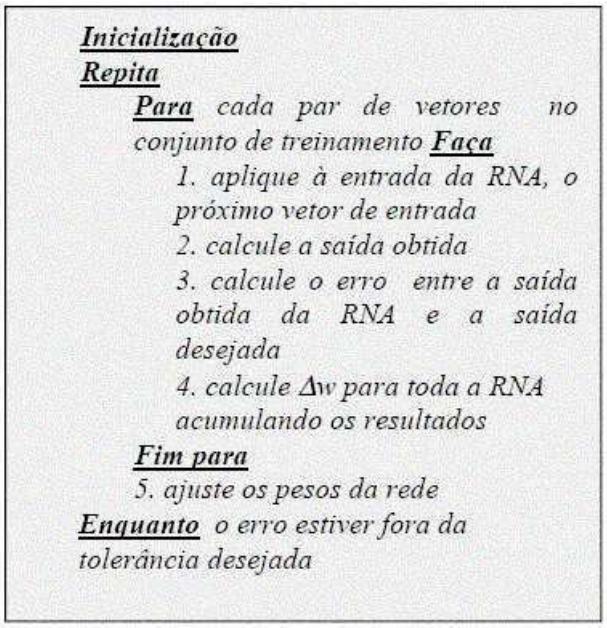

Figura 12: Algoritmo *Backpropagation* no modo *Batch*(PIMENTEL; FIALLOS, 1999)

No modo *On-line* os pesos são atualizados após a apresentação de uma certa quantidade de padrões, conforme figura 13. Este modo é um meio-termo entre o modo on-line e o modo batch e a a atualização dos pesos será realizada uma vez em cada bloco de M amostras ou pares do conjunto de treinamento, sendo $1 < M < N$.

2.7 Considerações sobre o projeto de RNA

Segundo SILVA e OLIVEIRA (2001):

> "a utilização de um grande número de camadas escondidas não é recomendado. Cada vez que o erro médio durante o treinamento é utilizado para atualizar os pesos das sinapses da camada imediatamente anterior, ele se torna menos útil ou preciso. A ú nica camada que tem uma noção precisa do erro cometido pela rede é a camada de saída. A última camada escondida recebe uma estimativa sobre o erro. A penúltima camada escondida recebe uma estimativa da estimativa, e assim por diante. Testes empíricos com a rede neural MLP *backpropagation* não demonstram vantagem significante no uso de duas camadas escondidas ao invés de uma para problemas menores. Por isso, para a grande maioria dos problemas utiliza-se apenas uma camada escondida quando muito duas e não mais que isso.)"

O número de neurônios nas camadas escondidas, este é geralmente definido empiricamente. Deve-se ter cuidado para não utilizar nem unidades demais, o que pode levar a rede a

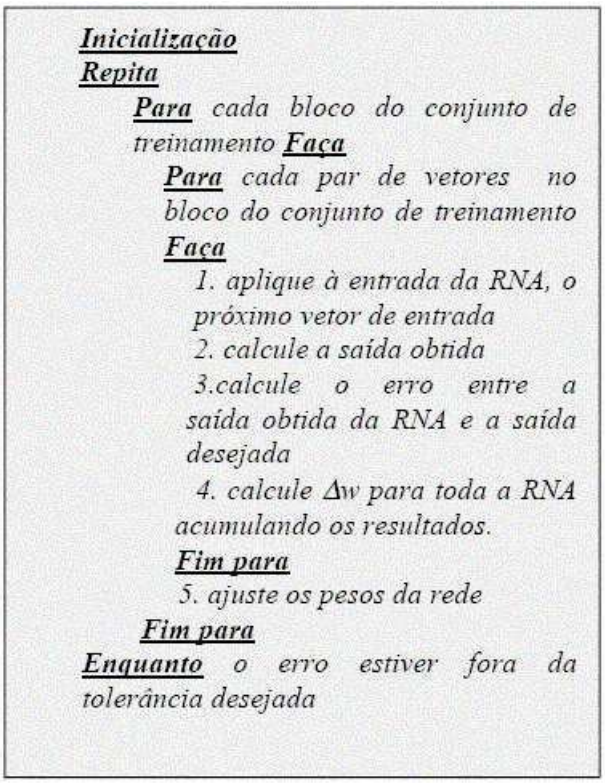

Figura 13: Algoritmo *Backpropagation* no modo *Block* (PIMENTEL; FIALLOS, 1999)

memorizar os dados de treinamento (*overfitting*), ao invés de extrair as características gerais que permitirão a generalização, nem um número muito pequeno, que pode forçar a rede a gastar tempo em excesso tentando encontrar uma representação ótima. Devido a estas dificuldades é recomendado manter o número de neurônios escondidos baixo, mas não tão baixo quanto o estritamente necessário. Existem várias propostas de como determinar a quantidade adequada de neurônios nas camadas escondidas de uma rede neural. As mais utilizadas são:

- definir o número de neurônios em função da dimensão das camadas de entrada e saída da rede. Pode-se definir o número de neurônios na camada escondida como sendo a média aritmética ou ainda como sendo a média geométrica entre tamanho da entrada e da saída da rede;

- utilizar um número de sinapses dez vezes menor que o número de exemplos disponíveis para treinamento. Se o número de exemplos for muito maior que o número de sinapses, *overfitting* é improvável,

3 Programação Paralela

3.1 Introdução

Embora os processadores estejam cada dia mais robustos, tanto em termos de frequência de clock quanto do tamanho da palavra de memória, as necessidades computacionais aumentam em um ritmo ainda mais acelerado. E, apesar do contínuo aumento das frequências de clock, as limitações físicas dos circuitos eletrônicos não permitirão um aumento infinito do clock. Problemas físicos como a dissipação de calor e o tamanho cada vez menor dos transístores tem ocupado a mente dos projetistas de circuitos eletrônicos e engenheiros da computação, como mostra TANENBAUM (2003).

Os sistemas de alto desempenho se distinguem, primeiramente, por suas finalidades. Os HPC - *High Performance Computers*[1] sao sistemas computacionais cuja principal função é prover um ambiente com elevado poder de processamento. Encontramos freqüentemente esses sistemas em ambientes de pesquisa científica, engenharia e em outros mais diversos, como no segmento da computação gráfica.

Os HAC - *High Available Computers*[2] sao sistemas onde a disponibilidade e a ininterruptibilidade dos serviços são as metas principais. Essa categoria de *clusters* é muito ensejada pela indústria e provedores de serviços, em especial provedores de acesso e conteúdo na internet, onde não é desejável que o sistema fique indisponível aos usuários.

Em ambos os casos têm sido adotados computadores paralelos, explorando recursos computacionais das CPU em conjunto. Este paralelismo pode ser de vários níveis, das instruções ao software. As máquinas paralelas organizam-se segundo a natureza, tamanho e quantidade dos seus elementos de processamento e memória e como estes estão interligados.

Conforme TANENBAUM (2003), os elementos de processamento podem variar de UALs[3] muito simples até processadores completos, ambos interconectados e operando em conjunto. O projeto do computador paralelo determinará a quantidade desses elementos de processamento e

[1] Computadores de Alta Performance

[2] Computadores de Alta Disponibilidade

[3] UAL - Unidade Lógico-Aritmética, elemento da CPU utilizado para fazer operações matemáticas inteiras.

a comunicação entre eles em função das suas limitações intrínsecas. Os elementos de memória são dividos em módulos independentes e em paralelo, o que permite que vários processadores tenham acesso simultâneo à memória. A interligação entre os elementos de processamento e os elementos de memória é o maior diferenciador entre as diversas tecnologias de máquinas paralelas. Segundo TANENBAUM (2003, p. 316):

> "A grosso modo, os esquemas de interconexão podem ser divididos em duas categorias: estáticos e dinâmicos. Os esquemas estáticos simplesmente ligam os componentes de um sistema paralelo de maneira fixa, como, por exemplo em estrela, em anel ou em grade. Nos esquemas de interconexão dinâmicos, todas as partes componentes do sistema estão ligadas a elementos comutadores, que podem rotear mensagens dinamicamente entre eles. Cada um desses esquemas tem seus pontos fortes e seus pontos fracos."

Sob essa perspectiva temos os multiprocessadores e os multicomputadores.Em TANENBAUM (2003, p. 396), encontra-se um comparativo entre multiprocessadores e multicomputadores:

> "Multiprocessadores são populares e atrativos porque oferecem um modelo de comunicação simples: todas as CPUs compartilham uma memória comum. Os processos podem escrever mensagens na memória, a qual pode depois ser lida por outros processos. A sincronização é possível mediante o emprego de mutex, semáforos monitores e outras técnicas bem definidas. A única desvantagem é que os multiprocessadores de grande porte são difíceis de construir e, portanto, são caros. Para solucionar este problema, muita pesquisa tem sido feita com multicomputadores, que são CPUs fortemente acopladas que não compartilham memória. Cada CPU tem sua própria memória local ... Esses sistemas são também conhecidos por uma variedade de outros nomes, como computadores *cluster* e COWS(*clusters of workstations* - *clusters* de estações de trabalho)"

Neste capítulo serão discutidas as arquiteturas de máquinas paralelas e suas características, dando ênfase aos *clusters Beowulf* e a biblioteca de programação paralela MPI.

3.2 Granularidade

Um problema a ser tratado na análise de uma solução paralela, seja de *software* ou *hardware*, é a granularidade ou tamanho do grão. Segundo TANENBAUM (2001) granularidade se expressa pela relação entre a quantidade de processamento e a quantidade de comunicação entre os nós necessária por uma tarefa. Quando a quantidade de processamento é maior do que a comunicação necessária entre os nós, diz-se Granularidade Grossa. Quando a comunicação entre os nós é maior do que a necessidade de processamento diz-se Granularidade Fina.

Alguns computadores paralelos são concebidos para rodar, simultaneamente, várias tarefas independentes. Essas tarefas nada tem haver umas com as outras e, portanto não precisam se comunicar. De outro, lado há projetos onde o foco é rodar uma única tarefa dividida entre diversos processos que executam simultaneamente. No *software* é preciso analizar o quanto as partes de um mesmo processo, distribuidas entre vários elementos de processamento vão se comunicar através da memória. Em sistemas *CPU Bounded*[4], o fluxo de dados entre esses elementos deve ser minimizado, mimizando também o tempo gasto com a transferência de dados para maximizar o tempo dedicado ao processamento.

No *hardware*, deve haver um canal de comunicação entre as CPUs e a memória com velocidade e largura de banda que permita altas taxas de transferência de dados. Tarefas independentes com pouca ou nenhuma comunicação entre si, não necessitariam de canais dedicados de comunicação.Os sistemas compostos de uma pequena quantidade de processadores grandes, independentes e com conexões de baixa velocidade estabelecidas entre si são chamados de sistemas fracamente acoplados. Os sistemas fortemente acoplados são compostos de processadores de pouca potência computacional, fisicamente próximos uns dos outros e que integram freqüentemente por meio de redes de comunicação de alta velocidade.

3.3 Taxonomias

3.3.1 Classificação de Flynn

Segundo FLYNN (apud JORDAN; ALAGHBAND, 2002, p. 2), as arquiteturas de computadores podem ser organizada segundo a combinação do fluxo de instruções (*Instruction Stream*) e do fluxo de dados (*Data Stream*). O fluxo de instruções seria divido em *Single Instruction* e *Multiple Instrution* e o fluxo de dados em *Single Data* e *Multiple Data*, conforme segue:

		Fluxo de Instruções (I)	
		Serial ou Único (**SI**)	Paralelo ou Múltiplo (**MI**)
Fluxo de Dados (D)	Serial ou Único (**SD**)	**SISD**	**MISD**
	Paralelo ou Múltiplo (**MD**)	**SIMD**	**MIMD**

Figura 14: Classificação de Flynn (MENDONÇA; ZELENOVSKY, 2005)

Na classe SISD - *Single Instruction Single Data*, um único fluxo de instruções opera sobre um único fluxo de dados. Isto corresponde ao processamento seqüencial característico das máquinas convencionais. Apesar dos programas estarem organizados através de instruções seqüenciais, elas podem ser executadas de forma sobreposta em diferentes estágios (*pipe-*

[4]Sistemas focados no processamento, que utilizam pouco os recursos de Entrada/Saida e periféricos

lining). Arquiteturas SISD caracterizam-se por possuírem uma única unidade de controle podendo possuir mais de uma unidade funcional.

Na classe SIMD - *Single Instruction Multiple Data*, o processamento de vários dados ocorre sob o comando de uma única instrução. Em uma arquitetura SIMD o programa ainda segue uma organização seqüencial com múltiplos dados. É preciso, nesse caso, uma organização de memória em diversos módulos com acessos independentes. A unidade de controle é única mas existem diversas unidades funcionais. Aqui enquadram-se os processadores vetoriais e matriciais.

Na classe MISD - *Multiple Instruction Single Data*, existem múltiplas unidades de controle executando instruções distintas que operam sobre um mesmo dado. Não existem implementações para esta classe, que segundo TANENBAUM (2003) considerada tecnologicamente inviável.

Por fim, a classe MIMD - *Multiple Instruction Multiple Data*, considera múltiplos fluxos de instruções simultâneos sobre múltiplos fluxos de dados. Nessa classe, várias unidades de controle comandam suas unidades funcionais, as quais tem acesso a vários módulos de memória. Qualquer grupo de máquinas operando como uma unidade (deve haver um certo grau de interação entre as máquinas) enquadra-se como MIMD. Alguns representantes desta categoria são os servidores multiprocessados, as redes de estações e as arquiteturas massivamente paralelas.

A classificação de Flynn não contempla a forma de acesso aos dados e a sincronização, essenciais na discussão de novas plataformas paralelas. Novas classificações surgiram nas décadas de 80 e 90, mas a mais importante é a classificação de Duncan, discutida na próxima sessão.

3.3.2 Classificação de Duncan

A classificação de DUNCAN (apud SENGER, 1997) surgiu como contra-proposta à classificação de FLYNN (1972), sugerindo uma classificação mais flexível para as arquiteturas mais recentes. Essa classificação divide as arquiteturas em dois grupos principais: arquiteturas síncronas e assíncronas, e mantem os elementos da classificação de Flynn, no que diz respeito ao fluxo de dados e instruções.

As Arquiteturas Síncronas caracterizam-se por coordenarem as operações concorrentes através da utilização de sinais de relógio globais, unidades de controle centralizadas ou unidades de controle vetoriais.

As Arquiteturas Assíncronas caracterizam-se pelo controle descentralizado de *hardware*, sendo que cada elemento de processamento executa diferentes instruções sobre diferentes

dados. Essa categoria é formada pelas máquinas MIMD, sejam elas convencionais ou não.

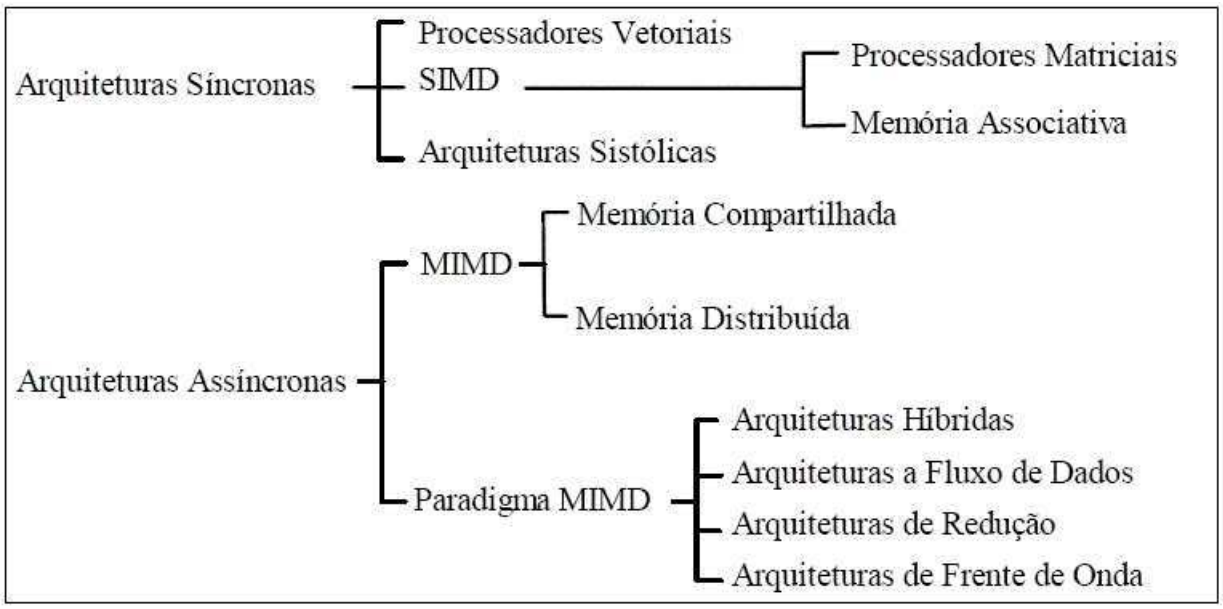

Figura 15: Classificação de Duncan (SENGER, 1997)

Os computadores multiprocessadores se enquadram nas categorias síncronas, cujo acesso à memória pode ser compatilhado (*Shared Memory*) ou exclusivo, no caso das arquiteturas UMA[5] e NUMA[6].

Os multicomputadores e *clusters* se enquadram nas categorias assíncronas, que se comunicam através de bibliotecas de troca de mensagens ou por memória compartilhada distribuída (*Distributed Shared Memory*). Os multicomputadores são o foco deste trabalho, em especial a arquitetura *Beowulf*, que serão discutidos em detalhes na próxima seção.

3.4 Multicomputadores e Clusters

Os multicomputadores *cluster* são, na definição de STALLINGS (2002, p. 671):

[5]Uniform Memory Access - Acesso de memória uniforme
[6]Non Uniform Memory Access - Acesso de memória não uniforme

> "Uma das áreas mais novas e promissoras de projeto de sistemas de computação é a de agregados ou algomerados de computadores ou, simplesmente, clusters. A organização de *clusters* constitui uma alternativa para os multiprocessadores simétricos (SMP), como abordagem para prover alto desempenho e alta disponibilidade, e é particularmente atrativa para aplicações baseadas em servidores. Podemos definir um *cluster* como um grupo de computadores completos interconectados, trabalhando juntos, como um recurso de computação unificado, que cria a ilusão de constituir uma única máquina. O termo computador completo significa um sistema que pode operar por si próprio, independentemente do *cluster*; na literatura, cada computador componente do *cluster* é usualmente denominado um nó."

A partir dessa definição, os *clusters Beowulf* são multicomputadores interligados por alguma tecnologia de rede local comercial, como Ethernet ou FDDI, e que utilizam um dos diversos padrões de troca de mensagens, como PVM ou MPI, para distribuir tarefas entre as unidades de processamento. Os *cluster Beowulf* nasceram no centro de pesquisas CESDIS, localizado no Goddard Space Flight Center da NASA. O primeiro *cluster* possuia 16 nós de processamento, interligados pelo padrão IEEE 802.3 (Fast Ethernet)[7], e foi utilizado para tratar dados recolhidos de satélites e outros problemas de ciências espaciais.

Atualmente a tecnologia se expandiu e popularizou, existindo *clusters* com algumas dezenas até milhares de nós. Um estudo sistemático sobre *clusters Beowulf* pode ser encontrada na própria página do projeto, em (BEOWULF, 2005).

Nos *clusters Beowulf* existe o papel da CPU mestre e das CPU escravos. A CPU mestre comanda todo o *cluster*, balanceando a carga de trabalho das CPU escravas e coordenando a comunicação. Geralmente, as CPU escravas não possuem periféricos tais como mouse e monitor pois seu acesso se dá somente através da CPU mestre. Nesse aspecto é importante diferenciar os *clusters Beowulf* dos COW, onde todos as CPU podem ser utilizadas diretamente.

Os *clusters* são computadores da classe SIMD, muitas vezes referenciados como SPMD - *Single Program Multiple Data*[8]. Sob essa ótica, nota-se que o mesmo código estará rodando em todos os nós do *cluster*mas processando dados diferentes. Todo essa trabalho é coordenado pelas primitivas de comunicação e sincronização das bibliotecas de programação paralela.

3.5 Bibliotecas de Programação Paralela

As bibliotecas de programação paralela são padrões de troca de mensagens entre os nós do *cluster*, que implementam primitivas de comunicação e sincronização. Há vários padrões, mas

[7] O padrão IEEE 802.3, também conhecido com Fast Ethernet define o protocolo de enlace CSMA/CD a 100 Mbps.

[8] Único programa, múltiplos dados

todas elas têm em comum uma biblioteca com funções e métodos de interface e um serviço que executa no sistema operacional coordenando as tarefas do *cluster*, recebendo e enviando dados entre os nós. A funções e métodos são um *front end*[9] para o processo servidor.

As bibliotecas não se restringem a uma linguagem de programação específica, mas as implementações, geralmente, são feitas em C, C++ e Fortran. Estas linguagens são tradicionalmente utilizadas em matemática computacional e aplicações de elevado processamento devido ao seu desempenho.

As bibliotecas, no entanto, oferecem apenas a possibilidade dos nós se comunicarem e se sincronizarem. Segundo SCHLEMER (2002), o fator preponderante no desempenho dos processos paralelos é a forma como os algoritmos são implementados:

> "A programação em ambientes paralelos é o ponto chave do ganho de desempenho. Alguns algoritmos, dada a necessidade de explicitar a comunicação entre os processadores, tendem a ser bastante complexos. O desenvolvimento de ambientes de programação que amenizem esta complexidade se torna crucial. Ao utilizar um computador paralelo, o que se deseja é um ambiente paralelo, onde o paralelismo seja auto-maticamente explorado. Assim, extensões de linguagens ou novas construções devem ser desenvolvidas para especificar o paralelismo ou facilitar sua detecção em vários níveis de glanuralidade através de compiladores mais inteligentes."

Nem todo algortimo seqüencial pode ser paralelizado ou totalmente paralelizado. As partes de um algoritmo tem ligações lógicas e relações de ordem de processamento que devem ser respeitadas, conforme figura 16.

Quando uma dada parte do algoritmo depende de outra, diz-se que há dependência de controle. Os dados trabalhados no algoritmo também guardam entre si relações de dependência, e se um dado depende do prévio processamento de outro, diz-se que há dependência de dados. Os pontos onde existem dependências de controle e dados devem ser seqüenciais, sendo que instruções e dados independentes podem ser tratados paralelamente. Se diversas partes do algoritmo pretendem trabalhar com o mesmo dado ao mesmo tempo deve-se então estabelecer primitivas de sincronização para evitar *deadlocks*[10]. Essas partes do algoritmo são conhecidas como regiões concorrentes.

Para auxiliar os programadores na observância desses preceitos e facilitar as tarefas de comunicação e sincronização no *cluster*foram desenvolvidas inúmeras bibliotecas de programação paralela que serão discutidas adiante.

[9] Funções de interface
[10] Travamento por dependência mútua

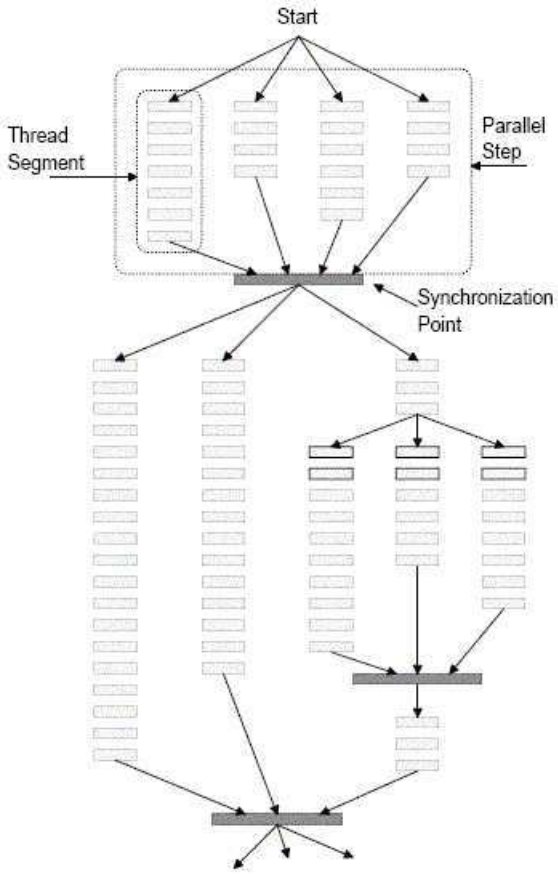

Figura 16: Fluxo de execução em um ambiente paralelo (DASGUPTA; KEDEM; RABIN, 1995)

3.5.1 PVM

Segundo SENGER (1997), o PVM (*Parallel Virtual Machine*) é um conjunto integrado de ferramentas de *softwaree* bibliotecas que emulam um sistema computacional concorrente heterogêneo, flexível e de propósito geral. O projeto PVM foi iniciado em 1989, no *Oak Ridge National Laboratory*. A primeira versão (PVM 1.0), foi desenvolvida por *Vaidy Sunderam e Al Geist*, sendo utilizada apenas pelo laboratório e não disponibilizada para outras instituições. A segunda versão (PVM 2.0), contou com o auxílio da *Universidade do Tennessee* no desenvolvimento e uma atualização em Março de 1991, ano em que o PVM começou a ser utilizado em aplicações científicas. Após a verificação de alguns problemas, o código foi completamente reescrito, gerando a terceira versão do sistema PVM (PVM 3.0), que começou a ser distribuído como um *software* de domínio público, fato que contribuiu significativamente para a sua divulgação e difusão. A partir daí, várias atualizações foram feitas, sendo que a versão mais recente é a PVM 3.4. Os princípios em que o PVM é baseado são os seguintes:

- Coleção de máquinas (*host pool*) configurada pelo usuário: as aplicações são executadas em um conjunto de máquinas selecionadas de maneira dinâmica pelo usuário;
- Transparência de acesso ao *hardware*: a aplicação enxerga o *hardware* como uma coleção de elementos de processamento virtuais, sendo possível a atribuição de tarefas para as arquiteturas mais apropriadas;
- Computação baseada em processos: a unidade de paralelismo do PVM é uma tarefa que alterna sua execução seqüencial entre computação e comunicação, sendo possível a execução de mais de uma tarefa em um elemento de processamento virtual;
- Passagem de mensagens: a coleção de tarefas que estão sendo executadas cooperam entre si, enviando e recebendo mensagens entre elas,sendo que o tamanho dessas mensagens é limitado apenas pelos recursos do sistema (memória disponível);
- Suporte a ambientes heterogêneos: o sistema PVM dá suporte à heterogeneidade em nível de arquiteturas de computadores, redes de comunicação e aplicações. Mensagens de máquinas com diferentes representações de dados podem ser trocadas e corretamente interpretadas.

O sistema PVM consiste basicamente em duas partes. A primeira parte é o serviço PVM (pvmd3 ou pvmd), que reside em todas as máquinas que fazem parte da máquina virtual. Quando o usuário necessita executar uma aplicação utilizando o PVM, ele precisa iniciar o processo PVM, através da linha de comando ou da aplicação, nas máquinas que serão utilizadas. Vários usuários podem configurar suas máquinas virtuais próprias, sem que uma interfira na máquina virtual de outro usuário A segunda parte consiste na biblioteca de comunicação

PVM (Libpvm), que deve ser encadeada (linked) com as aplicações que são desenvolvidas. Essa biblioteca disponibiliza as rotinas para comunicação, gerenciamento dinâmico e sincronização entre processos

3.5.2 MPI

O padrão MPI - *Message Passing Interface*[11] visa prover a comunicação inter processos[12] através de envio e recebimento de mensagens. É uma interface simples e extremamente poderosa de comunicação entre múltiplos processadores, que pode se comunicar um-a-um ou em grupo, utilizando primitivas de comunicação coletiva. O padrão foi gerido através de um fórum aberto entre empresas, instituições de pesquisa e programadores de todo o mundo, que pode ser acessado no endereço http://www.mpi-forum.org/.

O primeiro documento do padrão MPI foi apresentado em 1994 (versão 1.0) e atualizado em 1995 (versão 1.1), e está disponível em http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html. O documento *"MPI: A Message-Passing Standard"* que define o padrão MPI foi publicado pela *Tenessee College* e nele nos apoiamos para implementações em MPI. Outros aprofundamentos e comentários sobre esta tecnologia, bem como a versão mais recente do padrão (versão 2.0) disponivel em http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html, estão disponíveis livremente na internet, onde o padrão surgiu.

3.6 A biblioteca MPI

O principal objetivo do MPI é disponibilizar uma interface que seja largamente utilizada no desenvolvimento de programas que utilizem troca de mensagens. Além de garantir a portabilidade dos programas paralelos, essa interface deve ser implementada eficientemente nos diversos tipos de máquinas paralelas existentes (reais ou *clusters* de *workstations*).

O MPI é uma biblioteca com funções para troca de mensagens, responsável pela comunicação e sincronização de processos. Dessa forma, os processos de um programa paralelo podem ser escritos em uma linguagem de programação seqüencial, tal como C ou Fortran. Os nós componentes do *cluster*são organizados em grupos e identificados para que seja possível a comunicação e sincronização.

Ao iniciar o *cluster*todo nó pertencente a ele tem uma identificação, chamada *rank*, atribuída pelo sistema. Essa identificação é exclusiva e contínua, começando do 0 até $n-1$ computadores.

[11] Interface de Troca de Mensagens

[12] IPC - Inter Process Comunication, Comunicação entre processos geralmente é um dos serviços oferecidos pelo Sistema Operacional.

O *Communicator* define uma coleção de nós, que poderão se comunicar entre si, estabelecendo um contexto de comunicação. O MPI utiliza essa combinação de grupo e contexto para garantir uma comunicação segura e evitar problemas no envio de mensagens entre os processos.

O Grupo (*Group*) é um conjunto ordenado de N nós. Todo e qualquer grupo é associado a um *communicator* e, inicialmente, todos os processos são membros de um grupo com um *communicator* já pré-estabelecido (MPI_COMM_WORLD).

Além da identificação dos nós pertencentes ao *cluster*, o MPI empacota os dados que tramitam no *cluster* de forma a garantir que os dados certos cheguem nos nós corretos e que os dados transmitidos sejam de um tipo e tamanho conhecido pelo sistema. Para isso cada mensagem MPI contém duas partes: dado, na qual se deseja enviar ou receber, e o envelope com informações da rota dos dados.

O dado contém o endereço onde o dado se localiza, o número de elementos do dado na mensagem e o tipo do dado a ser transmitido. O envelope contém a identificação do processo que envia ou do processo que recebe, o rótulo (*tag*) da mensagem e o *communicator* do processo. Os *tags* são valores numéricos que funcionam como etiquetas e são usados para diversos fins, à critério do programador, como por exemplo identificar uma determinada fase do algoritmo ou um determinada transmissão de dados.

A biblioteca se organiza em três grupos principais de funções: Gerência de processos (inicializar, finalizar, determinar o número de processos, identificar processos), Comunicação Ponto à Ponto (Enviar e receber mensagens entre dois processos) e Comunicação Coletiva (*"broadcast"*, sincronizar processos). Existem dezenas de funções e variações de funções, mas um programa em MPI utiliza geralmente seis funções básicas conforme quadro 1:

Primitiva	**Função**
MPI_Init	Inicializa um processo MPI
MPI_COMM_RANK	Identifica um processo dentro de um determinado grupo.
MPI_COMM_SIZE	Retorna o numero de processos dentro de um grupo.
MPI_Send	Rotina basica para envio de mensagens no MPI.
MPI_Recv	Rotina basica para recepcao de mensagens no MPI.
MPI_Finalize	Finaliza um processo MPI

Quadro 1: Funções Básicas MPI

No entanto, as comunicações variam de acordo com a forma de tratar os dados na memória para envio e recebimento e seu modo de tratar os bloqueios de confirmação de recebimento ou sucesso. O *Aplication Buffer* é um endereço normal de memória aonde se armazena um dado que o processo necessita enviar ou receber. O *System Buffer* é um endereço de memória reservado pelo sistema para armazenar mensagens enviadas e recebidas. Dessa forma, dependendo do tipo de operação de *send/receive*, o dado no *aplication buffer* pode necessitar ser

copiado de ou para o *system buffer*. Neste caso teremos comunicação assíncrona, com o processo continuando seu fluxo normal de operação independentemente do sucesso ou fracasso do envio dos dados. Na comunicação síncrona o dado é enviado diretamento do *aplication buffer* e o programa somente retorna após a rotina de envio ser completada com seu recebimento por outro nó.

A comunicação síncrona ou assíncrona dependem dos *buffers* de memória e sua liberação pelas funções de envio e recebimento. Já os bloqueios dependem de eventos, ou seja, das *flags* enviadas pelo MPI sinalizando o status da tarefa. Uma rotina de comunicação é dita *bloking*, se a finalização da chamada depender de certos eventos, como a confirmação de recebimento ou um sinal de sucesso na operação. Comunicações *Non-Blocking* retornam sem esperar qualquer evento que indique o fim ou o sucesso da rotina.

As rotinas de comunicação Ponto a Ponto (*Peer to Peer*) se baseiam em duas primitivas básicas, Send para enviar dados a outro nó e Recv para receber dados advindos de outro nó. Com elas torna-se possível que dois, e apenas dois, nós se comuniquem diretamente e troquem dados específicos entre si.

Primitiva	**Função**
MPI_Send	Rotina basica para envio de mensagens no MPI (Standard Send).
MPI_Ssend	Blocking Synchronous Send.
MPI_Rsend	Blocking Ready Send.
MPI_Bsend	Blocking Buffered Send.
MPI_Isend	Non-Blocking Standard Send.
MPI_Issend	Non-Blocking Synchronous Send.
MPI_Irsend	Non-Blocking Ready Send.
MPI_Ibsend	Non-Blocking Buffered Send.
MPI_Recv	Rotina basica para recepcao de mensagens no MPI (Standard Recv).
MPI_Irecv	Non-Blocking Standard Recv

Quadro 2: Funções MPI de Comunicação Ponto a Ponto

As rotinas de comunicação Coletiva implementam um conjunto de funções que permitem que todos os nós do *cluster*possam trocar dados simultanemente, enviando e recebendo de todos os nós. Essas operações agilizam a troca de dados e facilitam o trabalho de programação, evitando inúmeras repetições de funções Ponto a Ponto.

Primitiva	**Função**
MPI_Bcast	Difusão de Dados
MPI_Gather	Concentração
MPI_Scatter	Espalhamento
MPI_Reduce	Redução

Quadro 3: Funções MPI de Comunicação Coletiva

3.6.1 A implementações MPI

Por ser um padrão aberto existe uma variedade de implementações do MPI. As implementações *open source* de maior destaque são a LAM - *Local Area Multicomputer*, e a MPICH.

Desenvolvida pela *Mathematics and Computer Science Division* do *Argonne National Laboratory*, a implementação Mpich, que pode ser encontrada em (MPICH, 2005), é uma das mais utilizadas no mundo pelo seu desempenho e é adotada por grandes empresas produtoras de *hardware* para multicomputadores.

No entanto, a implementação LAM desenvolvida pelo *Ohio Supercomputing Center*, que pode ser encontrada em (MPI, 2005), é a mais simples e exuta, oferecendo também grande desempenho com facilidade de configuração e gerenciamento. A LAM tem encontrado grande aceitação dentro do meio acadêmico e tem substituido a MPICH em alguns centros de pesquisa.

3.6.2 A biblioteca Java MPI

As implementações da biblioteca MPI originalmente não suportam a linguagem Java o que impedia inicialmente muitos programadores dessa plataforma utilizarem as potencialidades dos *clusters Beowulf*. A biblioteca mpiJava, que pode ser encontrada em (PROJECT, 2005), foi desenvolvida visando sanar essa lacuna deixada pelas implementações do padrão MPI.

A biblioteca mpiJAVA dá suporte completo ao padrão MPI 1.1 (FORUM, 2005b), implementando a hierarquia de classes mostrada na figura 17. A biblioteca não implementa o padrão MPI, apenas faz uma interface com as bibliotecas nativas para a linguagem C e C++.

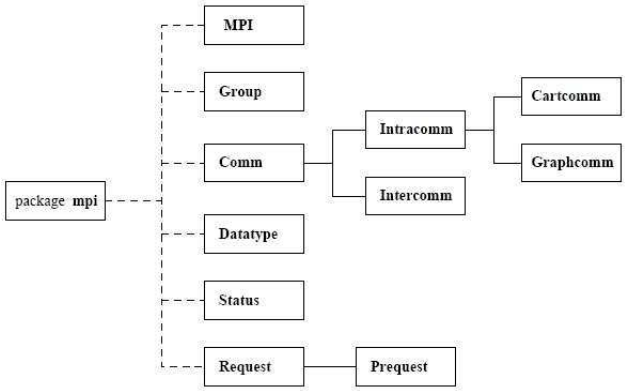

Figura 17: Classes da API JavaMPI (BAKER et al., 1999)

3.7 Medidas de Desempenho

Na passagem da solução seqüencial para a solução paralela, devem haver métodos e métricas para se medir o ganho em desempenho e a estabilidade da solução.

3.7.1 Speed Up e Eficiência

Para medir o ganho de desempenho é definida a razão *Speed Up*. O *Speed Up* é o ganho de tempo do algoritmo paralelo em relação ao algoritmo seqüencial:

$$Speedup = \frac{T_1}{T_p} \tag{3.1}$$

Onde T_1 é o tempo gasto com um processador e T_p é o tempo gasto por p processadores. A eficiência do algoritmo é definida como:

$$E = \frac{T_1}{pT_p} \tag{3.2}$$

Onde p é o número de processadores, T_1 é o tempo gasto com um processador e pT_p é o tempo gasto por p processadores. No entanto essa diminuição do tempo de execução é limitada pela comunicação entre as partes em execução, a sincronização entre processos e os pontos seqüenciais dos algoritmos. Gene Amdahl (Apud SCHLEMER 2002) definiu uma lei que expressa as limitações de aumento do *Speed Up*.

$$Speedup = \frac{1}{T_s + \frac{T_p}{n}} \tag{3.3}$$

Onde T_s é o tempo gasto nos pontos seqüenciais do algoritmo, T_p é o tempo total gasto nos pontos paralelos e n o número de processadores do cluster. A diferença entre o *Speed Up* definido em 3.1 e o o *Speed Up* definido em 3.3, segundo Amdahl, é que este último concentra os valores no intervalo $[0, 1]$.

3.7.2 Escalabilidade

Um sistema paralelo é dito escalável quando sua eficiência se mantém constante com o aumento do número de processadores no sistema. Isso se deve ao fato de que mantendo o tamanho do problema constante e aumentando o número de processadores, o *overhead* de comunicação tende a crescer e a eficiência diminuir.

A análise de escalabilidade considera a possibilidade de aumentar proporcionalmente o

tamanho do problema à medida que o número de processadores cresce, de forma a contra-balancear o *overhead* de comunicação gerado pelo aumento do número do processadores.

4 *Paralelização do Algoritmo Backpropagation*

4.1 Implementação do *Cluster*

O *cluster Beowulf* utilizado para implementar a Rede Neural Paralela constitui-se, conforme quadro 4, de quatro CPUs homogêneas sendo uma CPU mestre e três escravos interligados por uma rede *Fast Ethernet*, cuja topologia está conforme a figura 18.

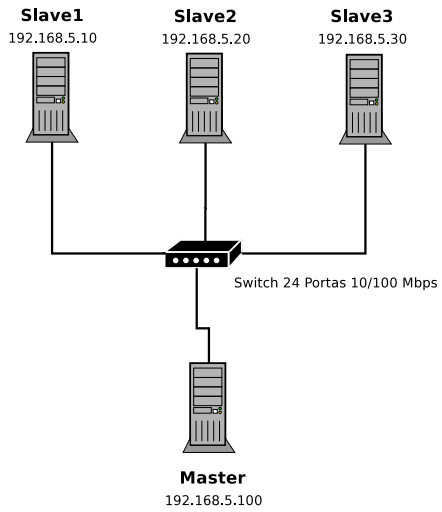

Figura 18: Arquitetura do *Cluster*

Recurso	Quantidade
CPU	4
Monitor	4
Switch Fast Ethernet 24 portas	1

Quadro 4: Descricação do *hardware* do *cluster*.

A tecnologia de rede utilizada foi a *Fast Ethernet* (IEEE 802.3) por ser uma rede local amplamente utilizada, cujo suporte é nativo para a maioria dos fabricantes de hardware e software, e de cabos UTP[1] Categoria 6. A utilização da tecnologia *Fast Ethernet* possibilita uma

[1]Unshilded Twisted Pair - Cabo de Par Trançado

velocidade de comunicação de 100 Mbits/s entre dois nós. Para evitar perda de desempenho na rede será utilizado um comutador multiporta - *switch*, responsável por enviar a mensagem ao nó correto via comutação de circuitos.

No quadro 5 temos uma descrição do *hardware* dos nós do *cluster*. Todos os nós do *cluster* possuem a mesma configuração de *hardware* que está disponível no laboratório da Faculdade de Ciências Exatas e Tecnológicas do Instituto Educacional Santo Agostinho.

Recurso	**Descrição**
Processador	AMD Athlon XP 2.8 GHz
Memória	256 MB DDR RAM (233 MHz)
Placa Mãe	nVidia
Placa Rede	3Com 3c920B 10/100 MBps
Disco Rígido	Maxtor 40 GB 7.200 RPM IDE

Quadro 5: Descrição dos Nós do *Cluster*

Em todos os nós estão instalados a distribuição Mandrake, versão 10.1 oficial, do sistema operacional Linux, com o *kernel* versão 2.6.8, em *Dual Boot* com o Windows XP. Essa é a configuração padrão das máquinas do laboratório onde foi realizada a pesquisa. No quadro 6 detalha-se a tabela de partições dos nós do *cluster*.

Partição	**Tamanho**	**Uso**
1	20 Gb	Partição NTFS (Windows XP)
2	18 Gb	Partição Ext3 (Linux)
3	1 Gb	Partição Swap (Linux)

Quadro 6: Particionamento de disco dos Nós do *Cluster*

Para que as máquinas funcionem em conjunto, é necessário a instalação de uma distribuição da biblioteca MPI. Neste experimento foi adotada a distribuição LAM MPI, já descrita na seção.

O arquivo /etc/hosts faz o mapeamento estático de nome e endereço IP, o que acelera as resoluções de nome. O arquivo foi configurado conforme figura 19:

```
master 192.168.5.100
slave1 192.168.5.10
slave2 192.168.5.20
slave3 192.168.5.30
```

Figura 19: Arquivo /etc/hosts

Os nós que compõe o *cluster* precisam executar comandos remotamente uns nos outros. Para isso são utilizados os serviços in.rlogind e in.rshd rodando nas portas tcp 513 e 514 respectivamente bem como os clientes rsh e rlogin. Os serviços são instalados na distribuição

Mandrake Linux 10.1 através do pacote rsh-server e os clientes através do pacote rsh. Para configurar o serviço são necessárias alterações nos arquivos listados no anexo A. Os clientes precisam poder logar e executar comandos remotamente sem que a autenticação requira senha.

Os pacotes de instalação da versão 7.1.1 foram baixados em (MPI, 2005). A distribuição automaticamente cria a pasta /etc/lam onde estão armazenados alguns dos arquivos de configuração. O único arquivo que precisa ser alterado é o arquivo /etc/lam/lam-bhost.def que define quais máquinas constituirão o *cluster*, conforme figura 20. Os nomes presentes nesse arquivo devem estar listados no arquivo /etc/hosts, conforme figura 19.

```
#LAM MPI: /etc/lam/lam-bhost.def

master
slave1
slave2
slave3
```

Figura 20: Arquivo /etc/lam/lam-bhost.def

O suporte a java foi baixado a partir de (SUN, 2005), na versão J2SDK 1.4.2-09 e instalado em /j2sdk, o caminho /j2sdk/bin deve ser incluido na variavel PATH do shell para facilitar a execução dos aplicativos. A biblioteca mpiJava deve ser baixada em (PROJECT, 2005), na versão 1.2.5, e instalada em /mpijava. A biblioteca mpiJava deve ser compilada apartir dos fontes conforme figura 21. Os arquivos .class gerados deverão ser copiados para a pasta raiz onde será desenvolvido o projeto em java.

```
bash$ cd /mpijava
bash$ ./configure --with-LAM
bash$ make all
bash$ cp /mpijava/lib/* /lib
bash$ PATH=$PATH:/mpijava/scripts
bash$ CLASSPATH=$CLASSPATH:/mpijava/mpi
```

Figura 21: Compilação da Biblioteca mpiJava

Terminado o processo de construção do *cluster* o mesmo pode ser inicializado com o comando lamboot. O comando lamboot contacta todos os nós do *cluster* listados conforme a figura 20, inicializa o serviço MPI nas máquinas e retorna. Apartir de então o *cluster* está pronta para ser utilizado. A biblioteca mpiJava vem com um *script* pronto para a execução de aplicativos java em *cluster*, que deve ser invocado segundo a linha de comando da figura 22, onde N é o numero de nós em que o prograva deverá rodar e ArquivoClass é o nome compilado do executável java.

```
bash$ prunjava N ArquivoClass
```

Figura 22: Criação de processo no *cluster* para java usando o script prunjava

O aplicativo também poderá ser executado diretamente apartir do comando mpirun, conforme figura 23. Para finalizar o *cluster* deve ser invocado o comando lamhalt, que acessará todos os nós finzliando o serviço MPI. Depois deste comando nenhuma aplicação que invoca a biblioteca MPI funcionará.

```
bash$ mpirun -np N /j2sdk/bin/java ArquivoClass
```

Figura 23: Criação de processo no *cluster* para java usando o comando mpirun

4.2 Paralelização do Algoritmo

Encontra-se na literatura diversas implementações de redes neurais em *cluster*, sendo PIMENTEL e FIALLOS (1999) e SEIFFERT (2002) as mais relevantes para este trabalho.

O algoritmo seqüencial implementado sobre uma rede *feed-forward* completamente conectada e acíclica, conforme vemos na figura 24. Foram utilizados os modos *on-line* e o modo *batch*;

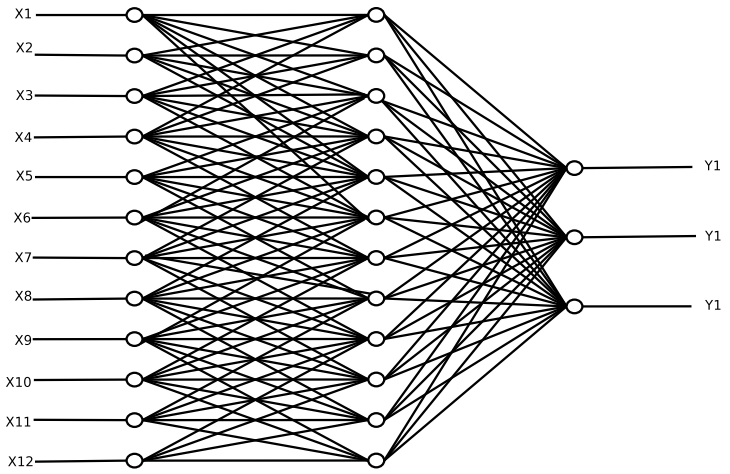

Figura 24: RNA seqüencial - Fase Forward

Nota-se que na camada de entrada, todos os neurônios possuem apenas uma entrada diferente dos demais, de forma que cada neurônio é independende dos outros da mesma camada.

O algoritmo seqüencial apresenta relações de depêndencias funcionais e de dados devido a ordem de processamento das camadas. Na fase *forward*, a saída é dependente do processamento das camadas imediatamente anteriores, conforme visualidado na equação 4.1. O valor net_k que será aplicado à função de ativação φ_k para gerar a saída da rede, conforme equação 4.2. Ele é sempre dependente da camada anterior, dado que o vetor inicial X foi passado a todos os neurônios da primeira camada (representada pela matriz de pesos W_1).

$$net_k = W_k * [\underbrace{\varphi_{k-1}(W_{k-1} * \varphi_{k-2}(\underbrace{W_{k-2} * \ldots \varphi_1(\underbrace{W_1 X + b_1}_{net_1}) \ldots + b_{k-2}}_{net_{k-2}}) + b_{k-1}}_{net_{k-1}}) + b_k \quad (4.1)$$

Onde:

- k representa o numero de camadas da rede;
- net_k Combinador linear final da rede;
- W_k é matriz de pesos da k-ésima camada da rede;
- b_k é o *bias* da k-ésima camada da rede;
- X é o vetor de entrada da RNA.

$$Y_k = \varphi_k(net_k) \quad (4.2)$$

Onde:

- Y_k é a saída final da rede;
- φ_k Função de Ativação.

Todavia, o processamento dos neurônios dentro das camadas é independente, uns dos outros, funcionalmente e por dados já que não existe realimentação dos neurônios. Este se torna o maior ponto paralelizável do algoritmo: as camadas intermediárias. As camadas têm uma relação de dependência de dados com as camadas anteriores desde que todos os neurônios aceitem entradas de todas as saídas da camada anterior, conforme equações 4.1 e 4.2. Se os dados forem divididos em blocos e os nós da primeira camada aceitarem apenas os dados relativos ao seu bloco, então a dependência diminui se restringindo a apenas alguns neurônios da camada anterior. Isso pode ser explorado já que partes de uma mesma camada

são independentes das demais, sendo dependentes apenas das partes correspondentes das outras camadas.

Dessa forma, para um *cluster* com n nós, o vetor de entrada X de tamanho t_x deverá ser divido em n partes, de tamanho igual a $\frac{t_x}{n}$, conforme equação 4.3, onde X é o vetor de entradas completo e $x_1, x_2, \ldots, x_n$ são as n partiçoes de tamanho $\frac{t_x}{n}$ do vetor X.

$$X = x_1 + x_2 + \ldots + x_n \tag{4.3}$$

Sob essa ótica, as equações 4.1 e 4.2 podem ser reescritas na forma das equações abaixo:

$$\begin{aligned}
Y_k^{x_1} &= \varphi_k^{x_1}(W_k^{x_1} * \varphi_{k-1}^{x_1}(W_{k-1}^{x_1} + \varphi_{k-1}^{x_1}(\ldots(\varphi_1^{x_1}(W_1^{x_1}x_1 + b_1^{x_1})\ldots) + b_{k-1}^{x_1}) + b_k^{x_1}) && (4.4)\\
Y_k^{x_2} &= \varphi_k^{x_2}(W_k^{x_2} * \varphi_{k-1}^{x_2}(W_{k-1}^{x_2} + \varphi_{k-1}^{x_2}(\ldots(\varphi_1^{x_2}(W_1^{x_2}x_2 + b_1^{x_2})\ldots) + b_{k-1}^{x_2}) + b_k^{x_2}) && (4.5)\\
&\vdots\\
Y_k^{x_n} &= \varphi_k^{x_n}(W_k^{x_n} * \varphi_{k-1}^{x_n}(W_{k-1}^{x_n} + \varphi_{k-1}^{x_n}(\ldots(\varphi_1^{x_n}(W_1^{x_n}x_n + b_1^{x_n})\ldots) + b_{k-1}^{x_n}) + b_k^{x_n}) && (4.6)
\end{aligned}$$

Onde:

- n é o número de nós do cluster;
- x_n é a partição do vetor X;

As saídas geradas por cada rede formam o vetor Y_{out} de saídas parciais das redes, como pode ser visto na equação 4.7:

$$Y_{out} = [Y^{x_1}, Y^{x_2}, \ldots, Y^{x_n}] \tag{4.7}$$

Apartir do vetor Y_{out} pode-se gerar a saída final da rede, Y_f, conforme equação 4.8, onde f representa a camada final da rede.

$$Y_f = \varphi_f(W_f * Y_{out} + b_f) \tag{4.8}$$

Para efetuar esse modelo, o nó mestre do *cluster* distribuiria as partições dos dados entre os nós componentes do cluster, usando a função MPI_Scatter que distribui um conjunto de dados em partes iguais para todos os membros do cluster e é chamada tanto pelo nó mestre como pelos nós escravos.

Em cada nó haverá uma sub-rede neural com $\frac{t_x}{n}$ neurônios na camada de entrada recebendo $\frac{t_x}{n}$ entradas, 1 entrada por neurônio, efetuando o processamento sobre os dados. Terminado o

processamento em cada nó, os nós se sincronizam utilizando a função MPI_Barrier e a saída de cada sub-rede seria então enviada ao nó mestre através da função MPI_Gather, onde haverá uma outra sub-rede que computará a saída final, conforme a figura 25.

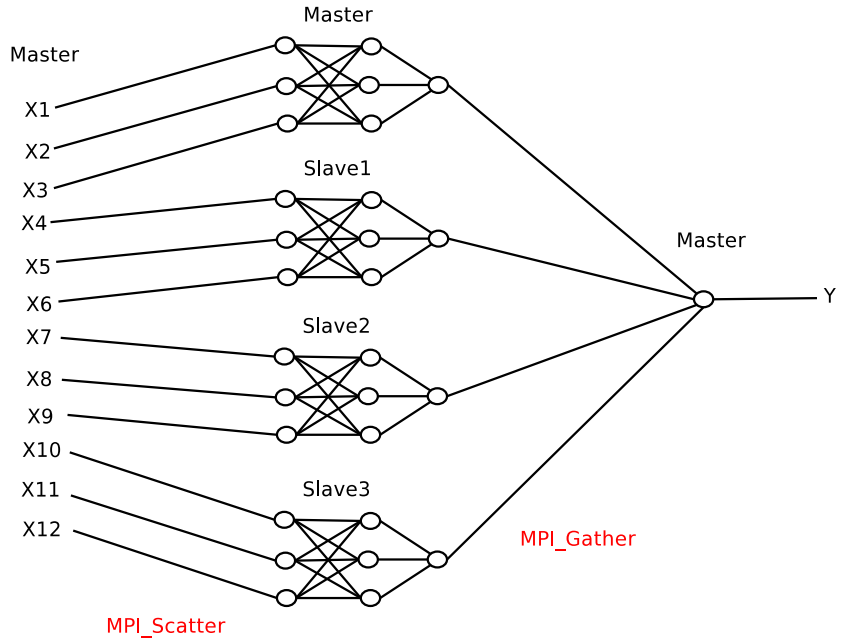

Figura 25: RNA Paralela - Fase *Forward*

O nó mestre, onde ocorre a computação da saída final da rede também computará o erro e o gradiente para a sua sub-rede neural que atua como a camada final da rede, e difundirá os valores de ambos para todos os nós, usando a função MPI_Bcast, que retropropagarão o erro e o gradiente em suas sub-redes neurais atualizando os pesos dos neurônios, conforme figura 26.

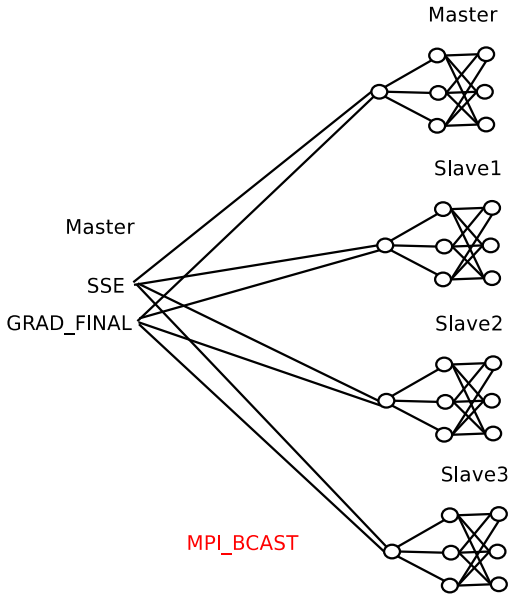

Figura 26: RNA Paralela - Fase *Backward*

As sub-redes neurais em cada nó se comportam como partes de um rede neural inteira em paralelo, usufruindo do poder computacional de cada nó para acelerar a computação da saída final.

A implementação do algoritmo foi feita na linguagem Java, devido à sua alta portabilidade e clareza de código. O diagrama de classes da implementação é mostrado na figura 27, e o código pode ser apreciado no anexo C.

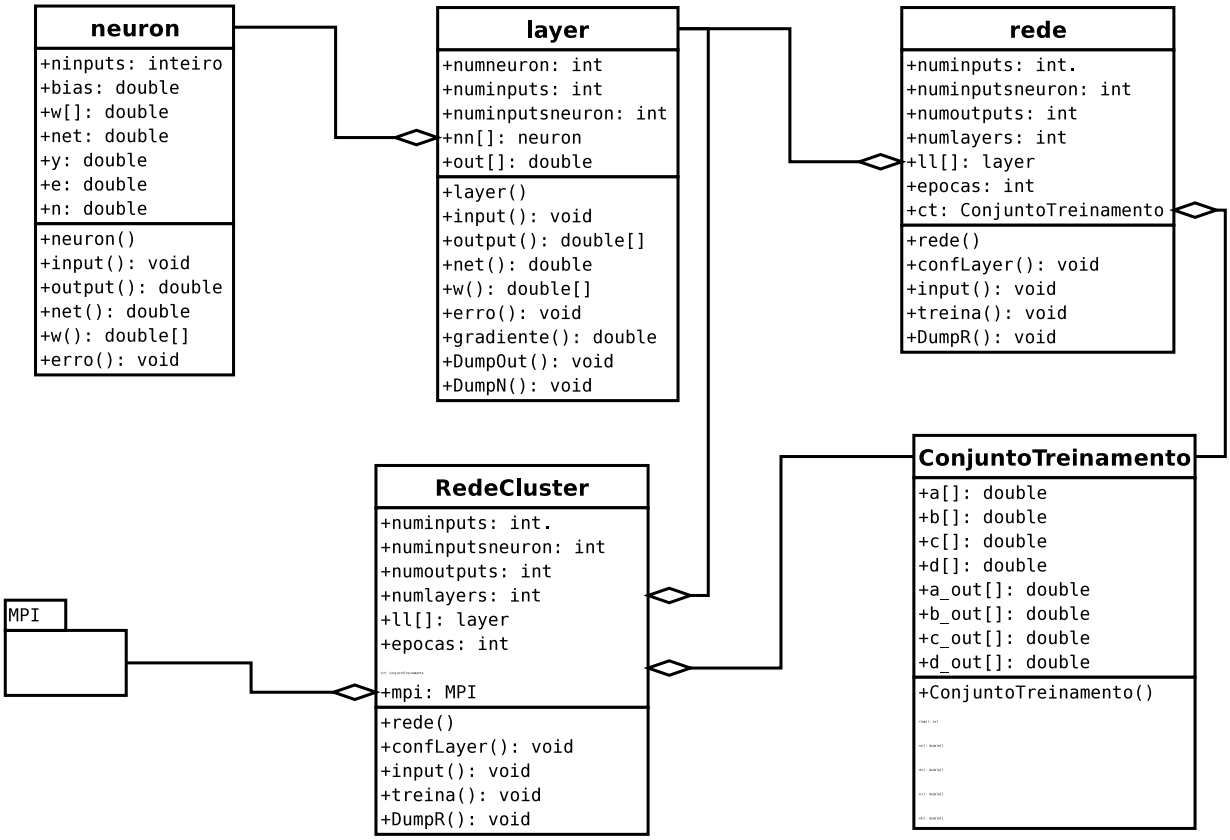

Figura 27: Diagrama de Classes

5 Testes

5.1 Parâmetros da RNA

Para a função de ativação foram testadas as funções sigmóide e a sigmóide bipolar, afim de encontrar qual dessas apresenta menor tempo de convergência no aprendizado.

Durante os testes, foi verificado que o valor da variável net estava no intervalo $[900, 1000]$. Para estes valores, as funções sigmoidais com o parâmetro α igual à 1, produziam sempre uma saída igual à 0. Para resolver esse problema, foram testados diversos valores para o parâmetro α nas funções sigmóide, conforme figura 28, e na sigmóide bipolar, conforme figura 29.

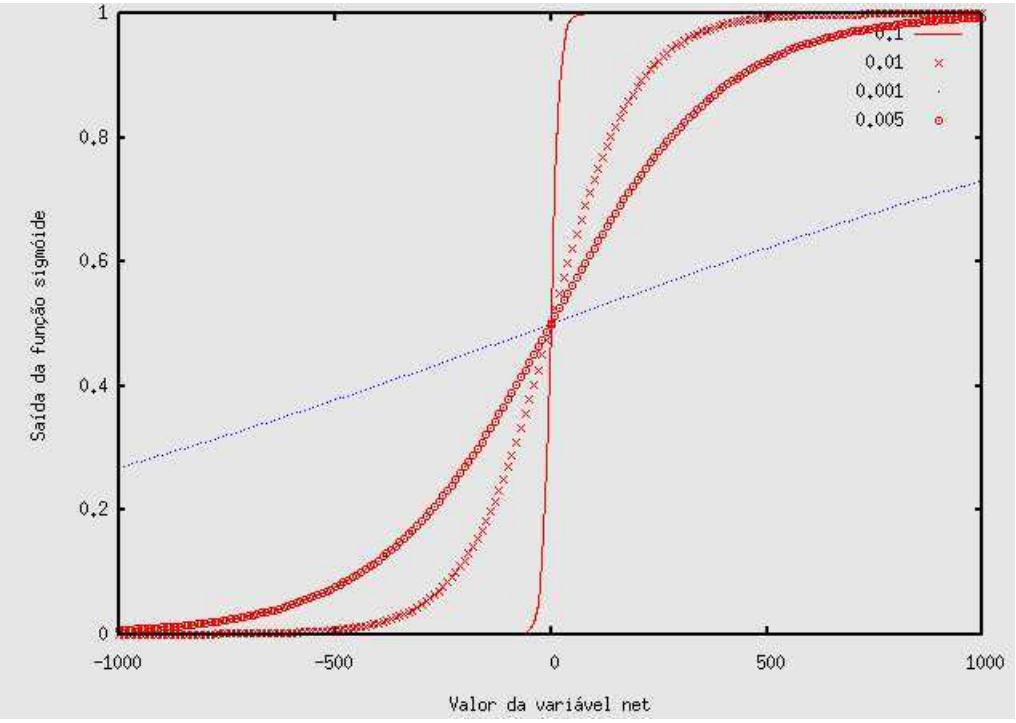

Figura 28: Parâmetro a na função sigmóide

Para a computação do gradiente utilizado no algoritmo *backpropagation*, são necessárias ainda as derivadas primeiras das funções. O parâmetro α também foi testado nas derivadas das funções sigmóide, conforme figura 30, e a sigmóide bipolar, conforme imagem 31.

Os gráficos demonstram que o melhor valor para α é $0,005$, pois, para o intervalo

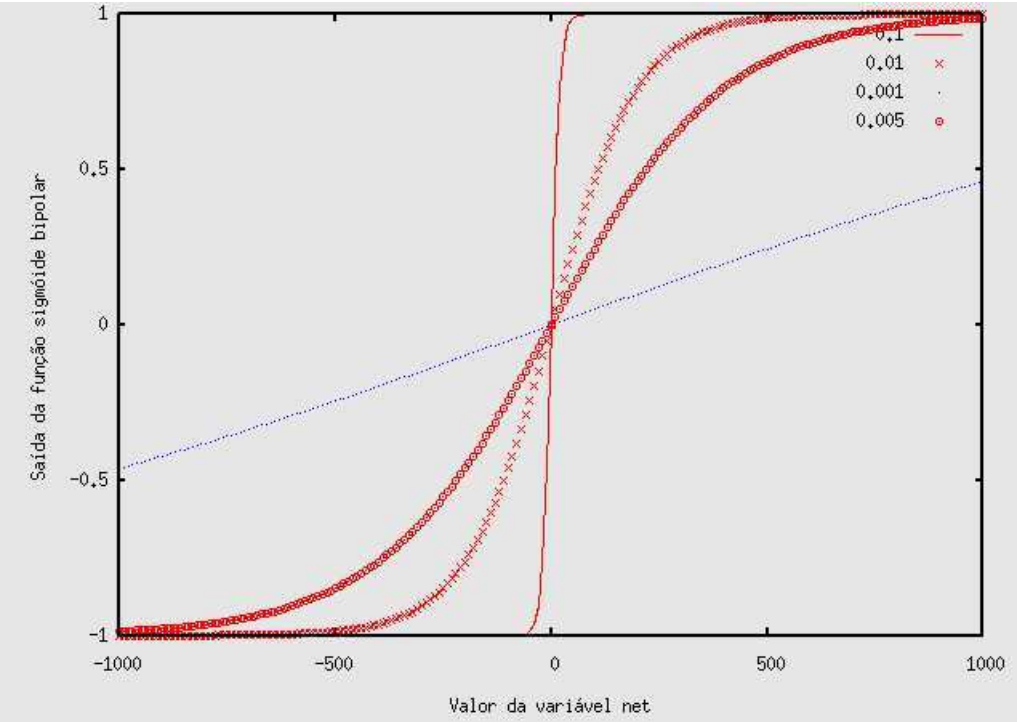

Figura 29: Parâmetro a na função sigmóide bipolar

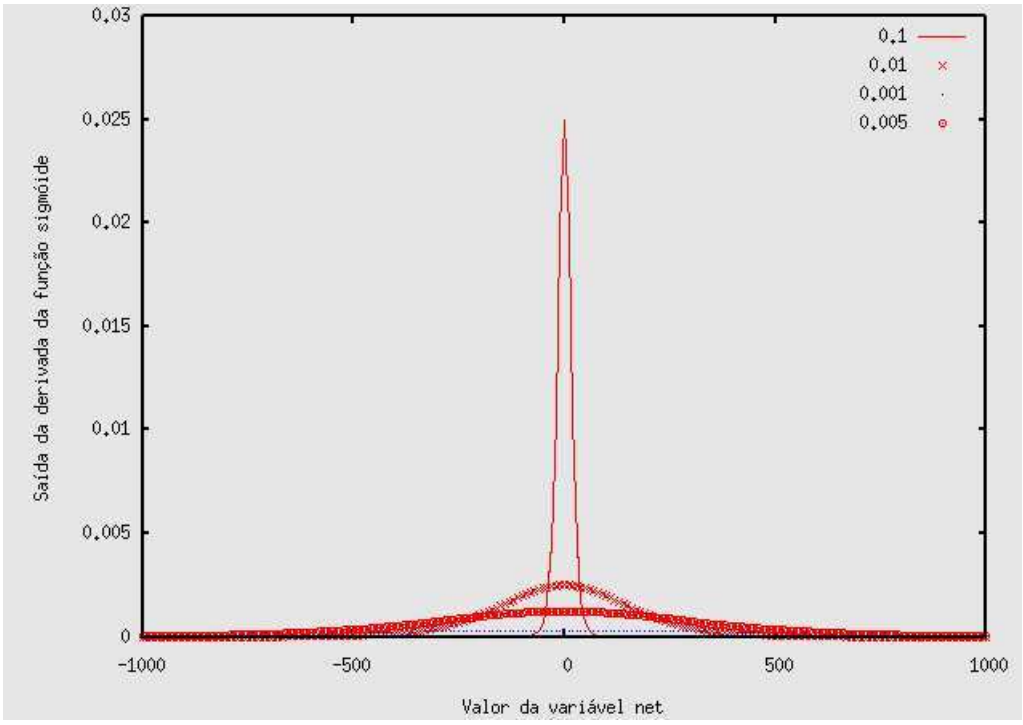

Figura 30: Parâmetro a na derivada da função sigmóide

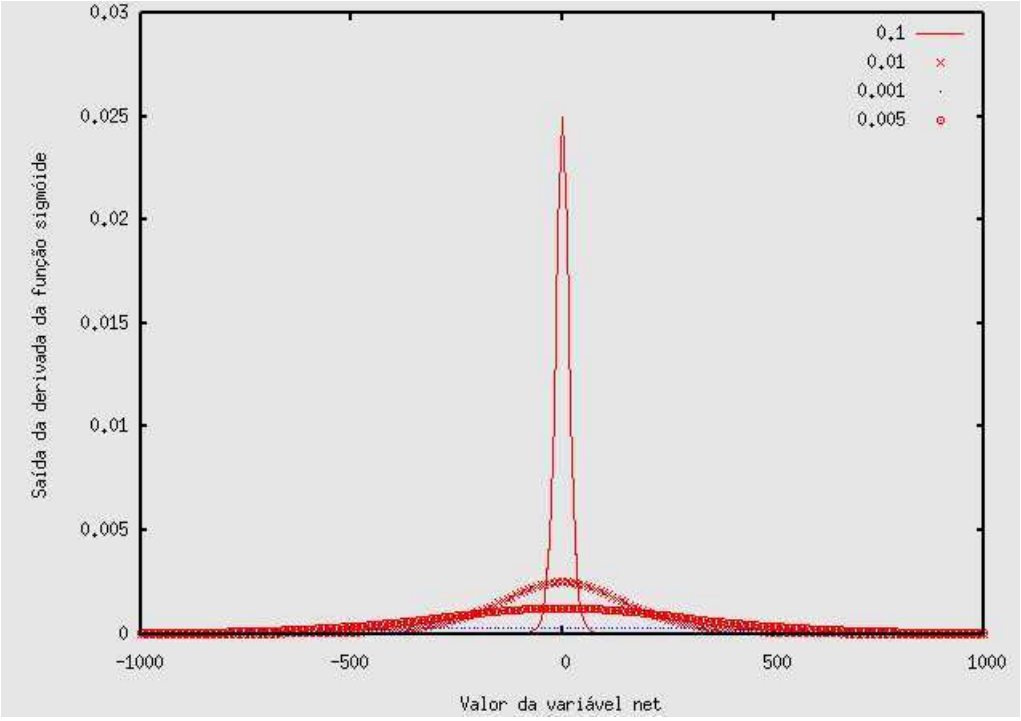

Figura 31: Parâmetro a na derivada da função sigmóide bipolar

$[-1000, 1000]$, as funções sigmóide e sigmóide bipolar produziam resultados dentro das faixas $[0, 1]$ e $[-1, 1]$ respectivamente.

A taxa de aprendizado η é um parâmetro cujo valor está comumente no intervalo $]0, 1]$. O valor é específico e dependente de cada tipo de aplicação e deve ser estipulado empiricamente. Valores altos, isto próximos de 1, tornam o aprendizado extremamente rápido mas podem contribuir para a ocorrência de *overfitting*. Valores muito baixos tornam o aprendizado excessivamente lento. Foram testados, conforme a figura 32, diversos valores para η, sendo que o valor que melhor apresenta a relação velcidade de aprendizado e não ocorrência de overfitting é 0.3.

O valor mínimo de erro define a acurácia e precisão do resultado. Quanto menor for esse valor, mais precisa será a saída da rede, e mais iterações do algoritmo serão necessárias. Não há na literatura um valor mínimo ou máximo permitido, sendo este definido empiricamente de acordo com a aplicação. Como esse valor impacta diretamente no número de iterações, foi escolhido o valor $10^{-5} = 0.00001$.

O Número de iterações máximo foi estipulado em função do número médio de iterações necessárias para a convergência com o valor mínimo de erro estipulado. Conforme figura 32, o número aproximado, arredondado para cima, é de 1000 iterações.

Na literatura, não existe um método definitivo para determinar o número de camadas na rede. Este número deve ser determinado empiricamente. Foram testadas arquiteturas com 2, 3 e 4 camadas, sendo que, por apresentarem um desempenho inferior, como a convergência

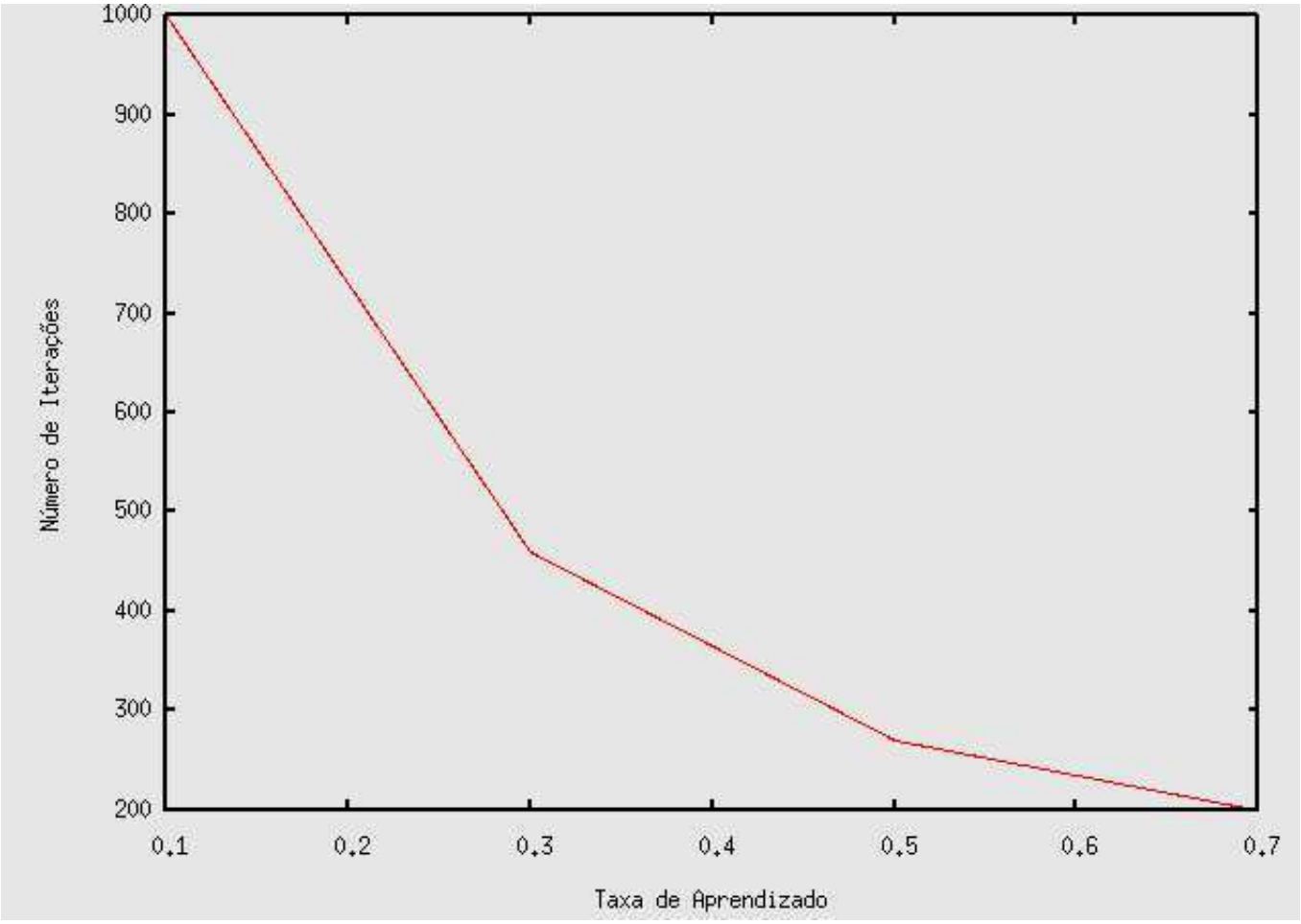

Figura 32: Impacto da taxa de aprendizado no número de iterações

mais lenta, as redes com 2 e 4 camadas foram descartadas. A rede final possui três camadas, sendo uma de entrada, uma oculta e uma de saída.

5.2 O conjunto de testes

Para testar a implementação foi criado uma aplicação de reconhecimento de caracteres, baseados em matrizes de valores binários ou bipolares representando as letras do alfabeto. A rede deve aprender e depois reconhecer os caracteres que lhe são apresentados.

Para tanto, um conjunto de teste foi estipulado, onde a RNA deve reconhecer caracteres codificados em uma matriz de 36 colunas por 40 linhas, totalizando 1440 elementos, no total de 24 matrizes, uma para cada letra do alfabeto, como exemplificado nas figuras 33 e 34. A saída é constituída de 24 matrizes de 24 elementos, onde um dos elementos sinaliza a posição no alfabeto da letra representada pela matriz de entrada, como exemplificado nas figuras 35 e 36.

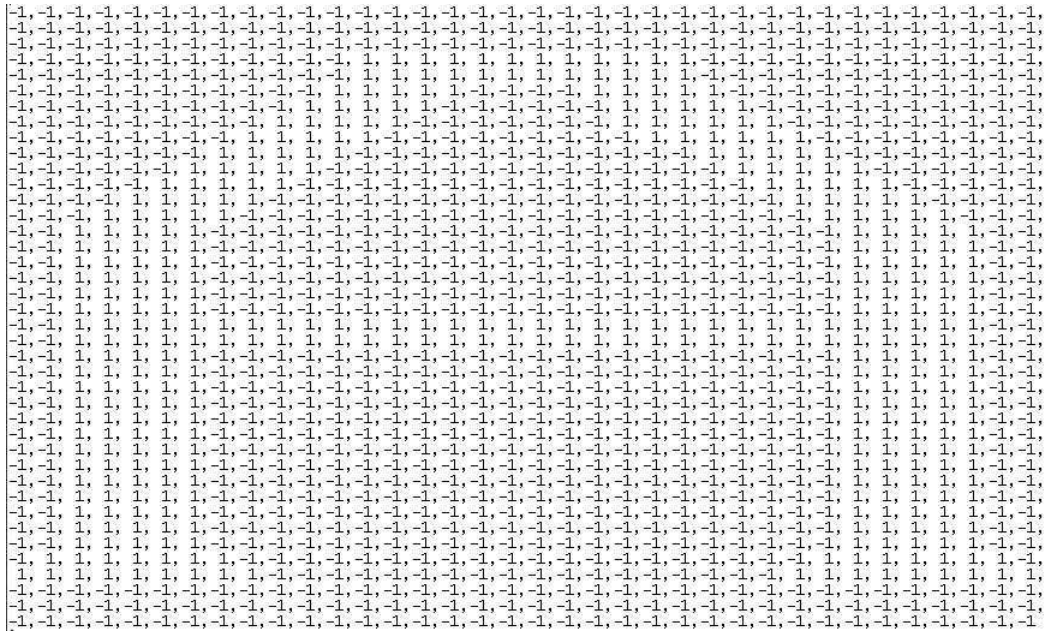

Figura 33: Matriz de entrada bipolar para o caractere A

O vetor de saída de 24 elementos representa como 1 a entrada de posição correspondente à letra representada na matriz de entrada, conforme figura 35

5.3 Testes

Nos testes realizados, a máquina seqüencial e os nós do *cluster* possuíam o mesmo ambiente operacional, com a mesma tabela de processos, que pode ser vista no anexo B. Os

```
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,
0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,
0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,
0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,
0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,0,
1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

Figura 34: Matriz de entrada binária para o caractere A

```
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1
```

Figura 35: Vetor de saída bipolar para o caractere A

```
1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

Figura 36: Vetor de saída binária para o caractere A

tempos de execução foram medidos em segundos, conforme as bibliotecas da linguagem JAVA.

Os tempos de execução comparados entre as funções sigmóide e sigmóide bipolar, conforme as tabelas 7 e 8, variam sutilmente, com a função sigmóide bipolar convergindo em tempos inferiores. Observa-se que o *Speed-up* e eficiências dos algoritmos paralelos alcançaram significativas reduções no tempo de execução do algoritmo seqüencial.

Algoritmo	**Num. Nós**	**Tempo**	**Speed Up**	**Eficiência**
Seqüencial	1	71,32	-	-
Paralelo	1	71,84	0,99	0,99
Paralelo	2	18,45	3,89	1,93
Paralelo	3	8,84	8,13	2,68
Paralelo	4	5,08	14,14	3,50

Tabela 7: Resultados aferidos com a função de ativação bipolar

Algoritmo	**Num. Nós**	**Tempo**	**Speed Up**	**Eficiência**
Seqüencial	1	73,29	-	-
Paralelo	1	72,02	1,02	1,01
Paralelo	2	19,6	3,73	1,86
Paralelo	3	9,78	7,49	2,49
Paralelo	4	6,89	10,63	2,65

Tabela 8: Resultados aferidos com a função de ativação binaria

O tempo, conforme figura 37, decai progressivamente a cada computador adicionado e mais rapidamente com a utilização da função sigmóide bipolar. Para essa função, com o cluster utilizando 2 nós, ocorreu a redução de 75% do tempo de execução do algoritmo seqüencial. Com três nós houve uma redução de 88% e com quatro, de 96%.

O *Speed-up* e a eficiência cresceram acompanhando o decrescimento do tempo de execução em relação ao número de nós do cluster, como pode ser visto nas figuras 38 e 39. Observa-se, contudo, que a taxa de crescimento do *Speed-up* e da eficiência diminui à medida que se acrescentam nós, o que leva à conclusão que o problema da granularidade, isto é, a relação do tempo efetivo de processamento e o tempo gasto com a sincronização e comunicação, se torna aparente e limitador do contínuo crescimento do *Speed-up*.

Os testes demonstraram que a função sigmóide bipolar apresenta desempenhos superiores e é mais eficiente do que a função sigmóide. As figuras 40 e 41 mostram que a função sigmóide bipolar converge mais rapidamente e apresenta uma maior taxa de *Speed-up*.

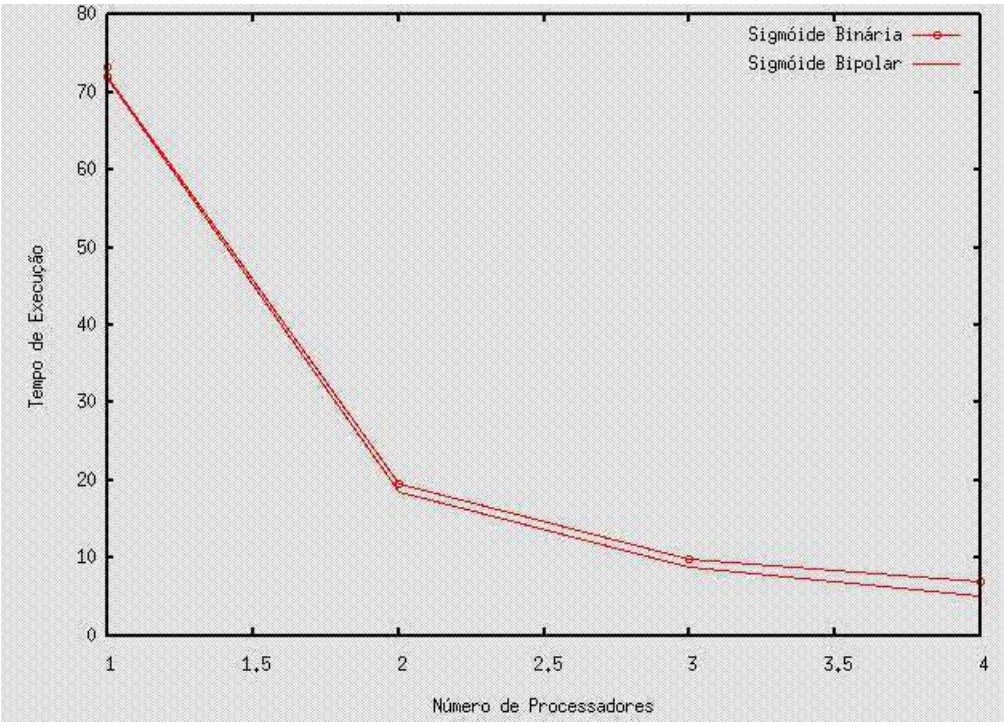

Figura 37: Variação do tempo

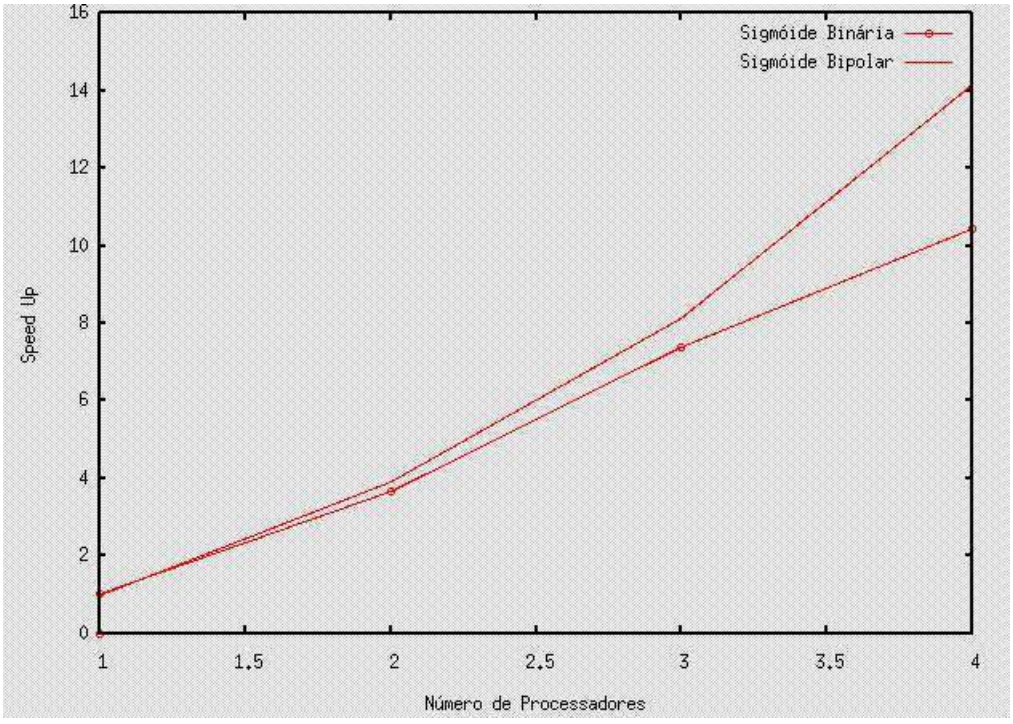

Figura 38: Variação do Speed Up

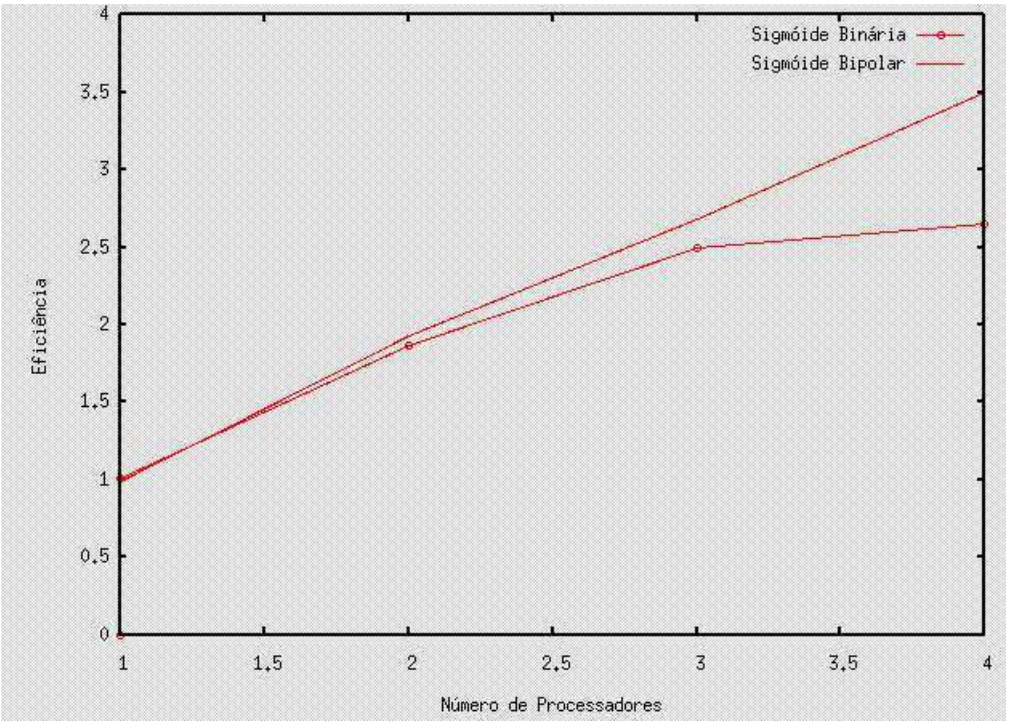

Figura 39: Variação da eficiência

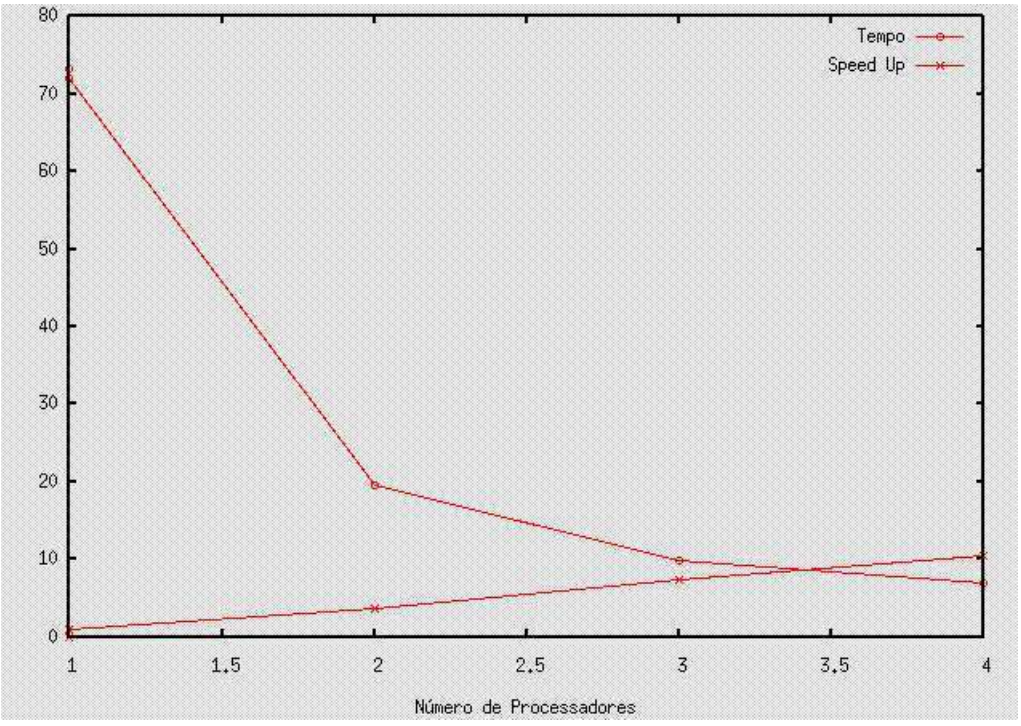

Figura 40: Variação do tempo e *speed-up* pelo número de processadores na função sigmóide

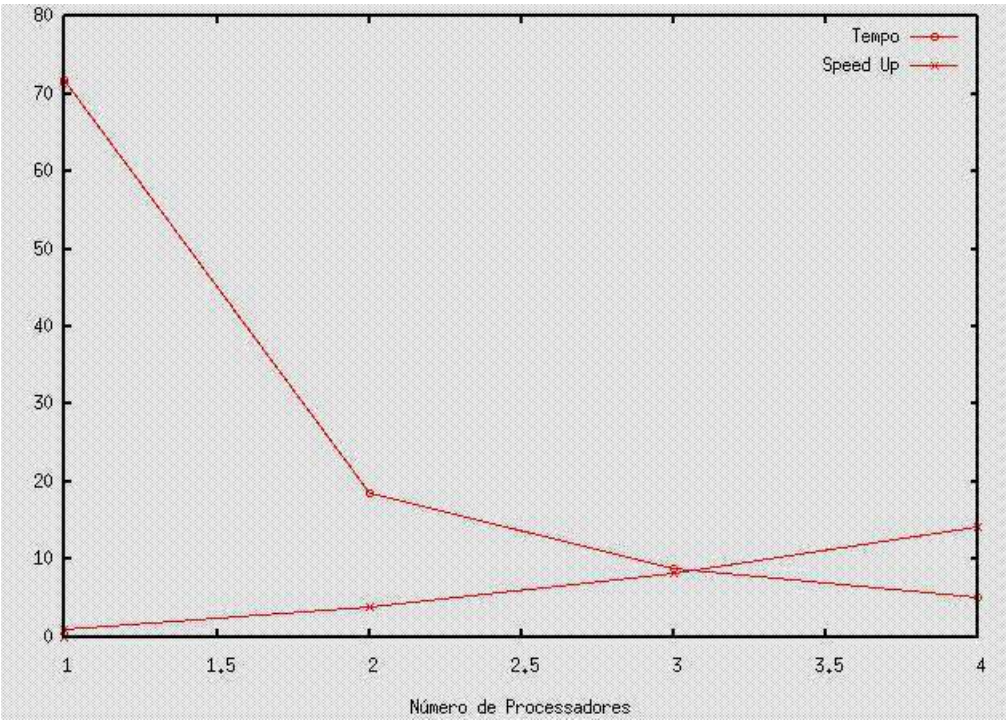

Figura 41: Variação do tempo e *speed-up* pelo número de processadores na função sigmóide bipolar

6 Conclusão

6.1 Considerações iniciais

Neste trabalho foi apresentada uma versão paralela para o algoritmo seqüencial clássico do *backpropagation*, utilizado no treinamento das RNA's. Este algoritmo foi implementado na linguagem Java, e para testá-lo foi utilizado um *cluster* da classe *beowulf* com quatro nós homogêneos.

O desenvolvimento deste trabalho dependeu das pesquisas na área da Inteligência Artificial, em especial no ramo Conexionista, que estuda as Redes Neurais Artificiais, com apresentado na capítulo 2, e das pesquisas na área da Engenharia da Computação, em especial no campo da Computação Paralela, presentes no capítulo 3.

A implementação do algoritmo foi discutida em detalhes, bem como a implementação do cluster e do ambiente de testes do algoritmo no capítulo 4, destacando-se a fundamentação teórica e as bibliotecas de programação utilizadas para explorar o paralelismo do algoritmo original. A definição dos parâmetros da rede e os testes de performance são descritos no capítulo 5, onde se demonstra e comprova as vantagens da paralelização do algoritmo.

O trabalho cumpriu os objetivos específicos que se propôs, avaliando os ganhos de performance do algoritmo paralelo em relação ao algoritmo seqüencial e demonstrando uma topologia de rede neural adequada para o ambiente paralelo.

6.2 Contribuições

Este trabalho apresentou como contribuições:

- Estratégias de paralelização do algoritmo *backpropagation*;
- Demonstração de uma implementação portável de redes neurais em um ambiente paralelo de baixo custo e elevado desempenho;
- Definição de uma arquitetura escalável para aplicações paralelas de redes neurais;

6.3 Trabalhos Futuros

A paralelização pode ser testada para outras variações do algoritmo de *Backpropagation*, bem como para os diferentes modos do algoritmo. Verifica-se ainda a possibilidade de estender para outros modelos de redes neurais, incluindo-se as redes recorrentes e de processamento temporal. Nessa última, é particularmente interessante testar o modelo paralelo, dado que nelas, o tempo de resposta é um parâmetro critico.

O cluster pode ser implementado com mais nós, a fim de testar até que ponto a eficiência se mantém crescente e o algoritmo permace estável e escalável. É importante verificar a relação do número de entradas da rede com o número de camadas da rede e o número de nós do cluster.Pode-se testar também outras arquiteturas de processamento paralelo.

O grau de dependência que os neurônios da camada intermediária têm entre si deve ser estudado, a fim de verificar se o particionamento dos dados de entrada reflete no aprendizado e na qualidade das respostas geradas pela rede.

ANEXO A – Arquivos de Configuração do Cluster

A.1 Arquivo /etc/xinet.d/rsh

```
# /etc/xinet.d/rsh

# default: on
# description: The rshd server is the server for the rcmd(3) routine and, \
#       consequently, for the rsh(1) program.  The server provides \
#       remote execution facilities with authentication based on \
#       privileged port numbers from trusted hosts.
service shell
{
        socket_type             = stream
        wait                    = no
        user                    = root
        log_on_success          += USERID
        log_on_failure          += USERID
        server                  = /usr/sbin/in.rshd
        disable                 = no
}
```

A.2 Arquivo /etc/xinet.d/rlogin

```
# /etc/xinet.d/rlogin

# default: on
# description: rlogind is the server for the rlogin(1) program.  The server \
#       provides a remote login facility with authentication based on \
#       privileged port numbers from trusted hosts.
service login
{
        socket_type             = stream
        wait                    = no
        user                    = root
        log_on_success          += USERID
        log_on_failure          += USERID
        server                  = /usr/sbin/in.rlogind
        disable                 = no
}
```

A.3 Arquivo /etc/pam.d/rsh

```
#%PAM-1.0
# For root login to succeed here with pam_securetty, "rsh" must be
# listed in /etc/securetty.
auth       required     pam_nologin.so
auth       required     pam_securetty.so
auth       required     pam_env.so
auth       required     pam_rhosts_auth.so
account    required     pam_stack.so service=system-auth
session    required     pam_stack.so service=system-auth
```

A.4 Arquivo /etc/pam.d/rlogin

```
#%PAM-1.0
# For root login to succeed here with pam_securetty, "rlogin" must be
# listed in /etc/securetty.
auth       required     pam_nologin.so
auth       required     pam_securetty.so
auth       required     pam_env.so
auth       sufficient   pam_rhosts_auth.so
auth       required     pam_stack.so service=system-auth
account    required     pam_stack.so service=system-auth
password   required     pam_stack.so service=system-auth
session    required     pam_stack.so service=system-auth
```

A.5 Arquivo /etc/hosts.allow

```
#
# hosts.allow   This file describes the names of the hosts which are
#               allowed to use the local INET services, as decided
#               by the '/usr/sbin/tcpd' server.
#
ALL:    master slave1 slave2 slave3
```

A.6 Arquivo /home/usuario/.rhosts

```
master usuario
slave1 usuario
slave2 usuario
slave3 usuario
```

ANEXO B – Tabela de Processos Dos nós do cluster

```
 PID TTY      STAT   TIME COMMAND
   1 ?        S      0:01 init [3]
   2 ?        SN     0:00 [ksoftirqd/0]
   3 ?        S<     0:00 [events/0]
   4 ?        S<     0:00 [khelper]
   5 ?        S<     0:00 [kblockd/0]
  39 ?        S      0:00 [pdflush]
  40 ?        S      0:00 [pdflush]
  42 ?        S<     0:00 [aio/0]
  41 ?        S      0:00 [kswapd0]
 145 ?        S      0:00 [kseriod]
 267 ?        S      0:00 [kjournald]
 423 ?        S<s    0:00 udevd
 820 ?        S      0:00 [khubd]
2292 ?        Ss     0:00 portmap
2316 ?        Ss     0:00 syslogd -m 0 -a /var/spool/postfix/dev/log
2338 ?        Ss     0:00 klogd -2
2399 ?        Ss     0:00 rpc.statd
2459 ?        Ss     0:00 /usr/sbin/atd
2504 ?        Ss     0:00 /usr/sbin/sshd
2545 ?        Ss     0:00 xinetd -stayalive -reuse -pidfile /var/run/xinetd.pid
2586 ?        S      0:00 [nfsd]
2587 ?        S      0:00 [nfsd]
2588 ?        S      0:00 [nfsd]
2589 ?        S      0:00 [nfsd]
2590 ?        S      0:00 [nfsd]
2591 ?        S      0:00 [nfsd]
2592 ?        S      0:00 [nfsd]
2593 ?        S      0:00 [nfsd]
2600 ?        S      0:00 [lockd]
2601 ?        S      0:00 [rpciod]
2606 ?        Ss     0:00 rpc.mountd
2897 ?        Ss     0:00 crond
3063 tty3     Ss+    0:00 /sbin/mingetty tty3
3064 tty4     Ss+    0:00 /sbin/mingetty tty4
3065 tty5     Ss+    0:00 /sbin/mingetty tty5
3066 tty6     Ss+    0:00 /sbin/mingetty tty6
4396 tty1     Ss+    0:00 /sbin/mingetty tty1
4411 tty2     Ss+    0:00 /sbin/mingetty tty2
4475 ?        Ss     0:00 sshd: root@pts/0
4477 pts/0    Ss     0:00 -bash
4548 ?        S      0:01 /usr/bin/lamd -H 192.168.4.100 -P 32808 -n 0 -o 0
5429 pts/0    R+     0:00 ps -ax
```

ANEXO C – Código Fonte da Rede Neural

C.1 Arquivo ativacao.java

```
1
2  import java.math.*;
3
4
5  public class ativacao {
6
7
8          public static double sigmoide(double a) {
9                  return 1.0 / (1.0 + Math.exp(-0.005*a));
10         }
11
12
13         public static double sigmoide_derivada(double a) {
14                 return 0.005*sigmoide(a) * (1.0 - sigmoide(a));
15         }
16
17         public static double sigmoidebipolar(double a) {
18                 return (2 / (1 + Math.exp(-0.005*a))) - 1;
19         }
20
21
22         public static double sigmoidebipolar_derivada(double a) {
23                 return (0.005/2) * (( 1 + sigmoidebipolar(a)) * ( 1 - sigmoidebipolar(a)));
24         }
25 }
```

C.2 Arquivo neuron.java

```
1
2  import java.math.*;
3
4  public class neuron {
5
6          ////////////////
7          // VARIAVEIS
8          ////////////////
9
10         public int ninputs;            // Numero de Entradas do neuronio
11         public double bias;            // Bias threshold gate
12         public double w[];             // Pesos (weight)
13         public double net;             // net
```

```
14          public double y;                // output
15          public double e;                // erro
16          public double n;                // Taxa de aprendizado
17
18          ////////////////
19          // INIT
20          ////////////////
21
22          public neuron(int ni) {
23                  ninputs = ni;
24                  y = 0;
25                  net = 0;
26                  n = 0.99;
27                  bias = Math.random() - 0.5;
28                  w = new double[ninputs];
29
30                  for(int tmp = 0; tmp < ninputs; tmp++) {
31                          w[tmp] = Math.random() - 0.5;
32                  }
33
34          }
35
36          ////////////////
37          // FUNCOES
38          ////////////////
39
40
41          public void input_binaria(double in[]) {
42
43                  // Calcula o net
44
45                  net = 0.0;
46                  for(int tmp = 0; tmp < ninputs; tmp++) {
47                          net = net + w[tmp]*in[tmp];
48                  }
49                  net = net + (1 * bias);
50
51                  // Calcula o y
52
53                  y = ativacao.sigmoide(net);
54
55          }
56
57          public void input_bipolar(double in[]) {
58
59                  // Calcula o net
60
61                  net = 0.0;
62                  for(int tmp = 0; tmp < ninputs; tmp++) {
63                          net = net + w[tmp]*in[tmp];
64                  }
65                  net = net + (1 * bias);
66
67                  // Calcula o y
68
69                  y = ativacao.sigmoidebipolar(net);
70
71          }
72
```

```
73
74          public double output() {
75                  return y;
76          }
77
78          public double net() {
79                  return net;
80          }
81
82          public double[] w() {
83                  return w;
84          }
85
86
87          public void erro(double in[], double er, double nn) {
88                  e = er;
89                  n = nn;
90                  for(int tmp = 0; tmp < ninputs; tmp++) {
91                          w[tmp] = w[tmp] + (n * er * in[tmp]);
92                  }
93                  bias = bias + (n * er);
94          }
95
96          ////////////////
97          // DUMPS
98          ////////////////
99
100         public void  DumpW() {
101
102                 for(int tmp = 0; tmp < ninputs; tmp++) {
103                         System.out.println(w[tmp]);
104                 }
105         }
106
107         public void  DumpO() {
108
109                 System.out.println("Net: " + Double.toString(net));
110                 System.out.println("Y: " + Double.toString(y));
111         }
112
113
114         ////////////////
115         // MAIN PARA TESTES
116         ////////////////
117
118
119         public static void main (String args [ ]) {
120
121         }
122 }
```

C.3 Arquivo layer.java

```
1
2   public class layer {
3
4           ////////////////
5           // VARIAVEIS
```

```
6         ////////////////
7
8
9         public int numneurons;          // Num. Neuronios da Camada
10        public int numinputs;           // Num. Entradas da Camada
11        public int numinputsneuron;     // Num. Entradas por neuronio
12                                        // numinputsneuron = numinputs / numneurons
13
14        public neuron nn[];
15
16        public double out[];
17
18        ////////////////
19        // INIT
20        ////////////////
21
22        public layer(int numn, int ni, int nim) {
23
24                numneurons = numn;
25                numinputs = ni;
26                numinputsneuron = nim;
27
28                nn = new neuron[numneurons];
29
30                for(int tmp = 0; tmp < numneurons; tmp++) {
31                        nn[tmp] = new neuron(numinputsneuron);
32                }
33
34                out = new double[numneurons];
35        }
36
37        ////////////////
38        // FUNCOES
39        ////////////////
40
41
42        public void input_binaria(double in[]) {
43
44                if(numinputsneuron == numinputs) {
45                        for(int tmp = 0; tmp < numneurons; tmp++) {
46                                nn[tmp].input_binaria(in);
47                                out[tmp] = nn[tmp].output();
48                        }
49                } else {
50                        for(int tmp = 0; tmp < numneurons; tmp++) {
51                                double in2[] = new double[numinputsneuron];
52                                for(int tmp2 = 0; tmp2 < numinputsneuron; tmp2++) {
53                                        in2[tmp2] = in[tmp * numinputsneuron + tmp2];
54                                }
55                                nn[tmp].input_binaria(in);
56                                out[tmp] = nn[tmp].output();
57                        }
58                }
59        }
60
61        public void input_bipolar(double in[]) {
62
63                if(numinputsneuron == numinputs) {
64                        for(int tmp = 0; tmp < numneurons; tmp++) {
```

```
65                              nn[tmp].input_bipolar(in);
66                              out[tmp] = nn[tmp].output();
67                      }
68              } else {
69                      for(int tmp = 0; tmp < numneurons; tmp++) {
70                              double in2[] = new double[numinputsneuron];
71                              for(int tmp2 = 0; tmp2 < numinputsneuron; tmp2++) {
72                                      in2[tmp2] = in[tmp * numinputsneuron + tmp2];
73                              }
74                              nn[tmp].input_bipolar(in);
75                              out[tmp] = nn[tmp].output();
76                      }
77              }
78      }
79
80
81      public double[] output() {
82              return out;
83      }
84
85      public double net(int i) {
86              return nn[i].net();
87      }
88
89      public double[] w(int i) {
90              return nn[i].w();
91      }
92
93      public void erro(double in[], double err, double n) {
94              for(int tmp = 0; tmp < numneurons; tmp++) {
95                      nn[tmp].erro(in,err, n);
96              }
97
98      }
99
100     // Gradiente para a ultima camada
101
102
103     public double gradiente_binaria(double in[], double o[], double n){
104             double grad_final = 0.0;
105             double g = 0.0;
106             for(int tmp = 0; tmp < numneurons; tmp++){
107                     grad_final = o[tmp] - out[tmp];
108                     nn[tmp].erro(in,grad_final,n);
109                     g = g + grad_final;
110             }
111             return g;
112
113     }
114
115     public double gradiente_bipolar(double in[], double o[], double n){
116             double grad_final = 0.0;
117             double g = 0.0;
118             for(int tmp = 0; tmp < numneurons; tmp++){
119                     grad_final = o[tmp] - out[tmp];
120                     nn[tmp].erro(in,grad_final,n);
121                     g = g + grad_final;
122             }
123             return g;
```

```
124
125         }
126
127
128         // Gradiente para a outras camadas
129
130         public double gradiente_binaria(double sse, double g, double in[], double n) {
131                 double grad = 0.0;
132                 for(int tmp = 0;tmp < numneurons; tmp++) {
133                         grad = 0;
134                         for(int tmp2 = 0;tmp2 < nn[tmp].ninputs; tmp2++) {
135                                 grad = grad + (g * nn[tmp].w[tmp2]);
136                         }
137                         grad = grad * ativacao.sigmoide_derivada(nn[tmp].net());
138                         nn[tmp].erro(in,grad,n);
139                 }
140
141                 return grad;
142         }
143
144         public double gradiente_bipolar(double sse, double g, double in[], double n) {
145                 double grad = 0.0;
146                 for(int tmp = 0;tmp < numneurons; tmp++) {
147                         grad = 0;
148                         for(int tmp2 = 0;tmp2 < nn[tmp].ninputs; tmp2++) {
149                                 grad = grad + (g * nn[tmp].w[tmp2]);
150                         }
151                         grad = grad * ativacao.sigmoidebipolar_derivada(nn[tmp].net());
152                         nn[tmp].erro(in,grad, n);
153                 }
154
155                 return grad;
156         }
157
158
159         ////////////////
160         // DUMPS
161         ////////////////
162
163
164         public void DumpOut() {
165                 for(int tmp = 0; tmp < numneurons; tmp++) {
166                         System.out.println(Double.toString(out[tmp]));
167                 }
168         }
169
170         public void DumpN() {
171
172                 for(int tmp = 0; tmp < numneurons; tmp++) {
173                         System.out.println("Neurônio:" + Integer.toString(tmp));
174                         //nn[tmp].DumpO();
175                         nn[tmp].DumpW();
176                 }
177         }
178
179         ////////////////
180         // MAIN PARA TESTES
181         ////////////////
182
```

```
        public static void main (String args []) {
                //System.out.println("Layer!!!\n");
                layer a = new layer(4, 16, 1);
                a.DumpN();
        }
}
```

C.4 Arquivo A.java

```
public class A extends padrao {

public A() {
        p_x = a;
        d_y = a_out;
}

        public static double a_out[] = {1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};


public static double a[] =  { -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,
-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,
-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,
-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,
-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,
-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,
-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,
-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,
-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,
-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,
-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,
-1,-1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,-1,-1,
-1,-1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,-1,-1,
-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,
-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,
-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,
-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,
-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,
-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,
-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,
-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,
-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,
-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,
-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,
-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,
-1,-1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1,-1,-1,
-1, 1, 1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1, 1, 1,-1,
 1, 1, 1, 1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
```

```
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1 };
}
```

C.5 Arquivo AA.java

```
public class AA extends padrao {

public AA() {
        p_x = a;
        d_y = a_out;
}

        public static double a_out[] = {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};


public static double a[] =  {
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,
0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,
0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,
0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,
0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
0,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,0,
1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 };
}
```

C.6 Arquivo B.java

```
public class B extends padrao {

public B() {
        p_x = b;
        d_y = b_out;
}

        public static double b_out[] = {-1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};


public static double b[] =  { -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,-1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,-1,-1,-1,
-1,-1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,-1,-1,-1,-1,
-1,-1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,-1,-1,-1,-1,-1,
-1,-1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,-1,-1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,-1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,-1,-1,
-1,-1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,-1,-1,
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, };
}
```

C.7 Arquivo BB.java

```
public class BB extends padrao {
```

```
2
3  public BB() {
4          p_x = b;
5          d_y = b_out;
6  }
7
8          public static double b_out[] = {0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
9
10
11 public static double b[] =  { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
12 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
13 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
14 0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,
15 0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,
16 0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,
17 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
18 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
19 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
20 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
21 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
22 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
23 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
24 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
25 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
26 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
27 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
28 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
29 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,
30 0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,
31 0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,
32 0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,
33 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,
34 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
35 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
36 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
37 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
38 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
39 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
40 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
41 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
42 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
43 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
44 0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
45 0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,
46 0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,
47 0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,
48 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
49 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
50 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, };
51 }
```

C.8 Arquivo ConjuntoTreinamento.java

```
1
2  public class ConjuntoTreinamento {
3          A a;    B b;    C c;    D d;    E e;
4          F f;    G g;    H h;    I i;    J j;
5          K k;    L l;    M m;    N n;    O o;
```

```
6           P p;    Q q;    R r;    S s;    T t;
7           U u;    V v;    X x;    Z z;
8
9           AA aa;  BB bb;  CC cc;  DD dd; EE ee;
10
11          padrao in_binaria[], in_bipolar[];
12
13          public ConjuntoTreinamento() {
14
15                  aa = new AA(); bb = new BB(); cc = new CC(); dd = new DD();
16                  ee = new EE();
17
18                  a = new A(); b = new B(); c = new C(); d = new D();
19                  e = new E(); f = new F(); g = new G(); h = new H();
20                  i = new I(); j = new J(); k = new K(); l = new L();
21                  m = new M(); n = new N(); o = new O(); p = new P();
22                  q = new Q(); r = new R(); s = new S(); t = new T();
23                  u = new U(); v = new V(); x = new X(); z = new Z();
24
25                  in_bipolar = new padrao[5];
26
27                  in_bipolar[0] = a;
28                  in_bipolar[1] = b;
29                  in_bipolar[2] = c;
30                  in_bipolar[3] = d;
31                  in_bipolar[4] = e;
32
33                  in_binaria = new padrao[6];
34
35                  in_binaria[0] = aa;
36                  in_binaria[1] = bb;
37                  in_binaria[2] = u;
38                  in_binaria[3] = v;
39                  in_binaria[4] = x;
40                  in_binaria[5] = z;
41
42          }
43
44          public int in_tam() {
45                  return in_binaria[0].p_x.length ;
46          }
47
48          public int out_tam() {
49                  return in_binaria[0].d_y.length ;
50          }
51
52          public double[] in_binaria(int i) { return in_binaria[i].p_x; }
53          public double[] out_binaria(int i) { return in_binaria[i].d_y; }
54
55          public double[] in_bipolar(int i) { return in_bipolar[i].p_x; }
56          public double[] out_bipolar(int i) { return in_bipolar[i].d_y; }
57
58  }
```

C.9 Arquivo rede.java

```
1
2   import java.math.*;
```

```
3
4   public class rede {
5
6           ////////////////
7           // VARIAVEIS
8           ////////////////
9
10          // Parametros da rede
11
12          public int numinputs;           // Num. Entradas da Rede
13          public int numinputsneuron;     // Num. Entradas dos neuronios da camada de entrada
14          public int numoutputs;          // Num. saidas
15          public int numlayers;           // Num. Camadas
16
17          public int iter_max;            // Num. Max. de Iteracoes
18          public double nr;               // Taxa de Aprendizado
19          public double precisao;         // Precisao do Erro
20
21          public double out[];            // Saida final da rede
22          public double e[];              // Erros
23          public double erro;             // Erro Final
24          public double sse;              // Squared Sum Error = Soma dos Erros Quadraticos = sum( (e^2)/2 )
25
26          layer ll[];
27
28          int epocas;
29
30          ////////////////
31          // INIT
32          ////////////////
33
34          public rede(int ni, int no, int nl) {
35
36                  numinputs = ni;
37                  numoutputs = no;
38                  numlayers = nl;
39
40                  iter_max = 1000;
41                  precisao = 0.001;
42                  nr = 0.3;
43
44                  epocas = 0;
45
46                  out = new double[no];
47                  e = new double[no];
48                  erro = 0.0;
49                  sse = 0.6;
50
51                  ll = new layer[nl];
52
53          }
54
55
56          ////////////////
57          // FUNCOES
58          ////////////////
59
60
61          public void confLayer(int index, int nn, int ni, int nin) {
```

```
62                  ll[index] = new layer(nn, ni, nin);
63          }
64
65          public void input_binaria(double in[]) {
66
67                  // Feed Forward
68
69                  // 1a Camada
70                  ll[0].input_binaria(in);
71
72                  for(int tmp = 1; tmp <= numlayers - 1; tmp++) {
73                          ll[tmp].input_binaria(ll[tmp-1].output());
74                  }
75
76                  // Salva a saida da Rede
77                  out = ll[numlayers-1].output();
78
79          }
80
81          public void input_bipolar(double in[]) {
82
83                  // Feed Forward
84
85                  // 1a Camada
86                  ll[0].input_bipolar(in);
87
88                  for(int tmp = 1; tmp <= numlayers - 1; tmp++) {
89                          ll[tmp].input_bipolar(ll[tmp-1].output());
90                  }
91
92                  // Salva a saida da Rede
93                  out = ll[numlayers-1].output();
94
95          }
96
97          public void treina_binaria(double in[], double d[], double n, double p) {
98
99                  double grad_final = 0.0;
100                 double grad_inter = 0.0;
101
102                 precisao = p;
103
104                 epocas = 0;
105                 sse = 10.0;
106
107                 System.out.println("################################################");
108                 System.out.println("## Iniciando o treinamento BINARIA ");
109                 System.out.println("## n: " + Double.toString(n) + " NC: " + Integer.toString(numlayers) + "Precisao: " +
110                 System.out.println("################################################");
111
112                 while(Math.abs(sse) > precisao && epocas <= 1000) {
113
114                         grad_final = 0.0;
115
116                         input_binaria(in);
117
118                         sse = 0.0;
119                         erro = 0.0;
120                         for(int tmp = 0; tmp < numoutputs; tmp++) {
```

```
121                             e[tmp] = (d[tmp] - out[tmp]);
122                             erro = erro + e[tmp];
123                             sse = sse + (e[tmp] * e[tmp]);
124                     }
125
126                     sse = sse / 2;
127
128                     // Correção de erros
129
130                     if(Math.abs(sse) > precisao) {
131
132                             //camada de saida
133
134                             grad_final = ll[numlayers-1].gradiente_binaria(ll[numlayers-2].output(),d,n);
135
136                             //demais camadas
137
138
139                             for(int tmp = numlayers -2; tmp > 0; tmp--) {
140                                     grad_inter = ll[tmp].gradiente_binaria(erro, erro, ll[tmp-1].output(),n);
141                             }
142
143                             grad_inter = ll[0].gradiente_binaria(erro, erro, in, n);
144
145                     }
146                     if((epocas % 10) == 0) {
147                             System.out.println("Iteração: " + Integer.toString(epocas) + "\tSSE: " + Double.toString(s
148                     }
149                     epocas = epocas + 1;
150             }
151
152             System.out.println("Num. Iterações: " + Integer.toString(epocas) + " SSE: " + Double.toString(sse) + " Err
153     }
154
155
156
157     public void treina_bipolar(double in[], double d[], double n, double p) {
158
159             double grad_final = 0.0;
160             double grad_inter = 0.0;
161
162             precisao = p;
163
164             epocas = 0;
165             sse = 10.0;
166
167             System.out.println("################################################");
168             System.out.println("## Iniciando o treinamento BIPOLAR");
169             System.out.println("## n: " + Double.toString(n) + " NC: " + Integer.toString(numlayers) + "Precisao: " +
170             System.out.println("################################################");
171
172             while(Math.abs(sse) > precisao && epocas <= 1000) {
173
174                     grad_final = 0.0;
175
176                     input_bipolar(in);
177
178                     sse = 0.0;
179                     erro = 0.0;
```

```
180                     for(int tmp = 0; tmp < numoutputs; tmp++) {
181                             e[tmp] = (d[tmp] - out[tmp]);
182                             erro = erro + e[tmp];
183                             sse = sse + (e[tmp] * e[tmp]);
184                     }
185
186                     sse = sse / 2;
187
188                     // Correção de erros
189
190                     if(Math.abs(sse) > precisao) {
191
192                             //camada de saida
193
194                             grad_final = ll[numlayers-1].gradiente_bipolar(ll[numlayers-2].output(),d,n);
195
196                             //demais camadas
197
198
199                             for(int tmp = numlayers -2; tmp > 0; tmp--) {
200                                     grad_inter = ll[tmp].gradiente_bipolar(erro, erro, ll[tmp-1].output(),n);
201                             }
202
203                             grad_inter = ll[0].gradiente_bipolar(erro, erro, in, n);
204
205                     }
206                     if((epocas % 10) == 0) {
207                             System.out.println("Iteração: " + Integer.toString(epocas) + "\tSSE: " + Double.toString(s
208                     }
209                     epocas = epocas + 1;
210             }
211
212             System.out.println("Num. Iterações: " + Integer.toString(epocas) + " SSE: " + Double.toString(sse) + " Err
213     }
214
215
216     ////////////////
217     // DUMPS
218     ////////////////
219
220     public void DumpR() {
221             /*for(int tmp = 0; tmp < numlayers; tmp++) {
222                     System.out.println("================================================================");
223                     System.out.println("= Camada: " + Integer.toString(tmp));
224                     System.out.println("================================================================");
225                     ll[tmp].DumpN();
226                     System.out.println("\nSaida:");
227                     ll[tmp].DumpOut();
228             }*/
229             ll[numlayers - 1].DumpOut();
230     }
231
232     ////////////////
233     // MAIN PARA TESTES
234     ////////////////
235
236     public static void main (String args [ ]) {
237             System.out.println("Iniciando a Rede\n");
238
```

```
239                 double ti = 0.0;
240                 double te = 0.0;
241                 double ts = 0.0;
242
243                 ConjuntoTreinamento ct = new ConjuntoTreinamento();
244
245
246                 rede mlp = new rede(ct.in_tam(),ct.out_tam(),2);               // 1400 entradas, 24 saidas, 3 camadas
247
248                 mlp.confLayer(0,ct.in_tam(),ct.in_tam(),ct.in_tam());   // 1a Camada: 1400 entradas, 1400 neurons, 1 entra
249                 mlp.confLayer(1,ct.out_tam(),ct.in_tam(),ct.in_tam());          // 2a Camada: 1400 entradas, 1400 neurons,
250
251                 ////////////////////////////////////////////////////////////////
252                 // TREINAMENTO
253                 ////////////////////////////////////////////////////////////////
254
255                 ti = System.currentTimeMillis();
256
257                 mlp.treina_bipolar(ct.a.a, ct.a.a_out, 0.1, 0.000001); mlp.DumpR();
258                 mlp.treina_bipolar(ct.b.b, ct.b.b_out, 0.1, 0.000001); mlp.DumpR();
259                 mlp.treina_bipolar(ct.c.c, ct.c.c_out, 0.1, 0.000001); mlp.DumpR();
260
261                 te = System.currentTimeMillis();
262                 ts = (te - ti)/1000;
263
264                 System.out.println("Tempo Gasto: " + Double.toString(ts));
265
266                 ti = System.currentTimeMillis();
267
268                 mlp.treina_binaria(ct.aa.a, ct.aa.a_out, 0.1, 0.000001); mlp.DumpR();
269                 mlp.treina_binaria(ct.bb.b, ct.bb.b_out, 0.1, 0.000001); mlp.DumpR();
270                 mlp.treina_binaria(ct.cc.c, ct.cc.c_out, 0.1, 0.000001); mlp.DumpR();
271
272                 te = System.currentTimeMillis();
273                 ts = (te - ti)/1000;
274
275                 System.out.println("Tempo Gasto: " + Double.toString(ts));
276
277
278                 ti = System.currentTimeMillis();
279
280                 mlp.treina_bipolar(ct.a.a, ct.a.a_out, 0.3, 0.000001); mlp.DumpR();
281                 mlp.treina_bipolar(ct.b.b, ct.b.b_out, 0.3, 0.000001); mlp.DumpR();
282                 mlp.treina_bipolar(ct.c.c, ct.c.c_out, 0.3, 0.000001); mlp.DumpR();
283
284                 te = System.currentTimeMillis();
285                 ts = (te - ti)/1000;
286
287                 System.out.println("Tempo Gasto: " + Double.toString(ts));
288
289                 ti = System.currentTimeMillis();
290
291                 mlp.treina_binaria(ct.aa.a, ct.aa.a_out, 0.3, 0.000001); mlp.DumpR();
292                 mlp.treina_binaria(ct.bb.b, ct.bb.b_out, 0.3, 0.000001); mlp.DumpR();
293                 mlp.treina_binaria(ct.cc.c, ct.cc.c_out, 0.3, 0.000001); mlp.DumpR();
294
295                 te = System.currentTimeMillis();
296                 ts = (te - ti)/1000;
297
```

```
298                 System.out.println("Tempo Gasto: " + Double.toString(ts));
299
300                 ti = System.currentTimeMillis();
301
302                 mlp.treina_bipolar(ct.a.a, ct.a.a_out, 0.5, 0.000001); mlp.DumpR();
303                 mlp.treina_bipolar(ct.b.b, ct.b.b_out, 0.5, 0.000001); mlp.DumpR();
304                 mlp.treina_bipolar(ct.c.c, ct.c.c_out, 0.5, 0.000001); mlp.DumpR();
305
306                 te = System.currentTimeMillis();
307                 ts = (te - ti)/1000;
308
309                 System.out.println("Tempo Gasto: " + Double.toString(ts));
310
311                 ti = System.currentTimeMillis();
312
313                 mlp.treina_binaria(ct.aa.a, ct.aa.a_out, 0.5, 0.000001); mlp.DumpR();
314                 mlp.treina_binaria(ct.bb.b, ct.bb.b_out, 0.5, 0.000001); mlp.DumpR();
315                 mlp.treina_binaria(ct.cc.c, ct.cc.c_out, 0.5, 0.000001); mlp.DumpR();
316
317                 te = System.currentTimeMillis();
318                 ts = (te - ti)/1000;
319
320                 System.out.println("Tempo Gasto: " + Double.toString(ts));
321
322                 ti = System.currentTimeMillis();
323
324                 mlp.treina_bipolar(ct.a.a, ct.a.a_out, 0.7, 0.000001); mlp.DumpR();
325                 mlp.treina_bipolar(ct.b.b, ct.b.b_out, 0.7, 0.000001); mlp.DumpR();
326                 mlp.treina_bipolar(ct.c.c, ct.c.c_out, 0.7, 0.000001); mlp.DumpR();
327
328                 te = System.currentTimeMillis();
329                 ts = (te - ti)/1000;
330
331                 System.out.println("Tempo Gasto: " + Double.toString(ts));
332
333                 ti = System.currentTimeMillis();
334
335                 mlp.treina_binaria(ct.aa.a, ct.aa.a_out, 0.7, 0.000001); mlp.DumpR();
336                 mlp.treina_binaria(ct.bb.b, ct.bb.b_out, 0.7, 0.000001); mlp.DumpR();
337                 mlp.treina_binaria(ct.cc.c, ct.cc.c_out, 0.7, 0.000001); mlp.DumpR();
338
339                 te = System.currentTimeMillis();
340                 ts = (te - ti)/1000;
341
342                 System.out.println("Tempo Gasto: " + Double.toString(ts));
343
344
345
346         }
347
348
349 }
350
```

C.10 Arquivo RedeCluster.java

```
1
2  import mpi.*;
```

```
3
4   class RedeClusterB {
5
6           ////////////////
7           // VARIAVEIS
8           ////////////////
9
10          private static double in[], ins[], outs[], m_out[], grad, g[], m_sse, d_out[], ti, te, ts;
11
12          private static ConjuntoTreinamento ct;
13
14          public static int count;
15
16          public static RedeClusterB r;
17
18          public static String nome, iter;
19
20          public static int num, tam, tamout, nip, nop, id;
21
22
23          public int numinputs;           // Num. Entradas
24          public int numinputsneuron;     // Num. Entradas dos neuronios da camada de entrada
25          public int numoutputs;          // Num. saidas
26          public int numlayers;           // Num. Camadas
27
28          public int iter_max;            // Num. Max. de Iteracoes
29          public double nr;               // Taxa de Aprendizado
30          public double precisao;         // Precisao do Erro
31
32          public double out[];            // Saida final da rede
33          public double e[];              // Erros
34          public double erro;             // Erro Final
35          public double sse;              // Squared Sum Error = Soma dos Erros Quadraticos = sum( (e^2)/2 )
36
37          public double grad_final = 0.0;
38          public double grad_inter = 0.0;
39
40          public layer ll[];
41
42          public int epocas;
43
44          ////////////////
45          // INIT
46          ////////////////
47
48          public RedeClusterB(int ni, int no, int nl) {
49
50                  numinputs = ni;
51                  numoutputs = no;
52                  numlayers = nl;
53
54                  epocas = 0;
55
56                  out = new double[no];
57                  e = new double[no];
58                  sse = 0.6;
59
60                  ll = new layer[nl];
61
```

```
62          }
63
64
65          ////////////////
66          // FUNCOES
67          ////////////////
68
69
70          public void confLayer(int index, int nn, int ni, int nin) {
71                  ll[index] = new layer(nn, ni, nin);
72                  return;
73          }
74
75          public void input_binaria(double in[]) {
76
77                  // Feed Forward
78
79                  // 1a Camada
80                  ll[0].input_binaria(in);
81
82                  for(int tmp = 1; tmp <= numlayers - 1; tmp++) {
83                          ll[tmp].input_binaria(ll[tmp-1].output());
84                  }
85
86                  // Salva a saida da Rede
87                  out = ll[numlayers-1].output();
88
89          }
90
91          public void input_bipolar(double in[]) {
92
93                  // Feed Forward
94
95                  // 1a Camada
96                  ll[0].input_bipolar(in);
97
98                  for(int tmp = 1; tmp <= numlayers - 1; tmp++) {
99                          ll[tmp].input_bipolar(ll[tmp-1].output());
100                 }
101
102                 // Salva a saida da Rede
103                 out = ll[numlayers-1].output();
104
105         }
106
107
108         public void erro(double d[]) {
109                 sse = 0.0;
110                 for(int tmp = 0; tmp < numoutputs; tmp++) {
111                         e[tmp] = (d[tmp] - out[tmp]);
112                         sse = sse + e[tmp] * e[tmp] ;
113                 }
114                 sse = sse /  2;
115                 return;
116         }
117
118         public void printk(String s) { System.out.println(s); return; }
119         public static void printq(String s) { System.out.println(s); return; }
120
```

```
121
122         ////////////////
123         // DUMPS
124         ////////////////
125
126         public void DumpR() {
127                 /*for(int tmp = 0; tmp < numlayers; tmp++) {
128                         System.out.println("================================================================");
129                         System.out.println("= Camada: " + Integer.toString(tmp));
130                         System.out.println("================================================================");
131                         ll[tmp].DumpN();
132                         System.out.println("\nSaida:");
133                         ll[tmp].DumpOut();
134                 }*/
135                 ll[2].DumpOut();
136                 return;
137         }
138
139
140         public static void fim() throws MPIException {          MPI.Finalize(); }
141
142
143         public static void init() throws MPIException {
144
145                 // Inicializacao MPI
146
147                 String args[] = {""};
148
149                 MPI.Init(args);
150
151                 id = MPI.COMM_WORLD.Rank();
152
153                 num = MPI.COMM_WORLD.Size();
154
155                 //Inicializacoes de Variaveis
156
157                 ct = new ConjuntoTreinamento();
158
159                 tam = ct.in_tam();
160                 nip = tam / num;
161
162                 in = new double[tam];           // Num entradas
163                 ins = new double[nip];          // Num entradas / Num Procs
164
165                 tamout = ct.out_tam();
166                 nop = tamout / num;
167
168                 outs = new double[nop];
169                 m_out = new double[tamout];             // Num Procs
170                 d_out = new double[tamout];
171
172
173                 r = new RedeClusterB(nip,nop,2);                // 360 Entradas, 1 Saida, 3 Camadas
174                 r.confLayer(0,nip,nip,nip);             // 1a Camada: 360 Neurons, 360 Entradas, 1 entrada por neuronio
175                 r.confLayer(1,nop,nip,nip);             // 3a Camada: 1 Neurons, 360 Entradas, 360 entradas por neuronio
176
177                 nome = MPI.Get_processor_name() + " (" + Integer.toString(id) + "): ";
178                 iter = "";
179
```

```
180                 grad = 0;
181                 m_sse = 10;
182
183                 count = 0;
184
185         }
186
187         public static void treina_bipolar(double ii[], double o[], double n, double err_min, double it) throws MPIExceptio
188
189
190                 if(id == 0) {
191                         in = ii;
192                         d_out = o;
193                 }
194
195                 ////////////////////////////////////////
196                 // FASE FORWARD
197                 ////////////////////////////////////////
198
199                 // Distribuicao inicial dos dados (Somente uma vez)
200
201                 MPI.COMM_WORLD.Barrier();
202
203                 MPI.COMM_WORLD.Scatter(in,0,nip,MPI.DOUBLE,ins,0,nip,MPI.DOUBLE,0);
204
205                 MPI.COMM_WORLD.Barrier();
206
207                 MPI.COMM_WORLD.Scatter(d_out,0,nop,MPI.DOUBLE,outs,0,nop,MPI.DOUBLE,0);
208
209
210                 // Loop de treinamento
211
212                 r.input_bipolar(ins);
213
214                 r.erro(outs);
215
216                 if(id == 0) {
217                 System.out.println("################################################");
218                 System.out.println("## Iniciando o treinamento BINARIA ");
219                 System.out.println("## Num Proc: " + Integer.toString(num) + " Num. Inp. Proc: " + Integer.toString(nip) +
220                 System.out.println("## n: " + Double.toString(n) + " NC: " + Integer.toString(r.numlayers) + "Precisao: "
221                 System.out.println("################################################");
222
223                 }
224
225                 while(Math.abs(r.sse) > err_min && count <= it) {
226
227                         r.grad_final = 0.0;
228                         r.grad_inter = 0.0;
229
230                         if(Math.abs(r.sse) > err_min) {
231
232                                 //camada de saida
233
234                                 r.grad_final = r.ll[r.numlayers-1].gradiente_bipolar(r.ll[r.numlayers-2].output(),outs,n);
235
236                                 //demais camadas
237                                 if(r.numlayers > 2) {
238                                         for(int tmp = r.numlayers -2; tmp > 0; tmp--) {
```

```
239                                             r.grad_inter = r.ll[tmp].gradiente_bipolar(r.sse, r.grad_final, r.ll[tmp-1
240                                     }
241                             }
242
243                             r.grad_inter = r.ll[0].gradiente_bipolar(r.sse, r.grad_final, in, n);
244                     }
245
246
247                     r.input_bipolar(ins);
248
249                     r.erro(outs);
250
251                     if(id == 0 && (count % 10) == 0) {
252                             iter = "Iter: " + Integer.toString(count);
253                             printq(nome + iter + " Erro: " + Double.toString(r.sse) + " GRAD: " + Double.toString(r.gr
254                     }
255
256                     count++;
257
258             }
259
260
261             System.out.println("Numero total de iterações: " + Integer.toString(count));
262
263             MPI.COMM_WORLD.Barrier();
264
265             MPI.COMM_WORLD.Gather(r.out,0,nop,MPI.DOUBLE,m_out,0,nop,MPI.DOUBLE,0);
266
267             if(id == 0) {
268                     for(int iii = 0; iii <= m_out.length -1; iii++) {
269                             System.out.println(Double.toString(m_out[iii]));
270                     }
271             }
272     }
273
274         public static void treina_binaria(double ii[], double o[], double n, double err_min, double it) throws MPIExceptio
275
276
277             if(id == 0) {
278                     in = ii;
279                     d_out = o;
280             }
281
282             ////////////////////////////////////////
283             // FASE FORWARD
284             ////////////////////////////////////////
285
286             // Distribuicao inicial dos dados (Somente uma vez)
287
288             MPI.COMM_WORLD.Barrier();
289
290             MPI.COMM_WORLD.Scatter(in,0,nip,MPI.DOUBLE,ins,0,nip,MPI.DOUBLE,0);
291
292             MPI.COMM_WORLD.Barrier();
293
294             MPI.COMM_WORLD.Scatter(d_out,0,nop,MPI.DOUBLE,outs,0,nop,MPI.DOUBLE,0);
295
296
297             // Loop de treinamento
```

```
298
299             r.input_binaria(ins);
300
301             r.erro(outs);
302
303             if(id == 0) {
304             System.out.println("################################################");
305             System.out.println("## Iniciando o treinamento BINARIA ");
306             System.out.println("## Num Proc: " + Integer.toString(num) + " Num. Inp. Proc: " + Integer.toString(nip) +
307             System.out.println("## n: " + Double.toString(n) + " NC: " + Integer.toString(r.numlayers) + "Precisao: "
308             System.out.println("################################################");
309
310             }
311
312             while(Math.abs(r.sse) > err_min && count <= it) {
313
314                     r.grad_final = 0.0;
315                     r.grad_inter = 0.0;
316
317                     if(Math.abs(r.sse) > err_min) {
318
319                             //camada de saida
320
321                             r.grad_final = r.ll[r.numlayers-1].gradiente_binaria(r.ll[r.numlayers-2].output(),outs,n);
322
323                             //demais camadas
324                             if(r.numlayers > 2) {
325                                     for(int tmp = r.numlayers -2; tmp > 0; tmp--) {
326                                             r.grad_inter = r.ll[tmp].gradiente_binaria(r.sse, r.grad_final, r.ll[tmp-1
327                                     }
328                             }
329
330                             r.grad_inter = r.ll[0].gradiente_binaria(r.sse, r.grad_final, in, n);
331                     }
332
333
334                     r.input_binaria(ins);
335
336                     r.erro(outs);
337
338                     if(id == 0 && (count % 10) == 0) {
339                             iter = "Iter: " + Integer.toString(count);
340                             printq(nome + iter + " Erro: " + Double.toString(r.sse) + " GRAD: " + Double.toString(r.gr
341                     }
342
343                     count++;
344
345             }
346
347
348             System.out.println("Numero total de iterações: " + Integer.toString(count));
349
350             MPI.COMM_WORLD.Barrier();
351
352             MPI.COMM_WORLD.Gather(r.out,0,nop,MPI.DOUBLE,m_out,0,nop,MPI.DOUBLE,0);
353
354             if(id == 0) {
355                     for(int iii = 0; iii <= m_out.length -1; iii++) {
356                             System.out.println(Double.toString(m_out[iii]));
```

```
357                         }
358                 }
359 }
360
361
362         ////////////////
363         // MAIN
364         ////////////////
365
366
367         public static void main(String args[]) throws MPIException {
368
369                 init();
370
371
372                 ti = System.currentTimeMillis();
373                 treina_bipolar(ct.a.a, ct.a.a_out, 0.1, 0.000001, 1000);
374                 treina_bipolar(ct.b.b, ct.b.b_out, 0.1, 0.000001, 1000);
375                 treina_bipolar(ct.c.c, ct.c.c_out, 0.1, 0.000001, 1000);
376                 te = System.currentTimeMillis();
377                 ts = (te - ti)/1000;
378
379                 System.out.println("Tempo Gasto: " + Double.toString(ts));
380
381
382                 ti = System.currentTimeMillis();
383                 treina_binaria(ct.aa.a, ct.aa.a_out, 0.1, 0.000001, 1000);
384                 treina_binaria(ct.bb.b, ct.bb.b_out, 0.1, 0.000001, 1000);
385                 treina_binaria(ct.cc.c, ct.cc.c_out, 0.1, 0.000001, 1000);
386                 te = System.currentTimeMillis();
387                 ts = (te - ti)/1000;
388
389                 System.out.println("Tempo Gasto: " + Double.toString(ts));
390
391
392                 ti = System.currentTimeMillis();
393                 treina_bipolar(ct.a.a, ct.a.a_out, 0.3, 0.000001, 1000);
394                 treina_bipolar(ct.b.b, ct.b.b_out, 0.3, 0.000001, 1000);
395                 treina_bipolar(ct.c.c, ct.c.c_out, 0.3, 0.000001, 1000);
396                 te = System.currentTimeMillis();
397                 ts = (te - ti)/1000;
398
399                 System.out.println("Tempo Gasto: " + Double.toString(ts));
400
401
402                 ti = System.currentTimeMillis();
403                 treina_binaria(ct.aa.a, ct.aa.a_out, 0.3, 0.000001, 1000);
404                 treina_binaria(ct.bb.b, ct.bb.b_out, 0.3, 0.000001, 1000);
405                 treina_binaria(ct.cc.c, ct.cc.c_out, 0.3, 0.000001, 1000);
406                 te = System.currentTimeMillis();
407                 ts = (te - ti)/1000;
408
409                 System.out.println("Tempo Gasto: " + Double.toString(ts));
410
411                 ti = System.currentTimeMillis();
412                 treina_bipolar(ct.a.a, ct.a.a_out, 0.5, 0.000001, 1000);
413                 treina_bipolar(ct.b.b, ct.b.b_out, 0.5, 0.000001, 1000);
414                 treina_bipolar(ct.c.c, ct.c.c_out, 0.5, 0.000001, 1000);
415                 te = System.currentTimeMillis();
```

```
416                 ts = (te - ti)/1000;
417
418                 System.out.println("Tempo Gasto: " + Double.toString(ts));
419
420
421                 ti = System.currentTimeMillis();
422                 treina_binaria(ct.aa.a, ct.aa.a_out, 0.5, 0.000001, 1000);
423                 treina_binaria(ct.bb.b, ct.bb.b_out, 0.5, 0.000001, 1000);
424                 treina_binaria(ct.cc.c, ct.cc.c_out, 0.5, 0.000001, 1000);
425                 te = System.currentTimeMillis();
426                 ts = (te - ti)/1000;
427
428                 System.out.println("Tempo Gasto: " + Double.toString(ts));
429
430                 ti = System.currentTimeMillis();
431                 treina_bipolar(ct.a.a, ct.a.a_out, 0.7, 0.000001, 1000);
432                 treina_bipolar(ct.b.b, ct.b.b_out, 0.7, 0.000001, 1000);
433                 treina_bipolar(ct.c.c, ct.c.c_out, 0.7, 0.000001, 1000);
434                 te = System.currentTimeMillis();
435                 ts = (te - ti)/1000;
436
437                 System.out.println("Tempo Gasto: " + Double.toString(ts));
438
439
440                 ti = System.currentTimeMillis();
441                 treina_binaria(ct.aa.a, ct.aa.a_out, 0.7, 0.000001, 1000);
442                 treina_binaria(ct.bb.b, ct.bb.b_out, 0.7, 0.000001, 1000);
443                 treina_binaria(ct.cc.c, ct.cc.c_out, 0.7, 0.000001, 1000);
444                 te = System.currentTimeMillis();
445                 ts = (te - ti)/1000;
446
447                 System.out.println("Tempo Gasto: " + Double.toString(ts));
448
449
450                 fim();
451
452
453         }
454
455 }
```

Referências

BAKER, M. et al. *mpiJava: An Object-Oriented Java interface to MPI*. SCS - University of Portsmouth, 1999. Disponível em: <http://ipdps.cc.gatech.edu/1999/java/baker.pdf>. Acesso em: 17/08/2005.

BARROS, M. J. A. de S.; TEIXEIRA, H. E. G.; COELHO, P. J. M. *Clusters Beowulf*. Porto: FEUP, 2000.

BEOWULF. *Beowulf.org: The Beowulf Cluster Site*. [S.l.], 2005. Disponível em: <http://www.Beowulf.org/>. Acesso em: 20/04/2005.

BRAGA, A. d. P.; CARVALHO, A. P. d. L.; LUDERMIR, T. B. *Redes Neurais Artificiais:teoria e aplicações*. Rio de Janeiro: LTC, 2000. 262 p.

CARDOSO, D. L.; PINTO, L. A. B. *Projeto, Configuração e Utilização de Cluster Beowulf: um estude de caso.* Belém: CCET-UNAMA, 2002. 65 p.

CENAPAD. *Curso de MPI*. [S.l.], 2005. Disponível em: <http://www.cenapad.unicamp.br/servicos/treinamentos/mpi.shtml>. Acesso em: 10/04/2005.

CNPQ. *Resenha Estatística do CNPq*. [S.l.], 2005. Disponível em: <http://ftp.cnpq.br/pub/doc/aei/>. Acesso em: 25/04/2005.

DASGUPTA, P.; KEDEM, Z. M.; RABIN, M. O. Parallel processing on networks of workstations:a fault-tolerant, high performance approach. 1995. Disponível em: <http://walfredo.dsc.ufcg.edu.br/cursos/2003/parcomp20032/networkstations.pdf>. Acesso em: 05/08/2005.

FAUSETT, L. *Fundamentals of neural networks: architectures, algorithms, and applications*. Upper Saddle River: Prentice Hall, 1994. 461 p.

FORUM, M. *MPI-2: Extensions to the Message-Passing Interface*. [S.l.], 2005. Disponível em: <http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html>. Acesso em: 10/04/2005.

FORUM, M. *MPI: A Message-Passing Standard*. [S.l.], 2005. Disponível em: <http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html>. Acesso em: 10/04/2005.

FORUM, M. *MPI Fórum*. [S.l.], 2005. Disponível em: <http://www.mpi-forum.org/>. Acesso em: 20/04/2005.

GROUP, T. . *Top 500 List*. [S.l.], 2005. Disponível em: <http://www.top500.org/>. Acesso em: 25/04/2005.

HAYKIN, S. *Redes Neurais: Princípios e Prática*. 2. ed. Porto Alegre: Bookman, 2001. 900 p.

JORDAN, L. E.; ALAGHBAND, G. *Fundamentals of parallel processing*. [S.l.]: Pretince-Hall, 2002.

KANDEL, E. R.; SCHWARTZ, J. H. *Principles of Neural Science*. Nova York: Prentice-Hall, 1991.

MENDONÇA, A.; ZELENOVSKY, R. Processadores para o próximo milênio. 2005. Disponível em: <http://www.clubedohardware.com.br/artigos/993>. Acesso em: 20/06/2005.

MPI, L. *LAM/MPI Parallel Computing*. [S.l.], 2005. Disponível em: <http://www.lam-mpi.org/>. Acesso em: 10/08/2005.

MPICH. *MP-MPICH - MPI for heterogeneous Clusters*. [S.l.], 2005. Disponível em: <http://www.lfbs.rwth-aachen.de/content/mp-mpich>. Acesso em: 10/08/2005.

PIMENTEL, C.; FIALLOS, M. *Paralelização do Algoritmo "Backpropagation" em Clusters de Estações de Trabalho*. [S.l.]: Artigo apresentado no IV Congresso Brasileiro de Redes Neurais, 1999.

PIROPO, B. *Lei de Moore: até quando? - Chegando aos 90 nm*. [s.n.], 2004. Disponível em: <Disponivel em http://www.forumpcs.com.br/coluna.php?b=102249>. Acesso em: Acessado em 05/03/2005.

PROJECT, H. *mpiJava Home Page*. [S.l.], 2005. Disponível em: <http://aspen.ucs.indiana.edu/pss/HPJava/mpiJava.html>. Acesso em: 10/08/2005.

REZENDE, S. O. *Sistemas Inteligentes: Fundamentos e Aplicações*. Barueri: Manole, 2003. 479 p.

RUSSELL, S. J. *Artificial Intelligence:a modern approach*. New Jersey: Pretince-Hall, 1995.

RUSSELL, S. J.; RUSSELL, P. N. *Inteligência Artificial: Tradução da Segunda Edição*. Rio de Janeiro: Editora Elsevier, 2004.

SANCHES, M. K. *Aprendizado de Máquina Semi-Supervisionado: proposta de um algoritmo para rotular exemplos apartir de poucos exemplos rotulados*. São Carlos: ICMC-USP, 2003. 142 p.

SCHLEMER, E. *Sistemas Distribuidos*. 1 ed.. ed. Gravataí: PPGC/UFGRS, 2002. 76 p.

SEIFFERT, U. *Artificial Neural Networks On Massively Parallel Computer Hardware*. Magdeburg: ESANN 2002 - European Symposium on Artificial Neural Networks, 2002. 319-330 p.

SENGER, L. J. *Avaliação de Desempenho do PVM-W95*. São Carlos: USP, 1997. 100 p.

SILVA, E.; OLIVEIRA, A. C. Dicas para configuração de redes neurais. 2001. Disponível em: <http://www.labic.nce.ufrj.br/downloads/dicascfgrna.pdf>. Acesso em: 05/10/2005.

STALLINGS, W. *Arquitetura e Organização de Computadores*. 5 ed.. ed. São Paulo: Prentice Hall, 2002. 786 p.

STERNBERG, R. J. *Psicologia Coginitiva*. Porto Alegre: Artmed, 2000. 494 p.

SUN. *Java Home Page*. [S.l.], 2005. Disponível em: <http://java.sun.com/j2se/1.4.2/>. Acesso em: 20/09/2005.

TANENBAUM, A. S. *Organização Estruturada de Computadores*. Rio de Janeiro: LTC, 2001.

TANENBAUM, A. S. *Sistemas Operacionais Modernos*. 2 ed.. ed. São Paulo: Prentice Hall, 2003.

TEIXEIRA, J. de F. *Mentes e Máquinas: Uma introdução à ciência cognitiva*. Porto Alegre: Artes Médicas, 1998.

Índice Remissivo

Algoritmos Genéticos, 8
Arquiteturas de Cluster
    HAC, 26
    HPC, 26
Arquiteturas de Máquinas Paralelas
    Interligação Dinâmica, 27
    Interligação Estática, 27
Arquiteturas de Redes Neurais, 15

Backpropagation, 19
    Gradiente, 22
    Modo Batch, 23
    Modo Block, 23
    Modo On-Line, 23
    Taxa de Aprendizado, 23
Beowulf, 30
Bias, 11
Bibliotecas Paralelas
    MPI, 34
    PVM, 33

Classificação de Flynn
    MIMD, 29
    MISD, 29
    SIMD, 28
    SISD, 28
Cluster Beowolf, 30
Combinador Linear, 12
Communicator, 35

Eficiência, 38
Escalabilidade, 39

Funções de Ativação, 12

Granularidade, 27
Group, 35

HAC, 26
High Available Computers, 26
High Performance Computers, 26
HPC, 26

Inteligência Artificial, 7
    Algoritmos Genéticos, 8
    Lógica Fuzzy, 8
    Raciocínio Baseado em Casos, 7
    RBC, 7
    Redes Bayesianas, 8
    Sistemas Baseados em Conhecimento, 7
    Sistemas Especialistas, 7
    Tendência Conexionista, 8
    Tendência Estaticista, 8
    Tendência Evolucionista, 8
    Tendência Simbolista, 7
Interligação Dinâmica, 27
Interligação Estática, 27

Java MPI, 37

Lógica Fuzzy, 8
LAM, 37
Lei de Amdahl, 38

Medidas de Desempenho
    Eficiência, 38
    Escalabilidade, 39
    Lei de Amdhal, 38
    Speed Up, 38
MIMD, 29
MISD, 29
MPI, 34
    Communicator, 35
    Funções Básicas, 35
    Group, 35
    Rank, 35
    Rotinas de Comunicação Coletiva, 36
    Rotinas de Comunicação Ponto a Ponto, 36
MPICH, 37
mpiJava, 37
Multicomputadores
    Cluster Beowolf, 30

Neurônio Biológico, 9
Neuronios Biológicos, 10

Overfitting, 18

Padrão de Entrada, 11

Perceptron, 11
PVM, 33

Raciocínio Baseado em Casos, 7
Rank, 35
Redes Bayesianas, 8
Redes Neurais
    Arquiteturas, 15
    Bias, 11
    Combinador Linear, 12
    Frank Rosenblatt, 8
    Funções de Ativação, 12
    Hebb, 8
    McCulloh & Pitts, 8
    Misky & Papert, 8
    Neurônio Biológico, 9
    Padrão de Entrada, 11
    Perceptron, 11
    Sistema Nervoso Biológico, 9
    Valor de Limiar, 13
Rotinas de Comunicação Coletiva, 36
Rotinas de Comunicação Ponto a Ponto, 36

SIMD, 28
SISD, 28
Sistema Nervoso Biológico, 9
Sistemas Baseados em Conhecimento, 7
Sistemas Especialistas, 7
Speed Up, 38

Tecnicas de Aprendizado
    Aprendizado Não Supervisionado, 18
    Aprendizado Supervisionado, 18
    Conhecimento, 17
    Overfitting, 18
    Tipos, 18
    Underfitting, 18
Tendência Estaticista, 8
Tendência Evolucionista, 8
Tendência Simbolista, 7
Threshold, 13

Underfitting, 18

Valor de Limiar, 13