JOSÉ REINALDO LEMES JÚNIOR

UM SISTEMA DISTRIBUÍDO PARA TREINAMENTO DE REDES NEURAIS

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do Curso de Ciência da Computação para obtenção do título de Bacharel em Ciência da Computação.

LAVRAS
MINAS GERAIS – BRASIL
2008

JOSÉ REINALDO LEMES JÚNIOR

UM SISTEMA DISTRIBUÍDO PARA TREINAMENTO DE REDES NEURAIS

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do Curso de Ciência da Computação para obtenção do título de Bacharel em Ciência da Computação.

Área de Concentração:
Redes Neurais

Orientador:
Prof. Wilian Soares Lacerda

LAVRAS
MINAS GERAIS – BRASIL
2008

JOSÉ REINALDO LEMES JÚNIOR

UM SISTEMA DISTRIBUÍDO PARA TREINAMENTO DE REDES NEURAIS

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do Curso de Ciência da Computação para obtenção do título de Bacharel em Ciência da Computação.

Aprovada em ____de novembro de 2008

____
Prof.ª Marluce Rodrigues Pereira

____
Prof. Bráulio Adriano de Mello

____
Prof. Wilian Soares Lacerda
(orientador)

LAVRAS
MINAS GERAIS – BRASIL
2008

AGRADECIMENTOS:

Agradeço a Deus por tudo!

A toda minha família. E, em especial os meus pais: José Reinaldo e Gasparina, sem vocês nada seria possível.

Ao meu irmão Danilo e, aos meus amigos que sempre me deram forças.

Ao professor Wilian, obrigado pela paciência e orientação!

Ficha Catalográfica preparada pela Divisão de Processo Técnico da Biblioteca Central da UFLA

Lemes Júnior, José Reinaldo

Um Sistema Distribuído para Treinamento de Redes Neurais/ José Reinaldo Lemes Júnior. Lavras – Minas Gerais, 2008. p. 73, il.

Monografia de Graduação – Universidade Federal de Lavras. Departamento de Ciência da Computação.

1. Redes Neurais. 2. Computação Paralela. 3. Java RMI. I. LEMES JÚNIOR, J. R. II. Universidade Federal de Lavras. III. Título.

Um Sistema Distribuído para Treinamento de Redes Neurais

RESUMO

O objetivo deste trabalho foi desenvolver um sistema distribuído para realizar o treinamento de uma rede neural *multilayer perceptron* seguindo a estratégia do paralelismo do conjunto de exemplos. A implementação foi feita na linguagem Java usando Java RMI como arquitetura de objetos distribuídos. A aplicação mostrou bons resultados na tentativa de reduzir o tempo de treinamento. No entanto, o sistema ainda pode ser melhorado com a utilização de uma arquitetura melhor adaptada a computação paralela de alto desempenho.

Palavras Chave: Redes Neurais Artificiais, Computação Paralela, Java RMI

A Distributed System for Training Neural Networks

ABSTRACT

The aim of this work was a development of a distributed system for training a multilayer *perceptron* following the strategy of training set parallelism. The implementation was done in Java using Java RMI as the architecture of distributed objects. The application has shown good results in an attempt to reduce the duration of training. However, the system can still be improved with the use of a better architecture suited to parallel computing for high-performance.

Keywords: Artificial Neural Networks, Parallel Computing, Java RMI.

SUMÁRIO

LISTA DE FIGURAS

1. INTRODUÇÃO 1

1.1. Motivação 1

1.2. Objetivos 2

1.3. Estrutura do Trabalho 2

2. REDES NEURAIS 3

2.1. Introdução 3

2.2. Características das Redes Neurais 4

2.3. Aprendizado nas Redes Neurais 6

2.4. *Perceptron* 9

2.5. *Perceptron*s de Múltiplas Camadas (MLP) 11

2.6. *Backpropagation* 11

2.7. A Regra Delta 14

3. PARALELISMO EM REDES NEURAIS 17

3.1. Introdução 17

3.2. Estratégias de Paralelismo 20

3.2.1. Paralelismo de sessão de treinamento. 20

3.2.2. Paralelismo em *pipelining* 21

3.2.3. Paralelismo de nós 22

3.2.4. Neuron parallelism. 23

3.2.5. Paralelismo de pesos. 25

3.2.6. Training set parallelism. 26

4. JAVA 28

4.1. Introdução 28

4.2. Java RMI 29

4.3. Java na Computação Paralela 31

5. IMPLEMENTAÇÃO 33

5.1. Introdução 33

5.2. Descrição do Problema 34

5.3. Estratégia de Implementação 35

5.4. Interfaces da Aplicação 42

6. RESULTADOS 45

7. CONCLUSÃO 52

7.1. Considerações Finais 52

7.2. Trabalhos Futuros 52

REFERÊNCIAS BIBLIOGRÁFICAS: 54

ANEXO 56

Código Fonte 56

LISTA DE FIGURAS

Figura 2.1 Modelo de treinamento supervisionado 7

Figura 2.2 Modelo de treinamento não supervisionado 8

Figura 2.3 Um *perceptron* e uma rede de *perceptron*s 10

Figura 2.4 Rede Neural de camada única 12

Figura 2.5 Adaptação do peso da conexão Δwkj 13

Figura 2.6 *Perceptron* de múltiplas camadas 14

Figura 3.1 Network Parallelism 19

Figura 3.2 Training-Set Parallelism 20

Figura 3.3 Mapeamento para as matrizes de pesos para o *Pipelining* 21

Figura 3.4 *Pipelining* dos padrões de treinamento 22

Figura 3.5 Paralelismo de neurônios 23

Figura 3.6 Produto entre o vetor de entradas pela matriz de pesos 23

Figura 3.7 Cálculo dos termos de erro na camada oculta no paralelismo de neurônios 24

Figura 3.8 Paralelismo de pesos 25

Figura 3.9 Paralelismo do conjunto de treinamento para aprendizagem do alfabeto 26

Figura 4.1 Aplicação distribuída RMI 30

Figura 4.2 Arquitetura em camadas do Java RMI 31

Figura 5.1 Diagrama de Casos de Uso 36

Figura 5.2 Diagrama de Classes 39

Figura 5.3: Conexão entre os clientes e as *threads* do servidor 41

Figura 5.4: Interface principal 42

Figura 5.5: Interface da aplicação para criação da rede neural 43

Figura 5.6: Tela de exibição dos valores de erro através das épocas de treinamento. 44

Figura 5.7: Interface do Cliente 44

Figura 6.1: Gráfico do tempo de treinamento em relação ao número de clientes 46

Figura 6.2: Generalização do treinamento no teste da rede 1-5-1 47

Figura 6.3: Convergência do erro quadrático médio em função das épocas de treinamento 47

Figura 6.4: Resultados do treinamento seqüencial e paralelo 49

Figura 6.5: Desempenho do paralelismo de exemplos com o aumento do conj. de treinamento 50

Figura 6.6: Desempenho do paralelismo de exemplos com o aumento do tamanho da rede 50

1. INTRODUÇÃO

Redes neurais artificiais fazem parte de uma classe da inteligência artificial cujo funcionamento tenta imitar, de forma simplificada, o funcionamento de uma rede de neurônios natural (cérebro). Pelo fato de simularem o processo de aprendizagem de um cérebro, elas são capazes de solucionar alguns problemas que humanos tendem a solucionar com facilidade, como reconhecimento de padrões e controle motor, mas que são tarefas difíceis para um computador. No entanto, o alto grau de complexidade presente nas redes neurais artificiais torna sua computação difícil, e em muitas situações, seu tempo de treinamento se torna inaceitável.

Na tentativa de contornar esse problema, muitos estudos têm sido realizados com o objetivo de realizar o processo de treinamento da rede utilizando técnicas de computação paralela, baseando se no alto grau de paralelismo inerente das redes neurais, usando máquinas paralelas ou *cluster*s de computadores por exemplo.

1.1. Motivação

A motivação deste trabalho se deve ao fato da atual e crescente utilização das redes neurais na solução de problemas do mundo real.

Além disso, a observação de trabalhos correlatos onde foram feitas implementações usando estratégias de paralelismo em redes neurais mostram os benefícios da paralelização do treinamento. Muitos desses trabalhos foram feitos usando MPI (*Message Passing Interface*) ou PVM (*Parallel Virtual Machine*) como ferramentas para a comunicação entre os processadores obtendo bons resultados. Espera-se com esse trabalho realizar uma implementação usando Java RMI obtendo resultados semelhantes além de contar com a facilidade da orientação a objetos e das vantagens conhecidas de Java.

No tratamento de problemas menores, a utilização de uma rede de computadores para a computação paralela pode proporcionar bons resultados quando não se pode contar com máquinas paralelas ou multiprocessadores.

1.2. Objetivos

O principal objetivo deste trabalho é desenvolver uma aplicação distribuída baseada em uma estratégia de paralelização de redes neurais visando reduzir o tempo necessário em seu treinamento.

Para isso, é necessária a realização de um estudo sobre redes neurais artificiais, e suas estratégias de paralelização encontradas na literatura além da análise de trabalhos correlatos onde estas estratégias foram implementadas e testadas.

1.3. Estrutura do Trabalho

Será apresentada no capítulo 2, uma breve revisão de literatura sobre redes neurais artificiais, com ênfase na rede de *perceptrons* de múltiplas camadas, na qual será baseado o sistema implementado.

O capítulo 3 traz uma revisão bibliográfica sobre paralelismo de redes neurais, apresentando as estratégias conhecidas na literatura.

No capítulo 4, será apresentada uma revisão resumida sobre a invocação remota de métodos de Java (Java RMI), e a utilização de Java na computação paralela, já que foi essa a linguagem escolhida para a implementação do sistema.

A explicação da implementação do sistema proposto neste trabalho, demonstrando as estratégias adotadas na implementação, será apresentada no capítulo 5.

No capítulo 6 serão apresentados os resultados obtidos através de testes realizados com a aplicação implementada e no capítulo 7, a conclusão do trabalho.

O código fonte implementado é apresentado no anexo.

2. REDES NEURAIS

2.1. Introdução

Alguns pesquisadores se destacaram nos primeiros anos de investigações em redes neurais (1943-1958) pelos seus trabalhos pioneiros (Haykin, 1999):

McCullock & Pitts (1943) por introduzir a idéia de redes neurais como máquinas computacionais. Hebb (1949) pela postulação da primeira regra para aprendizagem auto-organizada. Rosenblatt (1958) por propor o *perceptron* como o primeiro modelo para aprendizado supervisionado.

O *perceptron* de Rosemblat é a forma mais simples de rede neural usada na classificação de padrões linearmente separáveis (padrões situados em lados opostos de um hiperplano). É constituído basicamente de um simples neurônio com pesos sinápticos ajustáveis e bias. O algoritmo usado para ajustar os parâmetros desta rede neural surgiu primeiramente em um procedimento de aprendizado desenvolvido por Rosenblatt (1958, 1962) para o seu *perceptron*. De fato, Rosenblatt provou que se os padrões usados no treinamento do *perceptron* fossem retirados de duas classes linearmente separáveis, então o algoritmo convergiria e posicionaria a superfície de decisão na forma de um hiperplano entre as duas classes. A prova da convergência do algoritmo é conhecida como *perceptron convergence theorem*. Segundo Haykin (1999), o *perceptron* constituído de um simples neurônio se limita a realizar classificação de padrões com apenas duas classes. Expandindo a camada de saída do *perceptron* incluindo mais que um neurônio, pode-se realizar a tarefa de classificação com mais de duas classes de padrões. No entanto, essas classes devem ser linearmente separáveis para que o *perceptron* trabalhe corretamente.

Um neurônio único também representa a base de um bloco funcional usado no processamento de sinais conhecido como *adaptive filter* introduzido pelo trabalho de Widrow & Hoff (1960). Também foi apresentado neste trabalho o algoritmo conhecido como *least-mean-square (LMS) algorithm*, também conhecido como *delta rule*.

O algoritmo LMS é de simples implementação e ainda altamente efetivo. *Adaptive filters* têm sido aplicados com sucesso em diversas áreas como sistemas de comunicação,

sistemas de controle, antenas, radares, sonares, sismologia e engenharia biomédica (Widrow & Stearns, 1985; Haykin, 1996).

2.2. Características das Redes Neurais

Saramasinghe (2006), define uma rede neural na prática como uma coleção de neurônios interconectados que aprendem de forma incremental seu ambiente (dados) para capturar tendências lineares e não lineares em dados complexos, de modo que proporcione prognósticos confiáveis para novas situações, mesmo contendo ruídos e informações parciais. Neurônios são as unidades básicas computacionais que realizam procesamento local de dados dentro de uma rede. Estes neurônios formam redes massiçamente paralelas, cuja função é determinada pela topologia da rede (modo em que os neurônios estão conectados entre si), a força das conexões entre neurônios e o processamento realizado por eles.

As redes neurais artificiais vistas como máquinas adaptativas podem receber a seguinte definição (Haykin, 1999):

Uma rede neural é um processador massiçamente paralelo e distribuído constituído por unidades processadoras simples, as quais possuem uma propensão natural para armazenar conhecimento experimental e torná-lo disponível para utilização. Assemelha-se ao cérebro em dois aspectos:

1. *O conhecimento é adquirido pela rede de seu ambiente através de um processo de aprendizado.*
2. *As forças de conexão entre neurônios, conhecidas como pesos sinápticos, são usadas para armazenar o conhecimento adquirido.*

Redes neurais realizam uma variedade de tarefas, incluindo predicação ou aproximação de funções, classificação de padrões, aglomeração (*clustering*) e previsão (*forecasting*). Redes neurais são excelentes em ajustar modelos em dados. Elas podem ajustar modelos não-lineares complexos arbitrários em dados multidimensionais a qualquer precisão desejada. Por conseqüência, redes neurais são consideradas aproximadores

universais. Pelo ponto de vista funcional, elas podem ser vistas como extensões de algumas técnicas multivariadas, como regressão linear múltipla e regressão não-linear (Saramasinghe, 2006).

Redes neurais são também capazes de realizar tarefas de classificação de dados e sinais (séries temporais) que envolvem arbitrariamente limites não-lineares complexos de classificação. Em situações nas quais existam subconjuntos inicialmente desconhecidos dentro de um conjunto de dados, redes neurais são úteis na aglomeração de dados semelhantes (*unsupervised clustering*), onde elas usam as propriedades internas dos dados para descobrir estruturas de *cluster*s desconhecidos. Redes neurais são capazes também de realizar a previsão em séries temporais, nas quais são previstos os próximos valores da série. Isto é realizado através da captura de padrões temporais nos dados na forma de memória passada, que está embutida no modelo. Na previsão, este conhecimento sobre o passado define o comportamento futuro.

O método tradicional para treinamento de redes neurais se baseia na modificação dos pesos sinápticos. No entanto, é possível também que a rede neural modifique sua própria topologia, motivado pelo fato que os neurônios orgânicos podem vir a morrer e novas conexões sinápticas podem surgir (Haykin, 1999).

Haykin (1999) descreve as seguintes propriedades e capacidades oferecidas pelas redes neurais:

1. *Não linearidade.* Um neurônio artificial pode ser linear ou não-linear.
2. *Mapeamento entrada-saída.* A rede neural aprende através de exemplos construindo um mapeamento entre as entradas e saídas do problema em questão.
3. *Adaptação.* As redes neurais possuem uma capacidade própria de adaptar seus pesos sinápticos às mudanças no ambiente, podendo facilmente ser treinadas novamente para tratar as mudanças menores nas condições ambientais de funcionamento. Ou seja, podem ser usadas em ambientes onde ocorrem mudanças com o tempo.
4. *Veracidade da resposta.* No contexto da classificação de padrões, uma rede neural pode ser designada para prover informação não somente sobre um padrão em particular, mas também sobre a confiança na decisão tomada. Isso pode ser útil para

rejeitar padrões ambíguos, caso eles surjam, e assim melhorar a eficiência de classificação da rede.

5. *Informação contextual.* O conhecimento é representado pela estrutura e estado de ativação da rede. Cada neurônio é potencialmente afetado pela atividade global de todos os outros neurônios na rede. Conseqüentemente, o contexto da informação é tratado com naturalidade pela rede neural.
6. *Tolerância a falha.* Uma rede neural implementada em hardware tem potencial de ser inerentemente tolerante a falhas. Isso ocorre, por exemplo, se um neurônio ou suas conexões forem danificados, ocorre uma perda na qualidade da informação armazenada na rede sobre um padrão. No entanto, devido à natureza distribuída das informações armazenadas na rede, o dano tem que ser extensivo antes que a resposta total da rede esteja degradada seriamente.
7. *Implementável usando VLSI.* A natureza massiçamente paralela de uma rede neural a torna potencialmente rápida na computação de certas tarefas. Essa mesma característica faz com que se torne interessante a sua implementação usando a tecnologia VLSI (*very-large-scale-integrated*).
8. *Uniformidade de análise e design.* Basicamente, a mesma notação é usada em todos os domínios envolvendo aplicações de redes neurais.
9. *Analogia neurobiológica.* A modelagem das redes neurais artificiais é motivada pela analogia com o cérebro, o qual é uma prova viva que o processo paralelo tolerante às falhas é não somente fisicamente possível como também rápido e poderoso. Neurobiólogos vêem as redes neurais artificiais como uma ferramenta para interpretação de fenômenos neurobiológicos.

2.3. Aprendizado nas Redes Neurais

Um dos atributos mais significantes das redes neurais é sua habilidade em aprender através da interação com o ambiente ou com uma fonte de informação. Aprendizado em uma rede neural é normalmente realizado por meio de um procedimento adaptativo, chamado de regra ou algoritmo de aprendizado, através do qual os pesos da rede são ajustados de forma incremental com intuito de melhorar através do tempo uma medida de desempenho predefinida (Hassoun, 1995).

No contexto das redes neurais artificiais, o processo de aprendizagem é visto como um processo de otimização. Mais precisamente, o processo de aprendizado pode ser visto como uma busca em um espaço de parâmetros multidimensionais, o qual gradualmente otimiza uma função objetivo predefinida.

Existem três métodos básicos de treinamento para redes neurais. Em ordem crescente de autonomia, eles são denominados como treinamento supervisionado, treinamento reforçado e treinamento não supervisionado. No treinamento supervisionado (também conhecido como treinamento associativo), cada padrão de entrada do ambiente de treinamento está associado com um padrão de saída desejada específico. Geralmente, os pesos são sintetizados gradualmente, e a cada passo do processo de treinamento eles são atualizados de forma que seja reduzida a diferença entre a saída da rede e a saída desejada correspondente. Este tipo de treinamento é análogo a um estudante orientado por um professor. Como mostrado na figura 2.1, o sistema de treinamento é exposto ao ambiente, o qual é representado por vetores de valores característicos. Este vetor é apresentado também a um professor que, baseado na experiência, determina a resposta desejada. A resposta desejada é então usada para criar um sinal de erro que adapta os pesos do sistema de aprendizagem. Portanto, cada vetor de entrada possui um vetor de saída desejado, que é usado no treinamento da rede. O preceito importante é que o treinamento supervisionado requer uma entrada e a saída desejada correspondente. Em muitas situações o treinamento supervisionado não é nem prático nem possível (Priddy & Keller, 2005).

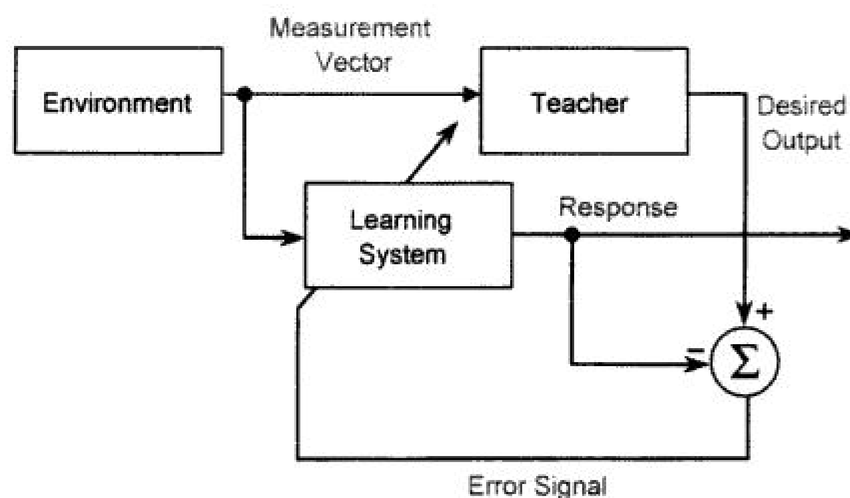

Figura 2.1: Modelo de treinamento supervisionado

Fonte: (Priddy & Keller, 2005).

Redes neurais com aprendizado supervisionado são aproximadores parcimoniosos, que podem servir como modelos estáticos (*feedforward neural networks*) ou como modelos dinâmicos (*recurrent neural networks*). Elas podem ser classificadores de alta precisão, no entanto, no campo da classificação ou reconhecimento de padrões, a representação dos padrões a serem reconhecidos é geralmente crucial para o bom desempenho do sistema como um todo (Dreyfus, 2005).

O treinamento não supervisionado envolve a aglomeração ou detecção de similaridades entre padrões não rotulados de um dado conjunto de treinamento. A idéia aqui é otimizar (maximizar ou minimizar) algum critério para melhorar o desempenho da rede em termos da atividade das saídas das unidades da rede. Procura-se a convergência dos pesos e das saídas da rede em representações que capturem as regularidades estatísticas dos dados de entrada (Hassoun, 1995).

O modelo do treinamento não supervisionado é similar ao modelo do treinamento supervisionado, porém difere no fato que nenhum "professor" é empregado no processo de treinamento, como mostrado na figura 2.2. É análogo a um estudante aprendendo por conta própria.

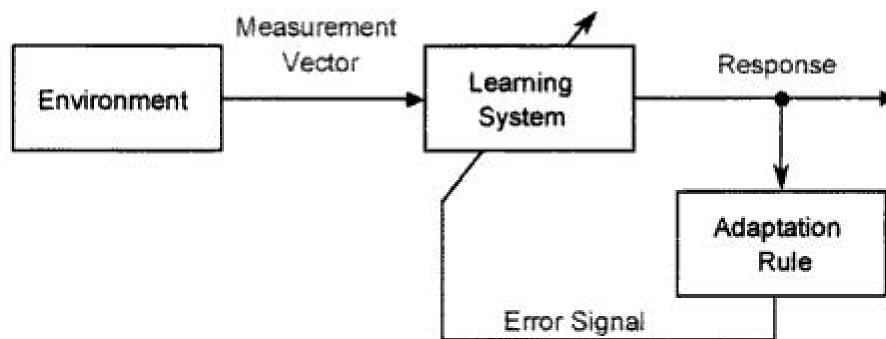

Figura 2.2 Modelo de treinamento não supervisionado

Fonte: (Priddy & Keller, 2005).

O processo de aprendizado não supervisionado possui um conjunto de regras de adaptação que conduzem o comportamento geral do treinamento. O modelo do treinamento não supervisionado consiste do ambiente, representado por um vetor de valores. Este vetor de valores é apresentado ao sistema de aprendizagem e uma resposta do sistema é obtida.

Baseado na resposta do sistema e na regra de adaptação empregada, os pesos do sistema são ajustados para obter o desempenho desejado. A regra de adaptação no treinamento não supervisionado realiza o papel de gerar o sinal de erro que o professor realiza no sistema de treinamento supervisionado. Desta forma, o comportamento do sistema de aprendizado não supervisionado depende muito da regra de adaptação usada para controlar quais pesos serão ajustados.

Já o aprendizado reforçado envolve a atualização dos pesos da rede em resposta a um sinal "avaliativo" do professor; isto difere do aprendizado supervisionado onde este sinal do professor é a "resposta correta". Em outras palavras, quando a rede retorna valores de saída incorretos, ao invés de ser fornecida a resposta correta, a rede é informada apenas que houve um erro. As regras do aprendizado reforçado podem ser vistas como mecanismos estocásticos de busca que tentam maximizar a probabilidade de reforços externos positivos para um dado conjunto de treinamento (Hassoun, 1995).

Este trabalho dará maior ênfase ao treinamento supervisionado e às redes de retropropagação.

2.4. *Perceptron*

As raízes do *perceptron* estão no domínio linear. O *perceptron* é o classificador mais simples e ainda poderoso, provendo a separabilidade linear das classes de padrões de exemplos. No entanto, existem vários problemas não lineares na vida real, e o *perceptron* foi substituído por neurônios e redes neurais mais sofisticados e poderosos. Porém, redes neurais usadas atualmente ainda possuem traços dos *perceptron*s como os *perceptron*s multicamadas com camadas ocultas de neurônios com funções de ativação sigmóides.

O *perceptron* foi um dos primeiros elementos processantes capaz de aprender. Na época de sua invenção o problema do aprendizado era uma tarefa difícil e não solucionada, e se buscava uma idéia para adaptação dos pesos de forma autônoma usando pares de dados. O aprendizado era um paradigma iterativo de aprendizagem supervisionada (Kecman, 2001).

No esquema de aprendizagem supervisionada, é escolhido um vetor de pesos iniciais aleatórios $\mathbf{w}_1$, e é apresentado ao *perceptron* um par de dados escolhido

aleatoriamente do vetor de entrada $\mathbf{x}_1$ e a saída desejada $\mathbf{d}_1$. O algoritmo de aprendizado do *perceptron* se baseia em uma regra de correção de erro que altera os pesos de forma proporcional ao erro $\mathbf{e}_1=\mathbf{d}_1-\mathbf{o}_1$ entre a saída atual $\mathbf{o}_1$ e a saída desejada $\mathbf{d}_1$. Após ser calculado o novo vetor de pesos de acordo com a seguinte regra:

$$\mathbf{w}_2 = \mathbf{w}_1 + \Delta\mathbf{w}_1 = \mathbf{w}_1 + \eta(\mathbf{d}_1 - \mathbf{o}_1)\,\mathbf{x}_1,$$

o próximo par de dados escolhido aleatoriamente do conjunto de dados e todo o processo é repetido. A constante η é chamada taxa de aprendizado, a qual determina a magnitude da variação de peso $\Delta\mathbf{w}$, mas não sua direção. No *perceptron* clássico, a taxa de aprendizado η não tem grande impacto no aprendizado, mas é uma parte importante em esquemas mais sofisticados de treinamento baseados na correção do erro.

A figura 2.3 exibe um *perceptron* e uma rede de *perceptron*s. O esquema de computação do *perceptron* da figura 2.3 é simples. Dado um vetor de entrada $\mathbf{x}$, é calculado o somatório dos pesos pelas entradas,

$$u = \sum_{i=1}^{n+1} w_i x_i = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + w_{n+1} x_{n+1}$$

e produzido uma saída de +1 se u é positivo, caso contrário, resulta uma saída de -1. A última entrada de $\mathbf{x}$, $x_{n+1}=+1$ é uma constante chamada *bias*.

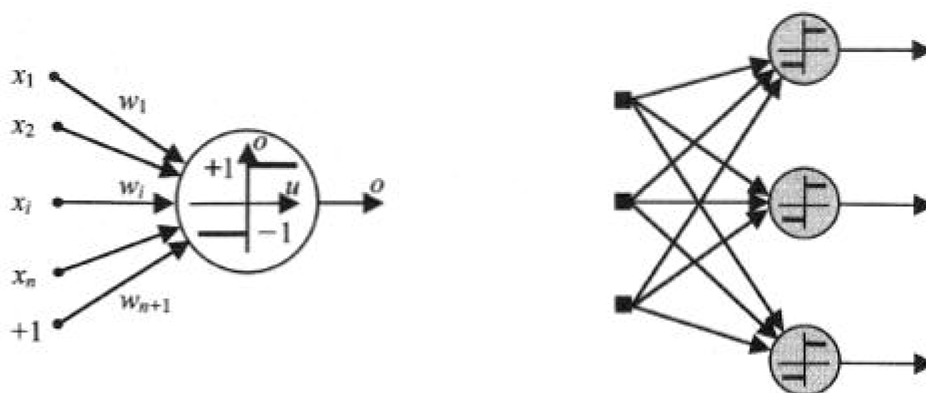

Figura 2.3 Um *perceptron* e uma rede de *perceptron*s

Fonte: (Kecman, 2001)

2.5. *Perceptron*s de Múltiplas Camadas (MLP)

Segundo Kecman (2001), redes neurais genuínas são aquelas com pelo menos duas camadas de neurônios, sendo uma camada oculta e uma camada de saída, onde os neurônios da camada oculta possuam funções de ativação não-lineares e diferenciáveis. As funções não-lineares em uma camada oculta fazem das redes neurais aproximadores universais. Assim, a não-linearidade das funções de ativação resolve o problema da representação dos *perceptron*s de camada única. O fato das funções de ativação serem diferenciáveis torna a tarefa de aprendizado não-linear possível.

Aqui, a camada de entrada não é tratada como uma camada de unidades neurais processadoras. Geralmente, nenhum processamento ocorre na camada de entrada, e apesar de se parecer uma camada na sua aparência gráfica, não se trata de uma camada de neurônios. Os neurônios da camada oculta podem ser lineares (para problemas de regressão), ou podem ser funções de ativação sigmóides (geralmente para tarefas de classificação ou reconhecimento de padrões).

A capacidade mais importante das redes neurais pode ser introduzida pelo algoritmo mais elementar baseado no gradiente descendente, o algoritmo de retro-propagação do erro.

2.6. *Backpropagation*

A idéia básica por trás do algoritmo *backpropagation* é que os termos de erro para os neurônios da camada oculta são calculados pela retro-propagação dos termos de erro dos neurônios da camada de saída. O *backpropagation* continua sendo o algoritmo de aprendizado mais comumente utilizado nos campos da computação flexível (*soft computing*) (Kecman, 2001).

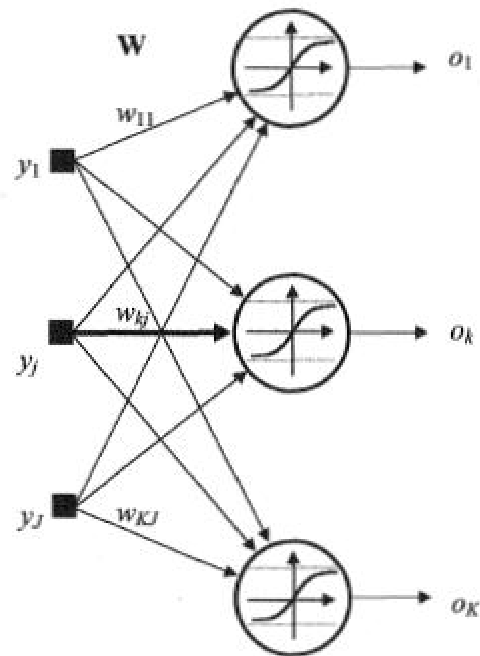

Figura 2.4 Rede Neural de camada única

Fonte: (Kecman, 2001)

Considerando a rede neural apresentada na figura 2.4, a função de erro para esta rede, tendo *K* neurônios na camada de saída e *P* pares de treinamento, é:

$$E = \frac{1}{2}\sum_{p=1}^{P}\sum_{k=1}^{K}\left(d_{pk} - o_{pk}\right)^2$$

A equação representa o erro total ao longo de todos os padrões de treinamento (o primeiro somatório) e todos os neurônios da camada de saída (o segundo somatório). Tipicamente, o algoritmo *backpropagation* atualiza os pesos de forma *on-line*, e neste caso o primeiro somatório deve ser retirado. O *backpropagation* é um método de otimização de primeira ordem que usa a técnica do gradiente descendente para ajuste dos pesos. Desta forma, uma mudança de um determinado peso será na direção de um gradiente negativo, e a cada passo da iteração será calculado como:

$$\Delta w_{kj} = \eta \frac{\partial E}{\partial w_{kj}}$$

A derivação aqui é da regra de treinamento para adaptação de pesos no modo *on-line*. Assim, o subscrito p é omitido durante a derivação. O sinal de entrada u_k para cada neurônio da camada de saída ($k=1,..., K$) é dado como:

$$u_k = \sum_{j=1}^{J} w_{kj} y_j$$

A equação da mudança de peso pode ser escrita como:

$$\Delta \mathrm{w}_{kj} = -\eta \frac{\partial E}{\partial w_{kj}} = \eta \delta_{ok} y_j$$

A expressão para o termo de erro δ_{ok} é:

$$\delta_{ok} = (d_k - o_k) f'(u_k)$$

Os ajustes dos pesos podem ser calculados da seguinte forma:

$$w_{kj} = w_{kj} + \Delta w_{kj} = w_{kj} + \eta (d_k - o_k) f'(u_k) y_j$$

$$w_{kj} = \eta \delta_{ok} y_j$$

Esta é a expressão geral para o cálculo das mudanças de peso entre os neurônios da camada oculta e os neurônios da camada de saída. A representação gráfica para a mudança do peso que conecta o j-ésimo neurônio da camada oculta com o k-ésimo neurônio da camada de saída é mostrado na figura 2.5.

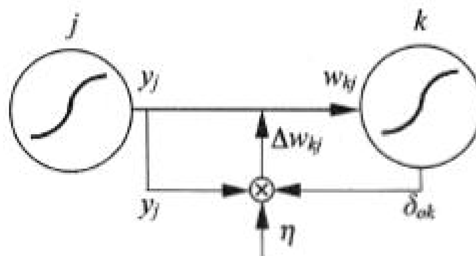

Figura 2.5 Adaptação do peso da conexão Δw_{kj}

As funções de ativação mais comuns são a função logística unipolar:

$$o = \frac{1}{1 + e^{-u}}$$

E a função sigmóide bipolar:

$$o = \frac{2}{1 + e^{-u}} - 1$$

2.7. A Regra Delta

O aprendizado com o *backpropagation* envia os dados através da rede em uma direção, e a percorre modificando os pesos na direção oposta. Quando houver mais camadas ocultas, cada camada é composta por neurônios que recebem entradas vindas de camadas anteriores e enviam as saídas aos neurônios da camada sucessiva. A mais simples estrutura é uma rede com uma camada oculta como mostrado na figura 2.6.

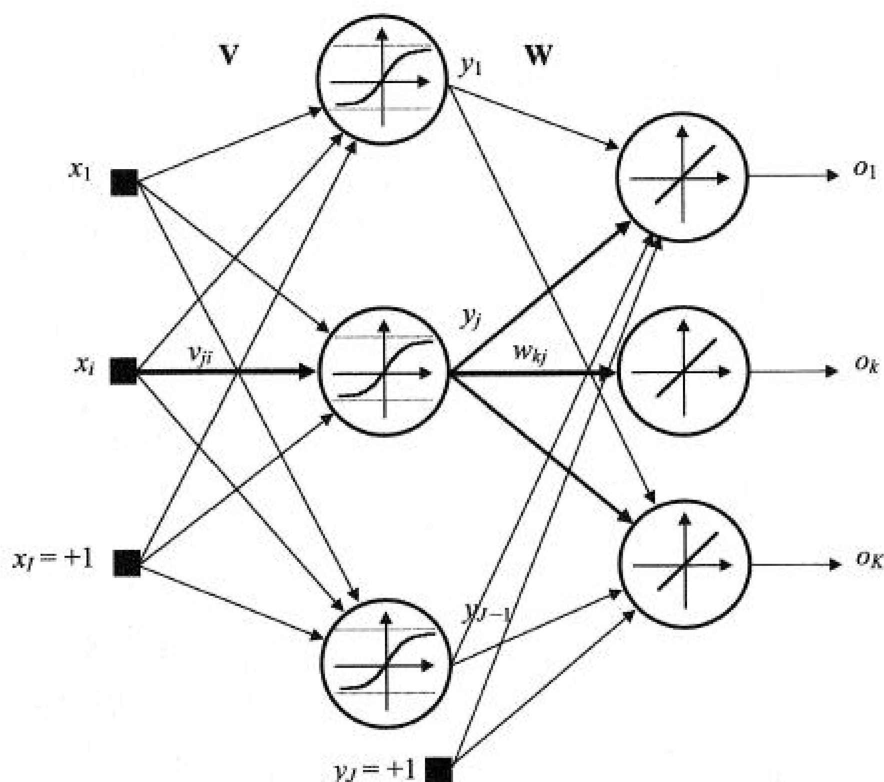

Figura 2.6 *Perceptron* de múltiplas camadas

Fonte: (Kecman, 2001)

A derivação da regra de aprendizado ou da equação para as mudanças de peso Δv_{ji} de qualquer neurônio da camada oculta é o gradiente de primeira ordem.

$$\Delta v_{ji} = -\eta \frac{\partial E}{\partial v_{ji}} = \eta \delta_{yi} x_i, \qquad j = 1, \dots, J-1, \qquad i = 1, \dots, I.$$

onde:

$$\delta_{yi} = -\frac{\partial E}{\partial v_{ji}}, \qquad j = 1, \dots, J-1.$$

O *j-ésimo* nó na figura 2.6 corresponde ao termo de bias $y_j = +1$ e não há pesos indo a este "neurônio". Por isso o índice j na equação acima termina em J -1.

O problema neste ponto é calcular o termo de erro δ_{yi}. Este passo é o mais importante na regra delta generalizada: a derivação da expressão para δ_{yi} foi a maior descoberta no procedimento de aprendizagem em redes neurais. A derivação desta expressão é explicada em detalhes em Haykin (1999).

A expressão final para δ_{yi} é:

$$\delta_{yi} = f_j'(u_j) \sum_{k=1}^{K} \delta_{ok} w_{kj}$$

E a expressão para atualização dos pesos é a seguinte:

$$\Delta v_{ji} = \eta f_j'(u_j) x_i \sum_{k=1}^{K} \delta_{ok} w_{kj}, \qquad j = 1, \dots, J-1, \qquad i = 1, \dots, I.$$

A equação acima é a mais importante na regra delta generalizada. Ela explica como são feitas as atualizações dos pesos das camadas ocultas.

As derivadas das funções de ativação dos neurônios nas camadas ocultas ou na camada de saída, necessárias para o cálculo do termo de erro δ, podem ser escritas em função da saída do neurônio:

$f'(u) = (1 - o)o$ (para a função unipolar),

$f'(u) = {}^{1}/_{2}\,(1 - o^2)$ (para a função bipolar),

Onde, para os neurônios na camada de saída $o = y$.

3. PARALELISMO EM REDES NEURAIS

3.1. Introdução

Uma rápida revisão das equações padrões usadas na descrição do *backpropagation* revelam dois graus evidentes de paralelismo em uma rede neural. Primeiro, há processamento paralelo nos neurônios de cada camada, e segundo, há paralelismo no processamento dos vários exemplos de treinamento. Um terceiro, e menos óbvio, aspecto do *backpropagation* possível de paralelismo vem do fato que as fases *forward* e *backward* de diferentes padrões de treinamento podem ser processadas em paralelo (Gironés & Salcedo, 1999).

Existem quatro estratégias que podem ser usadas para paralelizar de forma eficiente uma rede neural em máquinas massiçamente paralelas especialmente configuradas para tal finalidade: Paralelismo de sessão de treinamento*,* paralelismo de exemplos*,* paralelismo de nós e paralelismo de pesos. Cada uma destas estratégias representa um nível diferente de granularidade (Nordstrom & Svensson, 1992).

Para cada uma destas estratégias existem dois fatores que influenciam no tempo de processamento da aplicação paralela: o tempo gasto nas computações e o tempo gasto na troca de mensagens entre os processadores.

Segundo Saratchandran *et al*. (2000), existem dois paradigmas principais sobre paralelismo em redes neurais conhecidos como paralelismo baseado na rede (*Network-based parallelism*) e paralelismo de exemplos de treinamento (*Training-set parallelism*). O paralelismo de exemplos de treinamento possui menos trocas de mensagens entre os processadores, mas usa o *bacpropagation-batch* no aprendizado, o que pode acarretar em uma convergência global mais lenta para alguns problemas. Por outro lado, paralelismo baseado na rede introduz mais comunicação, porém, usa a forma padrão do *backpropagation* (*on-line*) que resulta em uma convergência mais rápida especialmente para grandes conjuntos de treinamento que possuem informações redundantes. Além disso, paralelismo baseado na rede é a única opção quando o conjunto de treinamento completo

não está disponível no inicio do processo de aprendizagem e uma adaptação contínua do fluxo de padrões de treinamento for necessária (Saratchandran, Sundrarajan, & Foo, 2000).

O aprendizado em *batch*, embora lento na convergência global, é o método escolhido para muitas aplicações especialmente quando é necessário um mapeamento de alta precisão (Forti, 1991).

De acordo com paradigma do paralelismo baseado na rede, a rede é divida entre os processadores de forma que cada processador deverá simular uma porção da rede ao longo de todo o conjunto de treinamento. Existem essencialmente duas formas para divisão da rede seguindo este paradigma. Uma é dividir as operações algébricas realizadas pela rede durante as fases *forward* e *backward* do algoritmo, e a outra é dividir a topologia da rede. Paralelismo baseado na rede pode ser usado tanto na fase de treinamento quanto na fase de reconhecimento (Saratchandran, Sundrarajan, & Foo, 2000).

A divisão algébrica da rede faz uso do fato de que vários cálculos nas fases *forward* e *backward* do algoritmo *backpropagation* podem ser expressas como operações algébricas sobre vetores e matrizes. Estas operações podem ser representadas na forma de um gráfico direcionado e então serem mapeadas sobre um vetor de elementos processadores. Esta técnica possui um paralelismo bem granulado e é bem adaptado para implementação em arranjos sistólicos de hardware.

Na divisão topológica, a rede neural é fatiada e distribuída entre os processadores os quais simulam uma fatia da rede ao longo de todo conjunto de treinamento. A forma mais conhecida que aborda esta técnica propõe a divisão da rede em fatias verticais, na qual cada processador fica com um subconjunto de neurônios de todas as camadas como ilustrado na Figura 3.1. Esta técnica possui um grau médio de granularidade de paralelismo, o que significa que é bem adaptável à implementação em máquinas paralelas como *transputers* e processadores de sinal digital (DSPs).

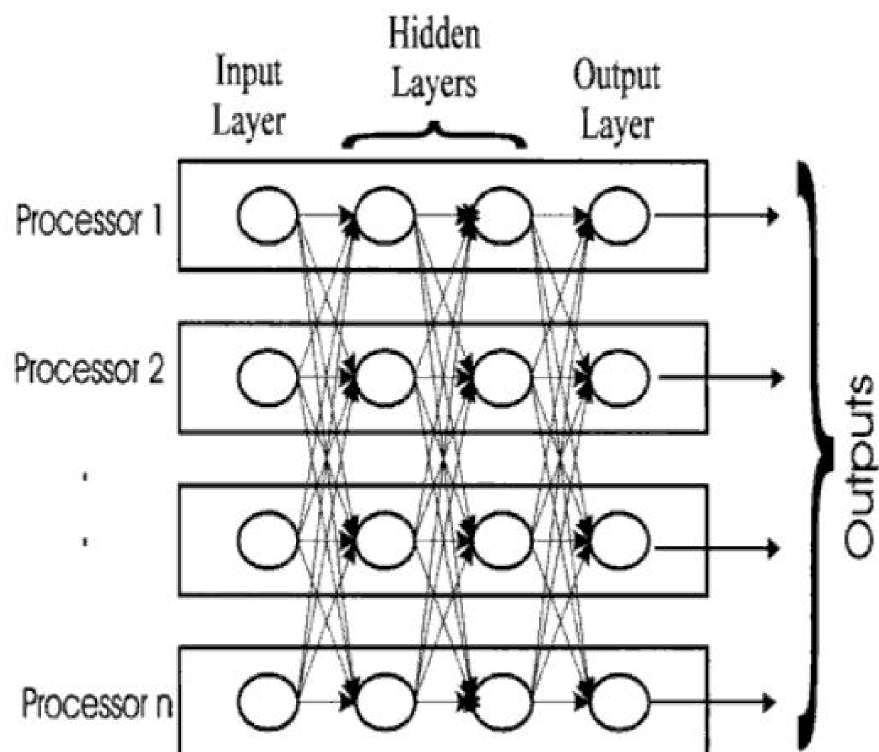

Figura 3.1: Network Parallelism

Fonte: (Saratchandran, Sundrarajan, & Foo, 2000).

Paralelismo de exemplos de treinamento (*Training-set parallelism*) envolve a distribuição dos exemplos de treinamento ao longo dos processadores, através da divisão do conjunto de treinamento em subconjuntos que são atribuídos a cada processador os quais mantêm uma cópia completa de toda a rede neural como mostrado na Figura 3.2. A única comunicação necessária para este método ocorre quando os pesos são atualizados. Um ponto crucial no paralelismo de conjunto de treinamento é como distribuir o conjunto de treinamento de forma ótima para que o tempo de treinamento de uma época seja mínimo. Esta técnica possui um grau de granularidade de paralelismo menor e é, portanto, admissível a implementação na maioria das máquinas MIND (*Multiple Instructions on Multiple Data*) disponíveis.

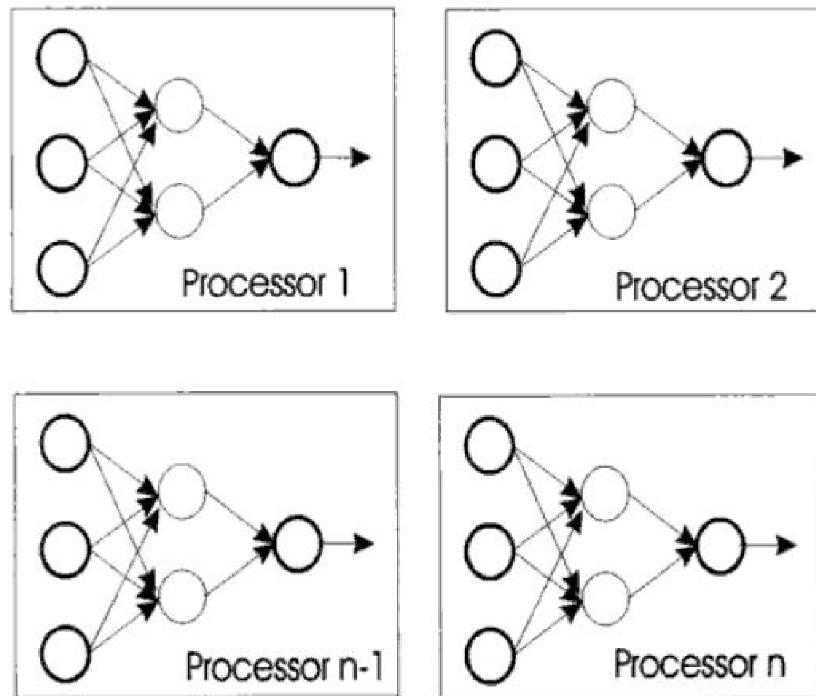

Figura 3.2: Training-Set Parallelism

Fonte: (Saratchandran, Sundrarajan, & Foo, 2000).

3.2. Estratégias de Paralelismo

3.2.1. Paralelismo de Sessão de Treinamento.

Assim como todos os algoritmos baseados no gradiente descendente, o treinamento de uma rede do tipo *backpropagation* pode consumir várias tentativas devido à propensão para ficar preso em mínimos locais. Isto requer que a rede seja reiniciada em um novo estado inicial e o treinamento seja repetido. Desenvolvendo uma cópia do *backpropagation* para cada processador e inicializando cada instância da rede em um diferente estado, elas podem ser treinadas simultaneamente com uma das instancias encontrando a melhor solução.

Paralelismo de sessão de treinamento não requer nenhuma troca de mensagens entre os processos, oferecendo teoricamente um perfeito *speedup*. Além disso, pelo fato da implementação serial poder ser usada, nenhuma implementação especial é necessária.

3.2.2. Paralelismo em *Pipelining*

Consiste em calcular cada camada em diferentes processos. Por exemplo, enquanto a camada de saída calcula as saídas e os valores de erro para o padrão de treinamento atual, o processo da camada escondida processa o próximo padrão de treinamento. As fases *forward* e *backward* podem também ser paralelizadas em *pipeline. Pipelining* requer atualização atrasada dos pesos, ou seja, a atualização dos pesos deve ocorrer após a passagem de todo ou parte do conjunto de treinamento. No *pipelining*, os pesos de diferentes camadas são processados em processos diferentes como mostrado na Figura 3.3. A Figura 3.4 mostra um exemplo de *pipelining*. Primeiramente, o processo responsável pela camada oculta calcula os valores de saída do padrão de treinamento A. O processo responsável pela camada de saída lê os valores e calcula os valores de saída e de erro de A. O processo da camada oculta simultaneamente processa o próximo padrão de treinamento (B). Então, este lê o erro da camada oculta para A, e ambos os processos acumulam as mudanças de peso para A. Neste método ocorre uma intercalação entre as fases forward e backward do treinamento (Torresen & Landsverk, 1998).

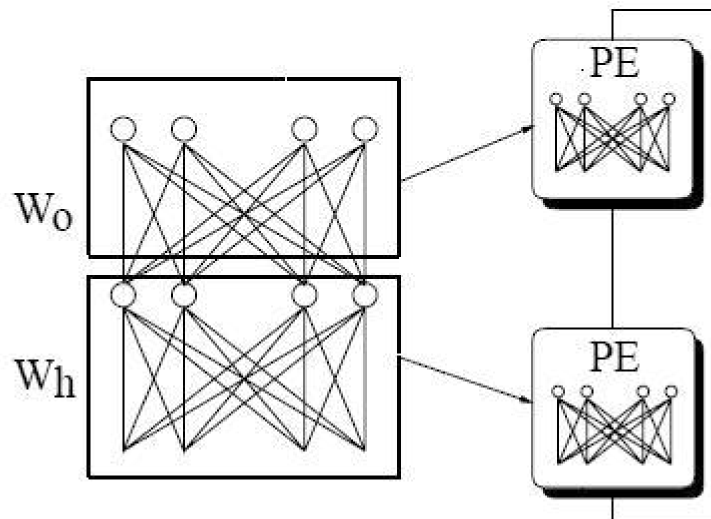

Figura 3.3 Mapeamento para as matrizes de pesos para o *Pipelining*

Fonte: (Torresen & Landsverk, 1998).

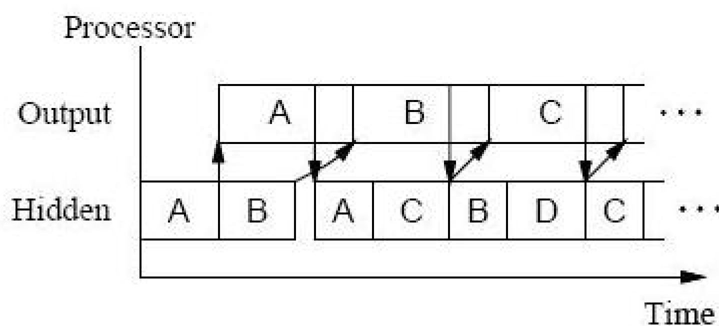

Figura 3.4 *Pipelining* dos padrões de treinamento

Fonte: (Torresen & Landsverk, 1998).

3.2.3. Paralelismo de Nós

O paralelismo de nós faz uso do paralelismo natural implícito da natureza distribuída das redes neurais. Na forma mais simples do paralelismo de nós, cada

processador fica responsável pelo cálculo da ativação de um simples neurônio, no entanto, isto geralmente não é nem prático nem vantajoso.

Segundo Torresen & Landsverk (1998), o paralelismo de nós possui duas subcategorias de paralelismo de redes: paralelismo de neurônios e paralelismo de pesos. A seguir, estas subcategorias são explicadas.

3.2.4. Neuron parallelism.

A forma mais comum de paralelizar uma rede *feed-forward* é usando o paralelismo de neurônios. A Figura 3.5 demonstra o princípio do paralelismo de neurônios para um exemplo com três elementos processadores.

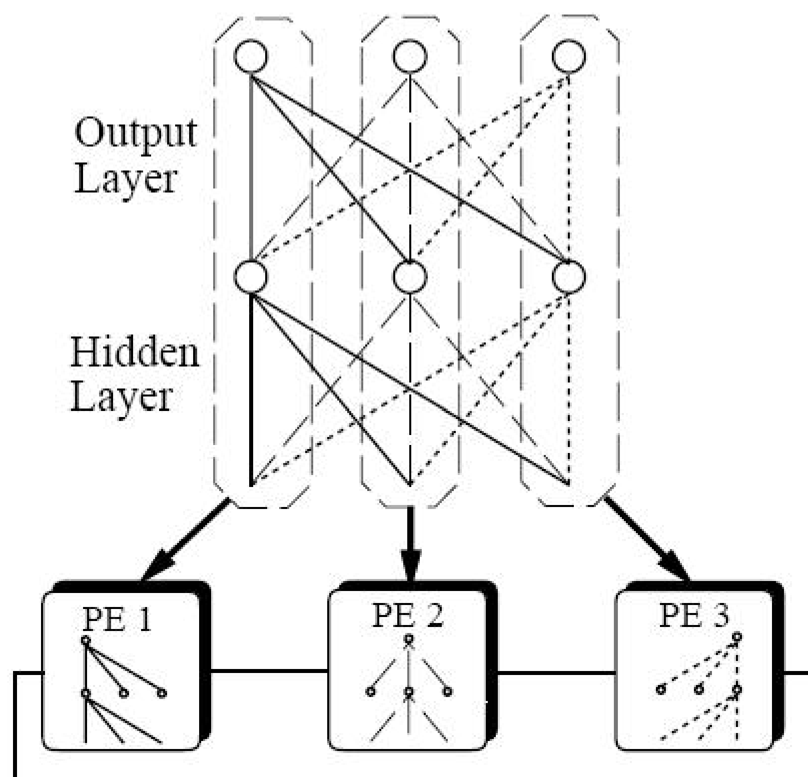

Figura 3.5 Paralelismo de neurônios.

Fonte: (Torresen & Landsverk, 1998).

Todas as ligações sinápticas que chegam a um neurônio da camada oculta ou de saída são mapeadas para cada processador. Ou seja, cada processador grava todos os pesos que chegam ao neurônio atribuído ao processador. O fatiamento da rede corresponde em armazenar uma linha da matriz de pesos em cada processador. As saídas da rede são calculadas através do produto entre matrizes e vetores.

$$\begin{bmatrix} y_{L,1} \\ \hline y_{L,2} \\ \hline y_{L,3} \end{bmatrix} = \begin{bmatrix} w_{L,11} & w_{L,12} & w_{L,13} \\ \hline w_{L,21} & w_{L,22} & w_{L,23} \\ \hline w_{L,31} & w_{L,32} & w_{L,33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Figura 3.6 produto entre o vetor de entradas pela matriz de pesos

Fonte: (Torresen & Landsverk, 1998).

Primeiramente, cada processador calcula os valores de entrada em um neurônio que lhe pertence. Então, cada processador informa os valores calculados aos outros processadores e continua com a computação dos valores dos neurônios de saída. O erro na camada oculta é obtido baseado no erro da saída da rede. Isto pode ser representado como um produto entre vetor e matriz.

$$\begin{bmatrix} \delta'_{h,1} & | & \delta'_{h,2} & | & \delta'_{h,3} \end{bmatrix} = \begin{bmatrix} \delta_{o,1} \\ \hline \delta_{o,2} \\ \hline \delta_{o,3} \end{bmatrix} \begin{bmatrix} w_{o,11} & w_{o,12} & w_{o,13} \\ \hline w_{o,21} & w_{o,22} & w_{o,23} \\ \hline w_{o,31} & w_{o,32} & w_{o,33} \end{bmatrix}$$

Figura 3.7. Cálculo dos termos de erro na camada oculta no paralelismo de neurônios.

Fonte: (Torresen & Landsverk, 1998).

Se o número de neurônios em uma camada é maior que o número de processadores, cada processador fica responsável por mais que um neurônio de cada camada.

3.2.5. Paralelismo de pesos.

É a solução paralela de maior granularidade considerada por Nordstrom & Svensson (1992). Nesta estratégia, a entrada de cada sinapse é calculada em paralelo para cada neurônio, e as entradas da rede são somadas por meio de algum esquema de comunicação adaptado.

Ao invés de mapear as linhas das matrizes de pesos em cada processador, são mapeadas as colunas. No paralelismo de pesos cada processador calcula uma soma parcial das saídas dos neurônios como indicado na Figura 3.8.

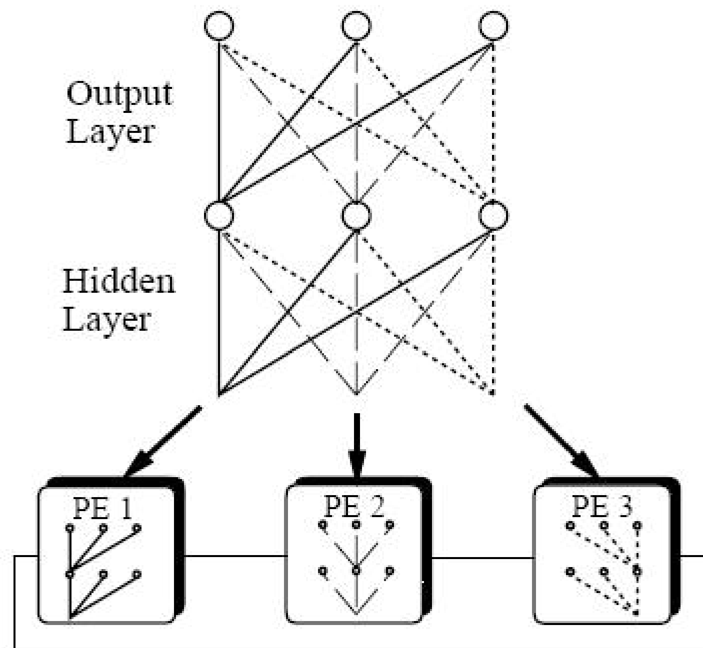

Figura 3.8 Paralelismo de pesos

Fonte: (Torresen & Landsverk, 1998).

A computação é mais granulada que no paralelismo de neurônios. Os resultados obtidos em cada processador devem ser adicionados e transmitidos a todos os processadores antes que a próxima camada seja computada. A vantagem é que os erros na camada oculta podem ser calculados sem comunicação.

$$\left[\begin{array}{c|c|c} \delta_{h,1} & \delta_{h,2} & \delta_{h,3} \end{array}\right] = \left[\begin{array}{c} \delta_{o,1} \\ \delta_{o,2} \\ \delta_{o,3} \end{array}\right] \left[\begin{array}{c|c|c} w_{o,11} & w_{o,12} & w_{o,13} \\ w_{o,21} & w_{o,22} & w_{o,23} \\ w_{o,31} & w_{o,32} & w_{o,33} \end{array}\right]$$

Figura 2.8. Cálculo do termo de erro na camada oculta no paralelismo de pesos.

Fonte: (Torresen & Landsverk, 1998).

O paralelismo de pesos não fornece nenhuma capacidade adicional em relação ao paralelismo de neurônios e aumenta significativamente o número de pequenas mensagens o que torna essa estratégia pouco interessante para implementação em um *cluster* de computadores (Pethick, Liddle, Werstein, & Huang, 2003).

3.2.6. Paralelismo de Exemplos.

O paralelismo de exemplos (*training set parallelism*, *exemplar parallelism* ou *data parallelism*) utiliza a população de treinamento como fonte de paralelismo. Cada processo determina as mudanças de peso para um subconjunto disjunto do conjunto de treinamento total. As mudanças são combinadas e aplicadas à rede neural no fim de cada época.

Um exemplo é dado na Figura 3.9 para o treinamento do alfabeto.

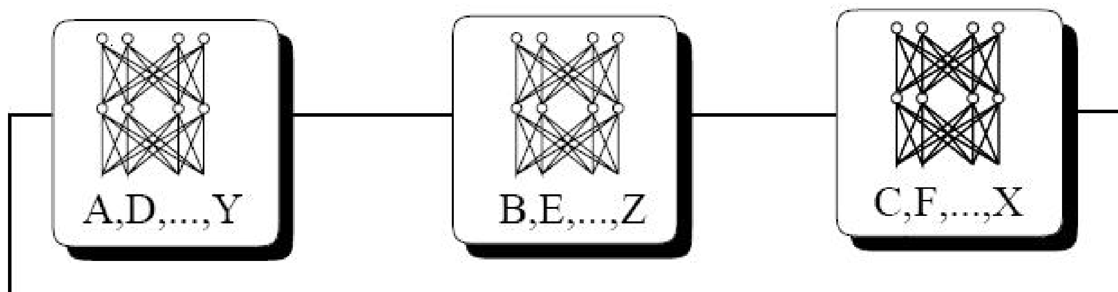

Figura 3.9. Paralelismo do conjunto de treinamento para aprendizagem do alfabeto.

Fonte: (Torresen & Landsverk, 1998).

Cada processador possui uma cópia local de toda a matriz de pesos e acumula os valores de mudanças de pesos para os padrões de treinamento apresentados. Os pesos da rede neural precisam ser consistentes em todos os processadores, e a atualização dos pesos da rede deve ser uma operação global. As mudanças de pesos calculadas em cada processador são somadas e usadas para atualizar as matrizes de pesos locais.

O paralelismo de exemplos fornece uma boa solução para implementação em *cluster* de computadores por necessitar de um nível de sincronização muito menor do que no paralelismo de neurônios ou no paralelismo de pesos. O baixo nível de sincronização vem do fato que as comunicações ocorrem no fim de cada época e geram um pequeno número de grandes mensagens (Pethick, Liddle, Werstein, & Huang, 2003).

Para ser vantajosa, esta estratégia requer um conjunto de treinamento adequadamente grande (em relação ao tamanho da rede), o que é uma situação comum já que muitos problemas possuem mais pares de entrada que neurônios.

Entre as possíveis estratégias de paralelização em redes neurais, o paralelismo de exemplos é muito superior a todas as outras estratégias aplicadas em diferentes configurações de hardware e para todos os tamanhos dos conjuntos de treinamento, menos os muito pequenos (Rogers & Skillicorn, 1998).

4. JAVA

4.1. Introdução

Java tem se tornado cada vez mais popular como uma linguagem de programação de uso geral. Implementações atuais em Java centram-se, sobretudo, na portabilidade e interoperabilidade, características estas que são necessárias na computação baseadas em modelos cliente-servidor na internet. A chave para o sucesso de Java vem do fato das aplicações serem interpretadas podendo, desta forma, ser transferidas e executadas por máquinas virtuais em praticamente qualquer plataforma. Para execução seqüencial, compiladores *Just-in-Time* aumentam o desempenho das aplicações. No entanto, a computação de alto desempenho tipicamente requer sistemas de múltiplos processadores, e uma comunicação eficiente entre os processadores se faz necessária para uma execução seqüencial eficiente.

Sendo uma linguagem orientada a objetos, Java utiliza a invocação de métodos como conceito principal de comunicação. Dentro de uma simples máquina virtual de Java, *threads* concorrentes de controle podem se comunicar via invocação de métodos sincronizados. Em um sistema de múltiplos processadores com memória compartilhada, esta abordagem permite alguma forma limitada de verdadeiro paralelismo pelo mapeamento das *threads* em diferentes processadores físicos. Para sistemas de memória distribuída, Java oferece o conceito da invocação remota de métodos (Java RMI). Aqui, a invocação de métodos, com seus parâmetros e retornos, é transferida por uma rede até um objeto em uma máquina virtual remota.

Com estes conceitos de simultaneidade e comunicação com memória distribuída, Java proporciona uma oportunidade única para uma linguagem de propósito geral amplamente aceita, com uma vasta base de código existente para o programador que pode também atender as necessidades da computação paralela.

4.2. Java RMI

O sistema de invocação remota de métodos de Java (Java RMI) permite que um objeto executando em uma máquina virtual Java invoque métodos em um objeto executando em outra máquina virtual Java (An Overview of RMI Applications, 2008).

Aplicações RMI geralmente são compostas de dois programas separados, um servidor e um cliente. Um típico programa servidor cria alguns objetos remotos, faz referências a esses objetos acessíveis, e aguarda que clientes invoquem métodos nestes objetos. Um programa cliente típico, obtém uma referência remota a um ou mais objetos em um servidor e invoca métodos destes objetos. RMI provê o mecanismo pelo qual servidor e cliente se comunicam e trocam informações. Uma aplicação desse tipo é geralmente chamada de aplicação de objetos distribuídos.

Aplicações de objetos distribuídos necessitam localizar os objetos remotos, comunicarem com os objetos remotos e carregar as definições das classes dos objetos que são passados. A localização dos objetos remotos em uma aplicação distribuída pode ser realizada de forma simples pelo registro RMI. O RMI é também responsável pelos detalhes da comunicação entre os objetos remotos, sendo que, para o programador, a comunicação remota parece similar às invocações de métodos regulares de Java. RMI fornece também os mecanismos para o carregamento das definições das classes assim como para a transmissão dos dados de um objeto.

A Figura 4.1 demonstra uma aplicação distribuída que usa o registro RMI para obter uma referência a um objeto remoto. O servidor chama o registro para associar um nome a um objeto remoto. O cliente procura pelo objeto remoto pelo nome no registro do servidor e então invoca um método neste objeto. A ilustração também mostra que o sistema RMI usa um servidor web disponível para, quando necessário, carregar definições de classes do servidor para o cliente e do cliente para o servidor.

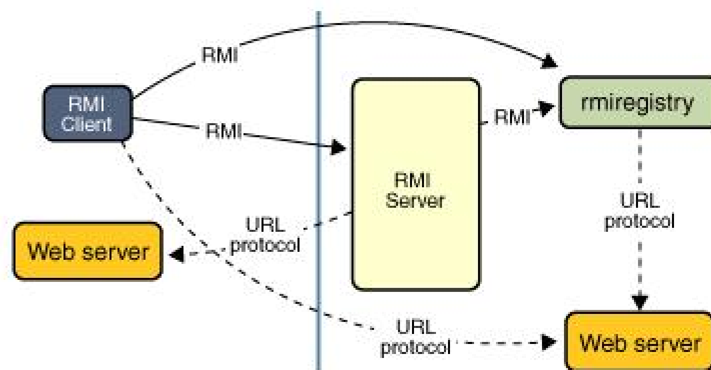

Figura 4.1 Aplicação distribuída RMI

Fonte: (An Overview of RMI Applications, 2008)

Portanto, Java RMI provê uma camada de rede intermediária que permite objetos Java residindo em máquinas virtuais diferentes se comunicarem usando chamadas normais de métodos. Isto significa que um cliente deverá ser capaz de acessar um servidor na máquina local ou em uma rede como se eles estivessem sendo executados no mesmo sistema (ver Figura 4.2), sendo que os detalhes de comunicação da aplicação distribuída não são requeridos. Para que um objeto remoto se torne acessível, é necessária uma interface que declara os métodos do objeto remoto. O servidor precisa implementar esta interface e quaisquer outras interfaces necessárias. Um *stub* e um *skeleton* precisam então ser gerados usando o compilador RMI. O *stub* é uma classe que automaticamente traduz chamadas de métodos remotos em propriedades de comunicação de redes e passagem de parâmetros. O *skeleton* é uma classe correspondente que reside na máquina virtual, e que aceita estas conexões de redes e as traduz em chamadas de métodos no objeto propriamente dito. Estes objetos remotos devem ser registrados com um serviço de nomeação que permite que os clientes os localizem. O cliente se conecta ao registro de nomeação e pede por uma referência a um serviço registrado através do nome registrado no servidor. O registro de nomeação então retorna uma referência remota a esse objeto.

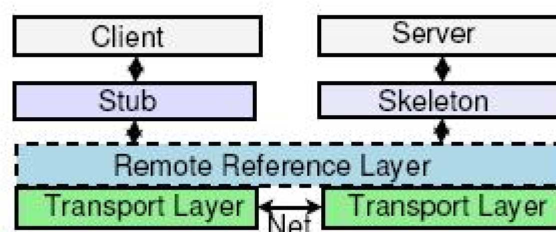

Figura 4.2 arquitetura em camadas do Java RMI.

Fonte: (Taboada, Teijeiro, & Touriño, 2007)

4.3. Java na Computação Paralela

Java, devido às suas atraentes características como independência de plataforma, portabilidade e crescente integração em aplicações existentes, vem ganhado cada vez mais espaços em ambientes onde linguagens mais tradicionais ainda são predominantes. Um desses ambientes é a computação paralela, onde o desempenho é um aspecto chave.

Em relação às aplicações paralelas de alto desempenho, a arquitetura mais comum é o *cluster*, uma vez que proporciona um desempenho paralelo excepcional a um preço razoável. No entanto, o uso de aplicações Java paralelas em *cluster*s permanece uma opção emergente, já que o uso de uma *middleware* de comunicação ineficiente atrasou sua utilização.

Em *cluster*s, o desempenho eficiente da comunicação é a chave para fornecer escalabilidade para aplicações paralelas, mas Java carece de uma *middleware* de comunicação eficaz. Mesmo que os nós do *cluster* sejam interconectados por uma rede de alta velocidade, Java não consegue tirar total proveito disso devido ao fato das tecnologias de interconexão não serem ainda bem suportadas. De fato, Java suporta completamente apenas interconexões através do protocolo TCP/IP o qual faz uso ineficiente de interconexões de alta velocidade (Taboada, Teijeiro, & Touriño, 2007).

Porém, segundo Taboada *et al*. (2007), diferentes frameworks têm sido implementados com o objetivo de melhorar a eficiência das comunicações RMI em *cluster*s. Os mais relevantes são KaRMI, RMIX, Manta e Ibis. KaRMI é uma alternativa de

substituição para a arquitetura Java RMI que usa um protocolo completamente diferente e introduz novas abstrações para melhorar as comunicações especialmente para ambientes de *cluster*s. No entanto, KaRMI sofre de perda de desempenho quando trabalha com grandes conjuntos de dados e sua interoperabilidade é limitada aos nós do *cluster*. RMIX estende a funcionalidade de Java RMI na tentativa de cobrir uma ampla variedade de protocolos de comunicação, mas a eficiência em *cluster*s de alto desempenho não é satisfatória. O projeto Manta é uma tentativa diferente para implementação RMI, baseada na compilação nativa de Java. O Manta permite uma melhor otimização, evita a serialização de dados e o processamento de informações de classes durante a execução e usa um protocolo de comunicação mais leve. Finalmente, o Ibis é uma solução Java que estende o Java RMI na tentativa de fazê-lo mais adaptável para *grid computing* (Taboada, Teijeiro, & Touriño, 2007).

Desta forma, apesar da implementação proposta neste trabalho se basear na forma padrão de Java RMI, é, aparentemente, possível realizar adaptações ao sistema com a utilização de alguma variação do RMI padrão no intuito de melhorar seu desempenho.

5. IMPLEMENTAÇÃO

5.1. Introdução

A aplicação proposta por este trabalho consiste na implementação de um sistema distribuído para treinamento de uma rede neural utilizando do paradigma de paralelismo de exemplos, ou paralelismo de conjunto de treinamento.

No paralelismo de exemplos, discutido no capítulo anterior, ocorre a divisão do conjunto de treinamento em subconjuntos, onde cada processo possui uma cópia da rede e calcula a atualização dos pesos da rede relativa ao seu subconjunto de treinamento. As matrizes de atualizações dos pesos obtidas por cada processo são, então, reunidas no intuito de se obter a matriz de pesos que será de fato utilizada para atualizar os pesos da rede no final de uma época de treinamento. Este processo de treinamento parcial pelos processadores, junção dos resultados e atualização das redes locais de cada processador é repetido até que a convergência do treinamento atenda um critério de parada, geralmente baseado no valor de erro alcançado pelo treinamento ou em um número de épocas.

Como a proposta do trabalho se trata de uma implementação em *software* que será executada em um *cluster* de computadores, a estratégia do paralelismo de exemplos é a mais indicada como foi discutido no capítulo 3.

A linguagem de programação escolhida para a implementação foi Java. Os motivos para a escolha desta linguagem para a implementação se baseiam nas próprias qualidades já conhecidas de Java, como portabilidade, robustez, segurança, orientação a objetos, alto desempenho e facilidade de implementação. Além disso, a possibilidade de se usar o Java *Remote Method Invocation* (Java RMI) torna a implementação distribuída mais simples, já que a invocação remota de métodos é praticamente tão simples quanto chamada de métodos locais. É possível, por exemplo, enviar mensagens contendo objetos entre computadores remotos de forma simples. No entanto, eventuais desvantagens na utilização de Java RMI em aplicações paralelas onde se busca um bom desempenho de processamento devem ser observadas. Problemas como, por exemplo, o tempo gasto na serialização de objetos (transformação de objetos em fluxos de *bytes*), assim como o tempo gasto em comunicação usando o Java RMI, pode reduzir o desempenho da aplicação em

relação a outras técnicas de transmissão de mensagens ou outras linguagens cujas aplicações sejam teoricamente mais rápidas que Java.

Nas próximas seções serão apresentadas a análise do problema e a modelagem para implementação da parte básica do sistema, envolvendo as classes do sistema referentes ao treinamento da rede de forma específica para o problema, e a modelagem do sistema distribuído como um todo.

5.2. Descrição do Problema

O objetivo do sistema é realizar o treinamento de uma rede neural de forma distribuída através do paradigma do paralelismo de exemplos. Portanto, é necessária a implementação de componentes que computem as atualizações de pesos para cada par de treinamentos de um subconjunto do conjunto de treinamento e acumulem os resultados gerando as matrizes de atualizações dos pesos resultantes da computação local. Os resultados obtidos por cada um destes componentes, que irão operar de forma paralela e independente, devem ser reunidos para se obter as reais matrizes de atualizações de pesos.

Deste modo, é preciso que haja outro componente do sistema que ficará encarregado de gerenciar o treinamento juntando esses resultados e disponibilizando o resultado final para os componentes encarregados do treinamento.

Tem-se então um modelo cliente-servidor, onde cada cliente realiza o treinamento parcial, e o servidor que administra todo o processo. Cada cliente deve estabelecer uma conexão com o servidor. No ponto de vista do paralelismo do treinamento, é importante que os clientes sejam independentes uns dos outros, e principalmente que o servidor seja capaz de lidar com todos os clientes ao mesmo tempo. Ou seja, o servidor deve ser capaz de lidar com todos os clientes de forma simultânea tanto na leitura dos resultados dos clientes quanto no envio das matrizes de pesos da rede para os clientes.

O usuário do sistema deve definir os parâmetros do treinamento, que correspondem à:

1. A topologia da rede, número de entradas, número de camadas e a quantidade de neurônios em cada camada, além das funções de ativação de cada camada;

2. O conjunto de treinamento compatível com a rede, ou seja, os tamanhos dos vetores de entrada e de saída do conjunto de treinamento devem coincidir com o número de entradas e de saídas da rede neural;
3. Os valores para a taxa de aprendizado e taxa de momentum que serão usados no treinamento;
4. O critério de parada do treinamento, que corresponde ao número máximo de épocas de treinamento e/ou o valor de erro médio quadrático (*mean-squared-error*).

Fica a cargo do servidor:

1. A divisão do conjunto de treinamento entre os clientes envolvidos no treinamento;
2. A atualização dos pesos da rede para cada época do treinamento;
3. Estabelecer o fim do treinamento quando for atingido o critério de parada.

A única função dos clientes é a de computar as mudanças de pesos resultantes do treinamento local, sem ainda efetuar qualquer mudança em suas redes locais. Os clientes só atualizaram suas redes quando o servidor disponibilizar os pesos da rede para a próxima época do treinamento.

Nas próximas seções serão apresentados os diagramas UML referentes à modelagem proposta do sistema.

5.3. Estratégia de Implementação

Portanto, considerando como atores do sistema, o usuário do sistema, o servidor e os clientes, pode-se estabelecer o diagrama de casos de uso da Figura 5.1. Este diagrama é útil para se observar apenas o que deve ser feito pelas partes do sistema, sem, no entanto detalhar a forma que irão funcionar.

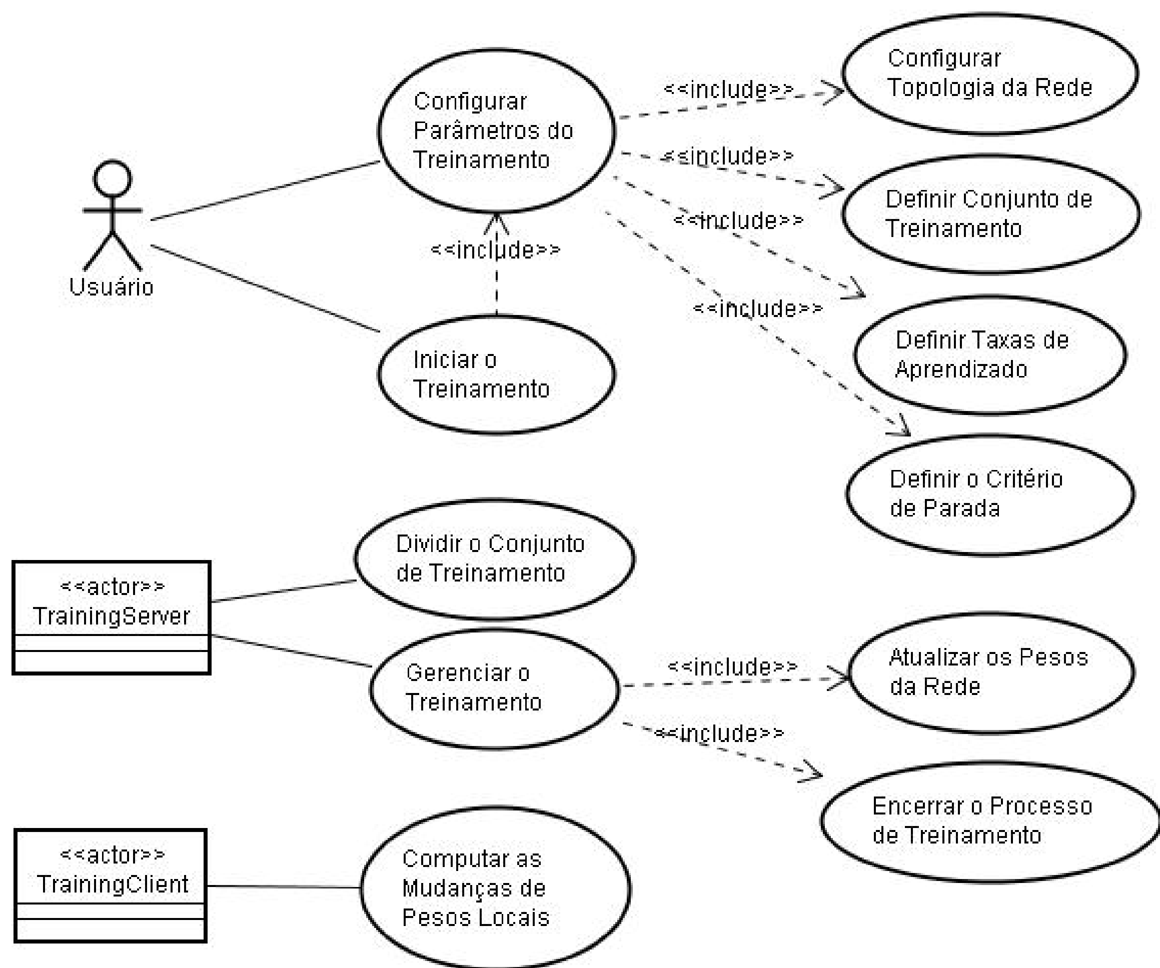

Figura 5.1: Erro! Use a guia Início para aplicar 0 ao texto que deverá aparecer aqui.**Diagrama de Casos de Uso**

Toda a informação presente em uma rede neural do tipo *feed-forward* treinada pelo algoritmo *backpropagation* consiste da topologia da rede, das funções de ativação dos neurônios e principalmente dos valores dos pesos entre as conexões. Os pesos sinápticos são representados por matrizes de pesos, que trazem consigo a forma de parte da topologia da rede. Por exemplo: uma rede neural com duas entradas, uma camada oculta com três neurônios. E, um neurônio na camada de saída pode ser representado por duas matrizes, sendo: uma matriz com duas linhas e três colunas representando os pesos entre as entradas e a camada oculta e uma matriz com três linhas e uma coluna representando os pesos entre a camada oculta e a camada de saída.

Assume-se que todos os neurônios de uma mesma camada tenham a mesma função de ativação. Pode-se, portanto, representar uma rede neural como uma coleção de matrizes, sendo uma matriz para cada camada da rede, e a especificação do tipo de função de ativação de cada camada.

O algoritmo *backpropagation batch* pode ser descrito da seguinte forma:

Para cada par de treinamentos:

Passo um: Calcular as saídas da rede;

Passo dois: Calcular os termos de erros para cada camada;

Passo três: Calcular as variações de pesos;

Passo quatro: Armazenar as variações de pesos.

A atualização dos pesos da rede acontece após cada época, ou seja, após calcular e armazenar as variações de pesos referentes a cada par de treinamentos. Todo processo é repetido até que se atinja o critério de parada.

No entanto, na aplicação proposta neste trabalho, as partes processadoras do treinamento (clientes) não efetuarão nenhuma atualização dos pesos durante o treinamento local. Portanto, o algoritmo implementado nos clientes do treinamento será um método que usa como parâmetro o conjunto de treinamento local, e retorna um objeto contendo as alterações dos pesos resultantes do processamento local.

Os passos do *backpropagation batch* citados anteriormente podem ainda ser mais bem detalhados para que se possa adotar uma estratégia de implementação:

O cálculo das saídas da rede corresponde à ativação das camadas da rede em seqüência. Primeiramente é calculado o vetor de valores de entrada na primeira camada de neurônios. Este vetor é o produto do vetor de entradas do par de treinamentos pela matriz de pesos da primeira camada. Este vetor resultante deve ser armazenado para o cálculo futuro das variações de pesos. Este vetor é utilizado para ativar os neurônios da primeira camada gerando outro vetor contendo as saídas dos neurônios desta camada. Este vetor de saídas por sua vez é multiplicado pela matriz de pesos da próxima camada gerando o vetor

de valores de entrada da segunda camada, o qual será usado na ativação desta camada. Este processo é repetido ao longo das camadas até gerar o vetor de saídas da rede.

O cálculo dos termos de erros começa com a comparação do vetor de saídas da rede com as saídas desejadas referentes ao padrão de treinamento em questão. O processo de cálculo dos termos de erros é semelhante ao processo do cálculo das saídas, porém no sentido inverso. O cálculo das variações de pesos também ocorre camada por camada, de forma seqüencial. É importante notar que ocorre uma intensa multiplicação de matrizes por vetores. Durante o cálculo das saídas da rede, são calculados os vetores de entrada e os vetores de saída nos neurônios de cada camada, que serão usados posteriormente respectivamente no cálculo das mudanças de peso e no cálculo dos termos de erro dos neurônios.

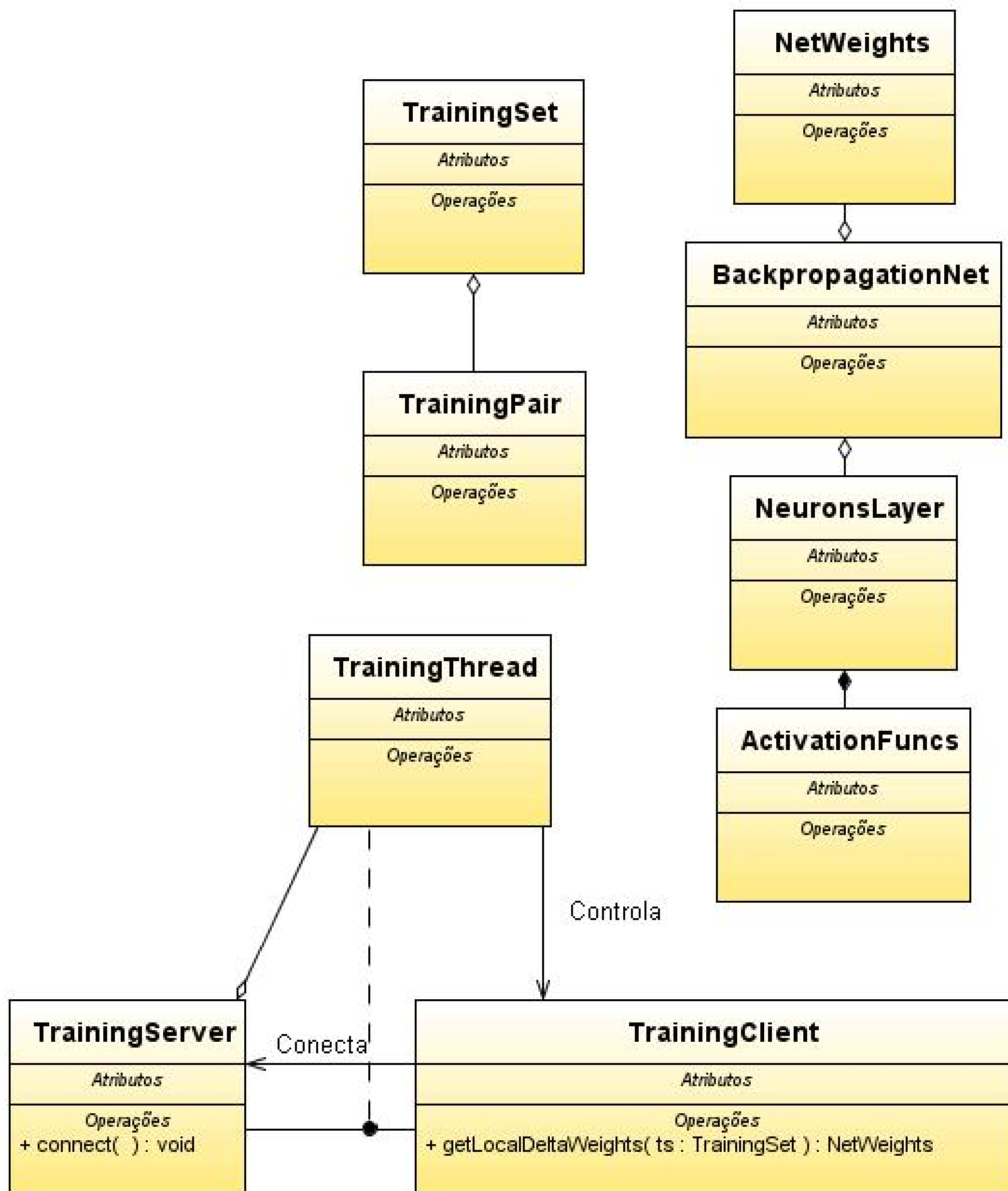

Figura 5.2: Diagrama de Classes

Portanto, a estratégia adotada na implementação foi a de usar uma classe única que abrangesse toda informação referente a uma camada da rede (***NeuronsLayer)*** na tentativa de simplificar a implementação do algoritmo de treinamento, ao invés de implementar os neurônios separadamente, o que aumentaria a granularidade da implementação. A figura 5.2 demonstra as relações entre as classes. No Anexo, pode-se encontrar o código fonte das principais classes da aplicação.

Um objeto da classe ***BackpropagationNet*** possui como principais atributos um objeto da classe ***NetWeigths*** e uma coleção de objetos do tipo ***NeuronsLayer***. A classe ***NeuronsLayer*** possui o método ***activateNeurons(double input*[])** a qual usa a função de ativação da camada para obter o vetor contendo as saídas dos neurônios da camada e as funções ***getInputs*()** e ***getOutputs*()** que retornam os vetores de entrada e de saída da camada. As implementações dos métodos das classes são encontradas no Anexo. No diagrama da Figura 5.2 são ocultados os atributos e operações das classes para possibilitar a melhor visualização das relações entre as classes.

Como diferentes funções de ativação podem ser usadas para um determinado problema (sigmóide bipolar ou binária) foi implemetada também a classe ***ActivationFuncs*** a qual possui dois métodos, ***actFunc(double* x)** e ***derivativeActFunc(double* x)**, que retornam o resultado da função de ativação e o resultado da derivada da função de ativação respectivamente.

A classe ***NetWeights*** corresponde a todos os pesos da rede e possui o método ***getWeights*(int i)** que retorna a matriz de pesos referente à camada **i**. As mensagens entre o servidor e os clientes do treinamento correspondem a objetos desta classe. Portanto, não será enviada toda a rede em uma mensagem, durante o treinamento. O servidor enviará um objeto da classe ***BackpropagationNet*** ao cliente apenas no estabelecimento da conexão entre cada cliente.

A classe ***TrainingServer*** e ***TrainingClient*** corresponde às classes que definem o servidor e o cliente de treinamento. Quando um objeto da classe ***TrainingClient*** se conecta a um ***TrainingServer*** o servidor cria um objeto do tipo ***TrainingThread*** representado pela classe associativa do diagrama de classes. Cada objeto do tipo ***TrainingThread*** possui uma referência do cliente que gerou o pedido de conexão, e é responsável pelo controle deste cliente. O objeto ***TrainingThread***, portanto, é quem faz a invocação remota do cliente através do método ***getLocalDeltaWeights(NetWeights*)**. Tal método tem como parâmetro o objeto ***NetWeights*** que corresponde aos pesos da rede disponibilizados pelo servidor (pesos atualizados), e retorna outro objeto do tipo ***NetWeights*** correspondente ao resultado do treinamento local do cliente. A classe ***TrainingThread*** implementa a classe ***Thread*** de Java, o que significa que se trata de objetos que terão sua execução de forma paralela e independente.

Portanto, existira uma *thread* no servidor para cada cliente conectado como mostrado na figura 5.3.

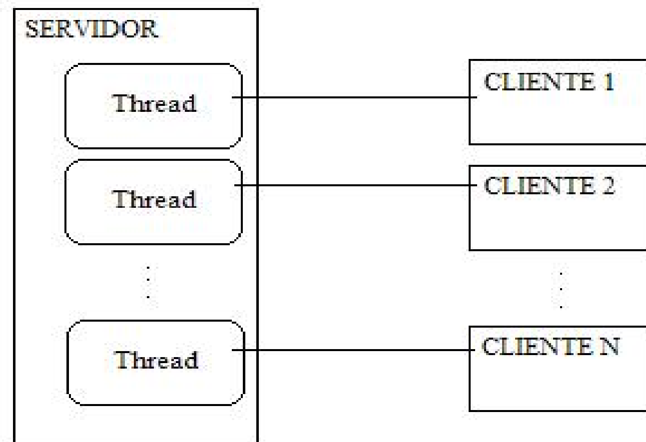

Figura 5.3: Conexão entre os clientes e as *threads* do servidor

A sincronização destes processos paralelos é de extrema importância para o funcionamento correto do sistema. Estes processos irão acessar o servidor para obter os pesos atualizados da rede e então enviá-los aos clientes. Então, após obterem as respostas de seus respectivos clientes e entregá-las ao servidor, os processos precisam aguardar até que o servidor compute as novas matrizes de pesos ou encerre o treinamento no caso de ter se alcançado o critério de parada. A classe ***TrainingServer***, portanto, possui dois métodos que serão utilizados pelas *threads* do treinamento e que estabelecem a sincronização da leitura e escrita dos pesos na memória do servidor. São os métodos: ***getServerWeights***, que retorna um objeto ***NetWeights*** correspondente aos pesos atualizados da rede, e ***setServerWeights*** que apresenta ao servidor os pesos recebidos dos clientes.

O servidor, portanto, só pode disponibilizar os pesos da rede para a próxima época do treinamento após receber todas as respostas dos clientes e fazer as devidas computações necessárias. Deste modo, tem-se uma espécie de memória compartilhada no servidor, na qual os processos paralelos terão acesso restrito de leitura e escrita. Ou seja, cada processo pode "escrever" na memória compartilhada somente depois de todos os outros processos terem realizado a leitura desta memória. Da mesma forma, cada processo só poderá "ler" esta memória compartilhada após todos os outros processos terem apresentado suas mudanças de pesos e o servidor computado as novas mudanças de pesos.

Vale ressaltar que a proposta de implementação aqui apresentada foi desenvolvida especificamente para o problema em questão. O objetivo deste trabalho não é desenvolver um modelo para reutilização ou adaptação em paradigmas diferentes de paralelismo ou mesmo em modelos diferentes de treinamento ou de redes neurais.

5.4. Interfaces da Aplicação

A Figura 5.4 mostra a interface principal usada na aplicação para a realização dos testes.

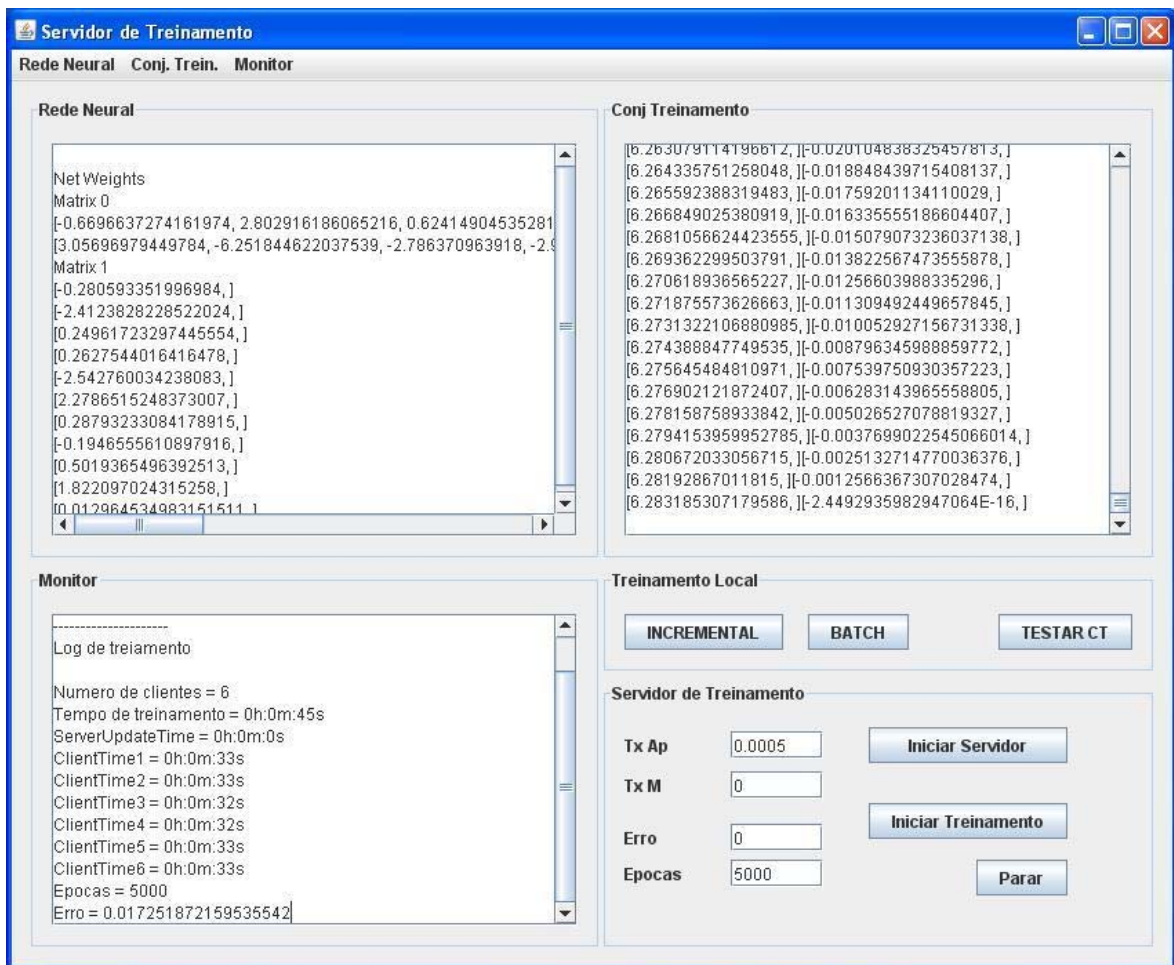

Figura 5.4: Interface principal

O usuário da aplicação, para realizar algum treinamento, deve definir a rede neural, o conjunto de treinamento e os parâmetros de treinamento. O menu Rede Neural possui as opções para o usuário criar uma nova rede, carregar uma rede previamente salva, ou salvar uma rede. Além destas há a opção de inicializar os pesos da rede, gerando valores aleatórios entre -0,5 e 0,5 para os pesos. Ao optar por criar uma nova rede, a tela de diálogo mostrada na figura 5.5 é exibida para que o usuário defina o número de entradas da

rede, a quantidade de neurônios em cada camada e a função de ativação dos neurônios da rede. O usuário define o conjunto de treinamento através da opção “carregar” no menu Conj. Trein. O painel Monitor é usado para visualizar os relatórios de treinamento paralelo contendo os tempos gastos por cada cliente, o número de épocas e o erro alcançado como também outros relatórios gerados pelo sistema acessáveis através do menu Monitor.

O painel Servidor de Treinamento é usado para a definição dos parâmetros de treinamento: taxa de aprendizado, taxa de momentum, épocas e erro, onde os valores de épocas e erro definem o critério de parada do treinamento. Após a definição dos parâmetros de treinamento, da rede e do conjunto de treinamento, o usuário pode “iniciar o servidor”, o qual ficará aguardando por conexões dos clientes, e então, “iniciar o treinamento” após serem estabelecidas as conexões.

A interface também possui a opção de realizar o treinamento de forma local, usando o algoritmo *backpropagation* padrão, com atualização incremental de pesos, ou usando o *backpropagation batch*. É ainda possível realizar o teste da rede sob um conjunto de treinamento, o que exibe no painel Monitor as saídas obtidas pela rede, as saídas desejadas do conjunto de treinamento e o erro obtido para cada exemplo do conjunto.

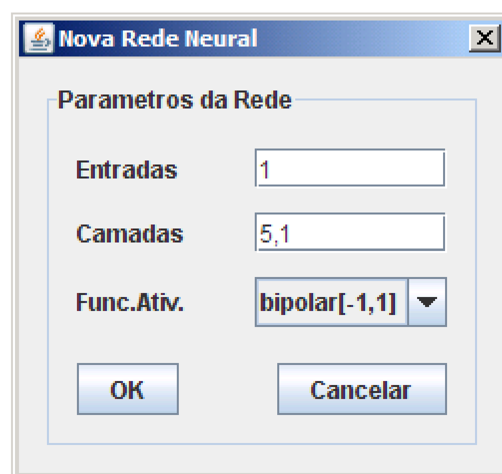

Figura 5.5: Interface da aplicação para criação da rede neural

A aplicação também permite a exibição do erro e da época durante o treinamento como mostrado na figura 5.6. Porém este tipo de exibição deixa a aplicação mais lenta

servindo apenas de ilustração para se poder observar o processo de convergência do treinamento e deve ser desabilitado para se obter um treinamento mais rápido.

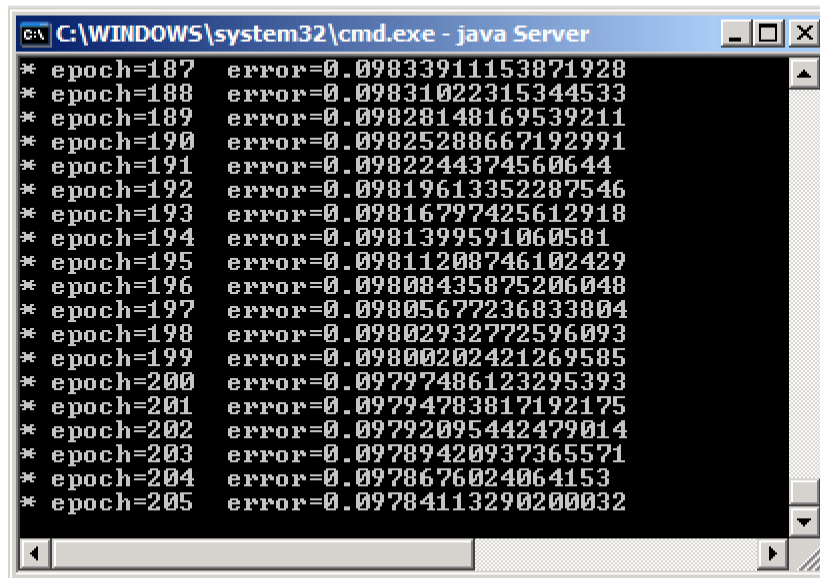

Figura 5.6: Tela de exibição dos valores de erro através das épocas de treinamento.

Figura 5.7: Interface do cliente.

A Figura 5.7 mostra a interface do cliente da aplicação. Através desta interface o usuário estabelece a conexão entre um cliente e o servidor definindo um nome para o cliente, o qual será usado pelo servidor para encontrar o cliente no registro do computador onde o cliente é executado, e o número de IP do servidor.

6. RESULTADOS

Nos testes realizados com o sistema implementado tentou-se observar em que situações a convergência era mais rápida utilizando o treinamento paralelo em relação ao treinamento seqüencial. Nos testes, foi realizado o treinamento da função seno(x) a partir de um conjunto de treinamento contendo cinco mil valores entre zero e dois PI, como valores de entrada, e os respectivos valores de seno como saída desejada. O treinamento foi realizado de forma paralela, variando o número de clientes entre um e seis, e de forma local (não paralela) usando o algoritmo backpropagation *batch*. Foi utilizada uma rede com cinco neurônios na camada interna com valores de pesos pré-estabelecidos. Todos os testes foram realizados com uma taxa de aprendizado de 0,0005.

Para todos os testes, a rede foi treinada durante cinco mil épocas. O objetivo destes não foi o de convergir às redes a um determinado erro, mas sim o de observar a relação entre o número de clientes, o tamanho da rede e o tempo gasto no treinamento com um número pré-estabelecido de épocas. Os testes foram realizados em máquinas *Pentium* 4 com sistema operacional *Windows XP* interligados por uma rede *ethernet* de 100MB/s.

A Tabela 6.1 mostra os resultados obtidos no treinamento com a rede neural contendo cinco neurônios na camada interna. A Figura 6.1 mostra os resultados em forma de gráfico. No treinamento da rede de forma local (não paralela) foi gasto um tempo de 99 segundos.

clientes	tempo total(seg)	tempo cliente(seg)	tempo restante(seg)	sub conjunto
1	102	97	5	5000
2	55	49	6	2500
3	42	34	8	1666
4	36	27	9	1250
5	32	22	10	1000
6	30	19	11	833

Tabela 6.1: Tempos obtidos no treinamento paralelo da rede 1-5-1.

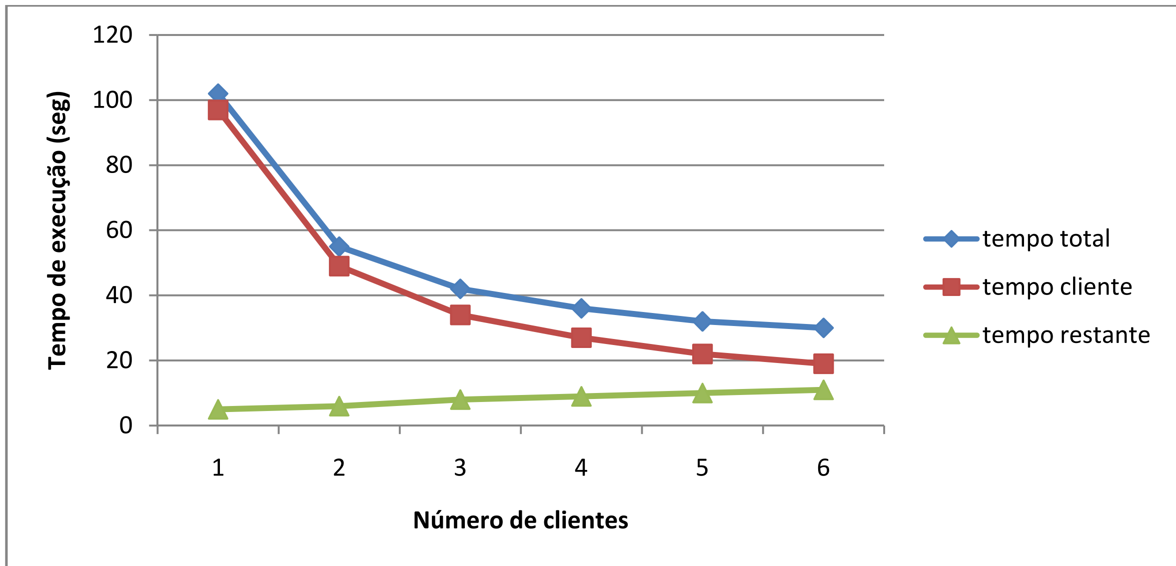

Figura 6.1: Gráfico do tempo de treinameto em relação ao número de clientes (rede 1-5-1).

O tempo total mostrado na Tabela 6.1 e na Figura 6.1 representa o tempo total gasto durante as cinco mil épocas. O tempo do cliente representa o tempo de processamento gasto pelos clientes no cálculo das variações de pesos para seus sub-conjuntos de treinamento. O tempo restante equivale a diferença entre o tempo total e o tempo de processamento dos clientes. Pode-se observar pelo gráfico apresentado que com a utilização de dois clientes, o tempo gasto no treinamento é de pouco mais que a metade do tempo gasto com um cliente apenas.

Na medida que aumenta-se o número de clientes envolvidos no processo, continua a redução do tempo total de treinamento, porém observa-se uma diminuição no ganho em desempenho. Isso se deve ao fato de que com um número maior de clientes, o tempo gasto pelo servidor na atualização dos pesos da rede e no manuseio das mensagens enviadas aos clientes (tempo restante) aumenta. O tempo restante mostrado no gráfico portanto, equivale ao tempo gasto nas transimssões das mensagnes entre o servidor e os clientes, mais o tempo de processamento no servidor nas atualizações dos pesos. De fato, em todos os testes realizados, o tempo de atualização dos pesos por parte do servidor foi mínimo em relação ao tempo restante. Porém, na medida que se aumenta o número de clientes, as *threads* do servidor que administram os clientes demoram mais tempo para serem processadas.

A Figura 6.2 mostra a generalização decorrente do treinamento do da rede 1-5-1.

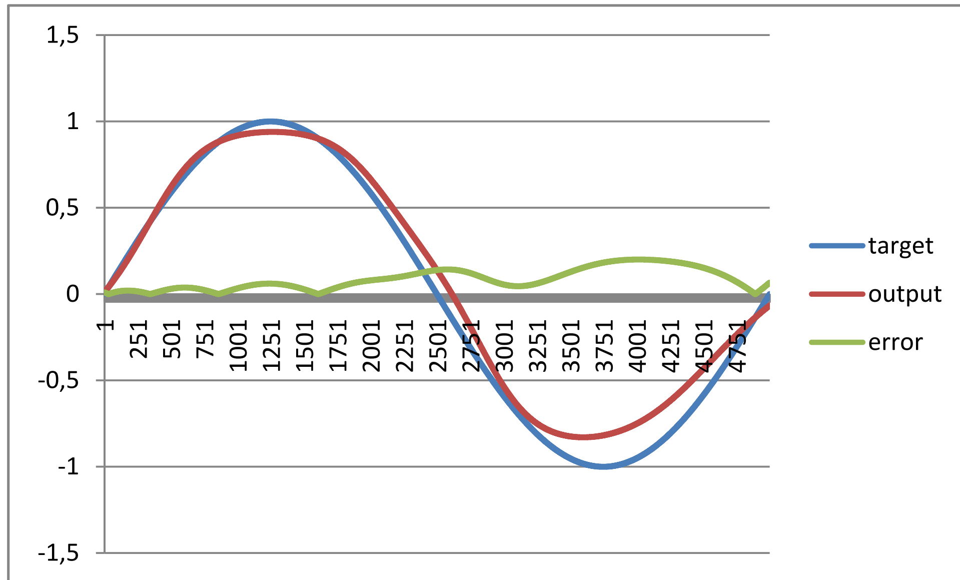

Figura 6.2: Generalização do treinamento no teste da rede 1-5-1.

A Figura 6.3 mostra o processo de convergência do erro para o treinamento da rede 1-5-1.

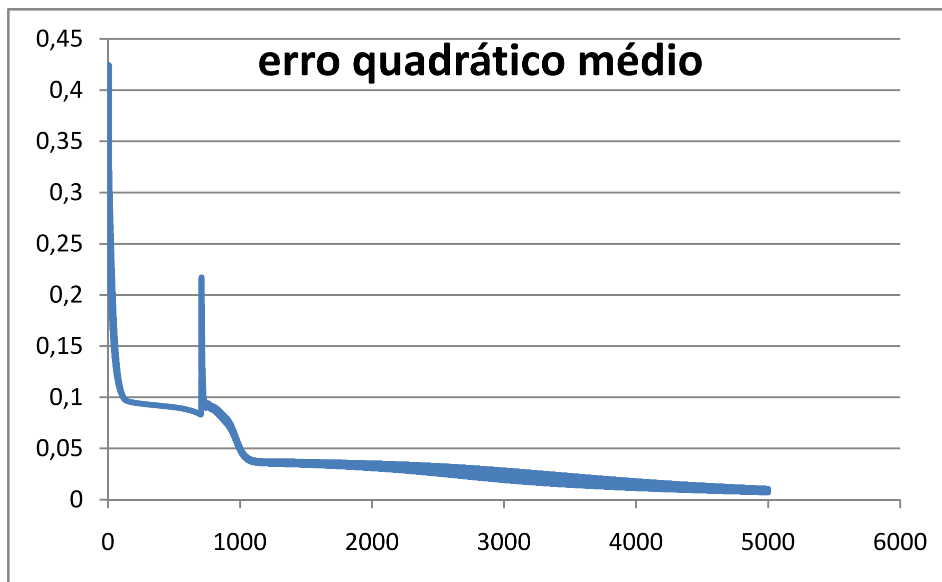

Figura 6.3: Convergência do erro quadrático médio em função das épocas de treinamento

Nos testes realizados foi pré-estabelecido que o treinamento durasse cinco mil épocas para que fosse possível analisar a relação entre o número de clientes e o tempo

gasto. No entanto, se fosse utilizado como critério de parada um número maior de épocas ou um valor de erro menor do que o obtido, teoricamente, a generalização da rede seria mais exata, e as linhas do gráfico da figura 6.2 que representam a saída desejada e a saída obtida seriam mais semelhantes. Da mesma forma, o valor de erro, representado na figura 6.3 tenderia a ser reduzido.

O ganho em desempenho obtido nos resultados e exibidos na figura 6.1 pode ser comparado aos resultados alcançados em trabalhos correlatos encontrados na literatura nos quais foi inspirado este trabalho, como por exemplo, em: Melcíades *et al*. (1999) e Pethick *et al*. (2003).

Em Melcíades *et al*. (1999) foi realizado o estudo da paralelização do algoritmo *backpropagation* em *clusters* de estações de trabalho. Neste trabalho foram comparados alternativas paralelas dos métodos de atualização de pesos: online, *batch* e *block*, onde se chegou à conclusão, a partir de resultados experimentais, que a paralelização do algorítmo em modo *batch* obteve os melhores resultados.

Já o trabalho realizado em Pethick *et al*. (2003) teve como objetivo a comparação de duas estratégias de paralelismo para redes neurais usando o *backpropagation* em *clusters* de computadores: paralelismo de exemplos e paralelismo de nós, e cujos resultados experimentais mostram as vantagens e desvantagens de cada estratégia em função do tamanho da rede, do tamanho do conjunto de treinamento e do número de processadores.

Em Melcíades *et al*. (1999), o *cluster* foi formado por estações de trabalho heterogêneas SUN e IBM, rodando UNIX e PVM (*Parallel Virtual Machine)* interligadas através de uma rede ethernet de 10Mb/s. Foi utilizado nos testes uma rede neural com duas camadas internas contendo 12 e 4 neurônios e 2 neurônios na camada de saída para o treinamento de um problema simplificado de classificação usando como critério de parada um valor de erro pré-determinado (maiores detalhes podem ser encontrados no trabalho original). A figura 6.4 mostra os resultados obtidos nesse trabalho.

Modos	Número de épocas
Online seqüencial	3139
Batch seqüencial	4501
Batch paralelo	4501
Block paralelo (1 bloco =1/2 do conjunto de treinamento)	2956
Block paralelo (1 bloco =1/3 do conjunto de treinamento)	2537
Block paralelo (1 bloco =1/4 do conjunto de treinamento)	2332

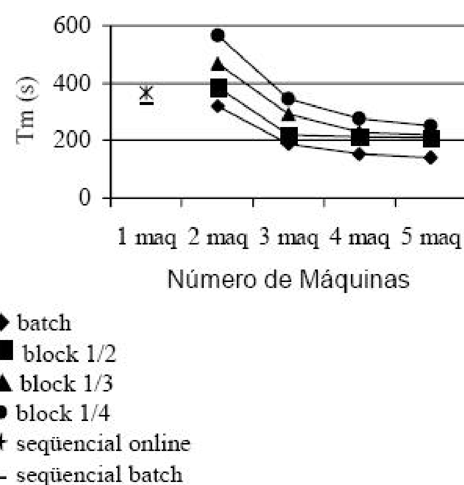

Figura 6.4: Resultados do treinamento sequencial e paralelo

Fonte: (Melcíades, Fiallos, & Pimentel, 1999)

Segundo Melcíades *et al*. (1999), os resultados mostram que o modo *batch* requer maior número de épocas de treinamento. Etretanto, os tempos de treinamento no modo *batch* são menores do que os correspondentes no modo *online*.

Os testes realizados em Pethick *et al*. (2003) foram feitos em um *cluster* de 32 máquinas Intel Pentium II 350MHz e 192MB de memória e sistema operacional Red Hat GNU-Linux interligados através de uma rede ethernet de 100MB/s. A implementação foi feita na linguagem C usando MPI (*Message Passing Interface*) como ferramenta para troca de mensagens. Os testes foram realizados com uma rede contendo três camadas contendo respectivamente N, ¾ N e ½ N neurônios em cada camada. Os tamanhos de rede testados (valor de N) foram 250, 500, 1000 e 2000. Os conjuntos de treinamento foram formados por valores de entrada e saída desejada dentro do intervalo [0,1 0,9]. Foram testados conjuntos de treinamento contendo 100, 1000, 10000 e 20000 pares de treinamento. Cada teste foi realizado usando 1, 2, 4, 8, 16 e 32 processadores. Todos os testes foram realizados durante 50 épocas. As figuras 6.5 e 6.6 mostram os gráficos obtidos através dos testes usando o paralelismo de exemplos variando o tamanho do conjunto de treinamento e o tamanho da rede respectivamente.

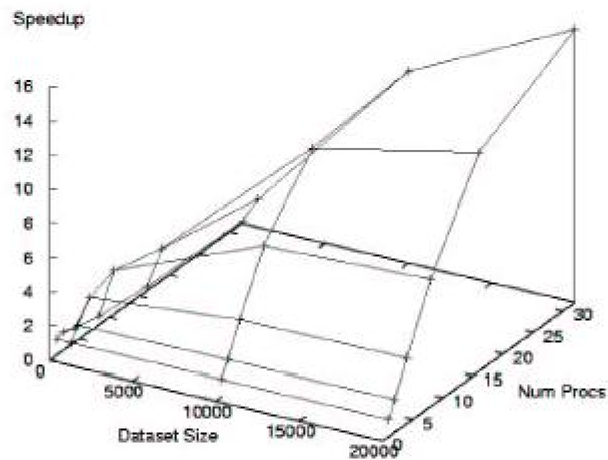

Figura 6.5: Desempenho do paralelismo de exemplos com o aumento do conjunto de treinamento

Fonte: (Pethick, Liddle, Werstein, & Huang, 2003)

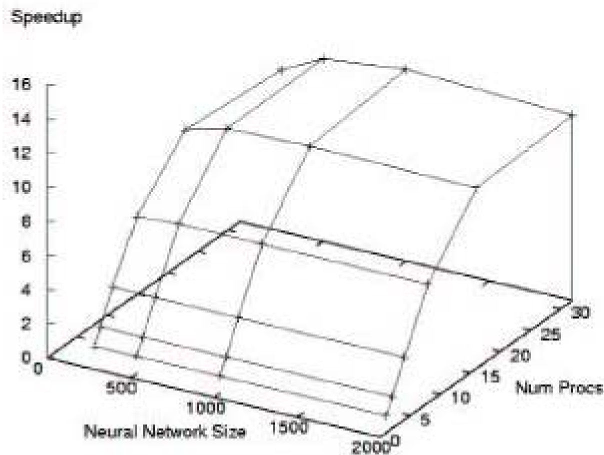

Figura 6.6: Desempenho do paralelismo de exemplos com o aumento do tamanho da rede

Fonte: (Pethick, Liddle, Werstein, & Huang, 2003)

Segundo Pethick *et al*. (2003), a implementação do paralelismo de exemplos mostrou uma forte correlação entre o desempenho e o tamanho do conjunto de treinamento. Como mostrado na figura 6.5, para um dado tamanho da rede, a aceleração aumenta rapidamente com o aumento do tamanho do conjunto de treinamento. Entretando, a figura 6.6 mostra que há uma correlação relativamente fraca entre o desempenho e o tamanho da rede para um dado tamanho do conjunto de treinamento. Isso se deve ao fato de que o tamanho do conjunto de treinamento determinam a quantidade de computação feita por cada processador para cada época, enquanto o tamanho da rede determina a quantidade de dados que é transmitida. Portanto, quanto maior o conjunto de treinamento, maior será a razão entre a computação e a comunicação, e assim, maior será o ganho em desempenho.

7. CONCLUSÃO

7.1. Considerações Finais

O tempo total gasto no processo de treinamento paralelo de uma rede pela implementação proposta é: basicamente igual ao tempo de processamento no servidor (atualização dos pesos) mais o tempo de processamento médio dos clientes (cálculo das variações de pesos locais) mais o tempo gasto na troca de mensagens (envio das matrizes de pesos).

O processamento no servidor se consiste basicamente de somar as matrizes de pesos recebidas dos clientes e, principalmente, distribuir as novas matrizes a cada cliente, portanto, proporcional ao número de clientes. Quanto maior o número de clientes conectados ao servidor, maior o tempo gasto pelo servidor no tratamento das *threads* que controlam os clientes.

O tempo de processamento do lado dos clientes é proporcional ao tamanho do conjunto de treinamento local. Para um conjunto de treinamento maior, pode-se aumentar o número de clientes para se obter um menor tempo total de treinamento. No entanto é difícil estipular com exatidão o número máximo de clientes possíveis, pois esse valor depende do tamanho e da complexidade do conjunto de treinamento.

Os resultados obtidos pelos testes realizados mostram que a paralelização do treinamento reduz o tempo gasto pelo processo. Assim como encontrado na literatura, a utilização da estratégia do paralelismo de exemplos é adequada à implementação em *software* para a execução em um *cluster* de computadores.

7.2. Trabalhos Futuros

O desenvolvimento de um sistema mais eficiente poderia ser realizado, utilizando alguma variação do RMI na tentativa de se reduzir o custo em comunicações presentes na aplicação paralela. Além disso, heurísticas que visão aperfeiçoar o algoritmo *backpropagation*, como por exemplo, taxas de aprendizado adaptativas e outras heurísticas

de otimização para aprendizado em redes neurais também poderiam ser empregadas para diminuir o tempo de treinamento.

Nos testes realizados foi feita a análise do tempo de treinamento em relação ao aumento do paralelismo. Uma proposta de continuidade do trabalho é a de analisar o erro em relação ao aumento do paralelismo. Ou seja, verificar se com o aumento do número de processos envolvidos no treinamento ocorre alguma mudança na convergência do erro.

REFERÊNCIAS BIBLIOGRÁFICAS:

An Overview of RMI Applications. (14 de fevereiro de 2008). Acesso em 16 de novembro de 2008, disponível em The Java Tutorials: <http://java.sun.com/docs/books/tutorial/rmi/overview.html>

Dreyfus, G. **Neural Networks:** Methodology and Applications. Secaucus, NJ, USA: Springer-Verlag New York, Inc. 2005. 497p.

Forti, M. A neural network for signal decomposition problems. **International Journal of Circuit Theory and Applications.** v. 19. p. 65-75. Electronic Engineering Department, University of Florence. 1991.

Gironés, R. G., & Salcedo, A. M. **Forward-Backward Parallelism in On-Line Backpropagation**. In: J. Mira, & J. V. Sanchés-Andrés, Engineering Applications of Bio-Inspired Artificial Neural. 1607. 1999 Alicante, Spain: Springer. p. 157-165.

Hassoun, M. H. **Fundamentals of Artificial Neural Networks**, 1st edition. MIT Press. 1995. 551 p.

Haykin, S.. **Neural Networks: A Comprehensive Foundation** ***2. ed***. Mc Master University Hamilton Ontario, Canada: Prentice Hall. 1999. 842 p.

Kecman, V. **Learning and Soft Computing: Support Vector Machines, Neural Networks, and Fuzzy Logic Models**. Massachusetts Institute of Technology: MIT Press. 2001. 608 p.

Melcíades, W., Fiallos, M., & Pimentel, C. **Paralelização do Algoritmo "Backpropagation" em Clusters de Estações de Trabalho.** IV Congresso Brasileiro de Redes Neurais. p. 231-236. 20-22 julho, 1999. ITA, São José dos Campos - SP - Brasil,

Nordstrom, T., & Svensson, B. (1992). **Using and designing massively parallel computers for artificial neural networks.** Journal of Parallel and Distributed Computing, p. 260-285.

Pethick, M., Liddle, M., Werstein, P., & Huang, Z. Parallelization of a Backpropagation Neural Network on a *Cluster* Computer. In T. Gonzalez, **Proceedings of the Fifteenth**

IASTED International Conference on Parallel and Distributed Computing and Systems. Marina del Rey, USA. 2003. p. 574–582.

Priddy, K. L., & Keller, P. E. **Artificial Neural Networks: An Introduction.** SPIE Press 2005. 165p.

Rogers, R. O., & Skillicorn, D. B. **Using the BSP cost model to optimise parallel neural network training**. Future Generation Computer Science , 14, dez de 1998. p. 409-424.

Saramasinghe, S. **Neural Networks for Applied Sciences and Engineering**. Boston, USA: Auerbach Publications. 2006. 570 p.

Saratchandran, P., Sundrarajan, N., & Foo, S. K. **Parallel Implementations of Backpropagation Neural Networks on Transputers: a Study of Training Set Parallelism.** v. 3. World Scientific Publishing Co., Inc. 2000. 220 p.

Singer, A. **Implementations of artificial neural networks on the Connection Machine.** Parallel Computing , 14. p. 300-315. 1990.

Taboada, G. L., Teijeiro, C., & Touriño, J. **High Performance Java Remote Method Invocation for Parallel Computing on Clusters.** ISCC , p. 233-239. 2007.

Torresen, J., & Landsverk, O. A Review of Parallel Implementations of Backpropagation Neural Networks. In: N. Sundararajan, & P. Saratchandran**, Parallel Architectures for Artificial Neural Networks: Paradigms and Implementations.** Los Alamitos, CA, USA: IEEE Computer Society Press. 1998. p. 41-118.

ANEXO

Código Fonte

```
/*
 * TrainingServer.java
 */

package remote;

/**
 *
 * @author josé
 */
public interface TrainingServer extends java.rmi.Remote {

    public void addTrainingClient( String clientName ) throws
java.rmi.RemoteException;

}

/* TrainingServerImpl.java
 * classe que implementa TrainingServer
 */

package remote;

/**
 *
 * @author josé
 */
import classes.*;
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;
import java.util.Vector;


public class TrainingServerImpl extends UnicastRemoteObject implements
TrainingServer {

    private Vector clientsVector, threadsVector;
    private TrainingSet trainingSet;
    private BackpropagationNet neuralNet;
    private NetWeights deltaWeights, oldDeltaWeights;
    private NetWeights bufferWeights[];

    private boolean running = false;
    private boolean readable, writable;
    private int readed, writed;

    private int nClients;
    private double trainingError;
    private int actualEpoch;

    private double error = 0.001;
    private int epochs = 100000;
```

```
    long initTime,finishTime;
    private long time;

    //long updateTime;


    /** Creates new TrainingServerImpl */
    public TrainingServerImpl() throws RemoteException {

        super();

        try{
            String address = InetAddress.getLocalHost().getHostAddress();
            System.out.println( address);
            Naming.rebind("rmi://"+address+"/TrainingServer", this);

            clientsVector = new Vector();
        }

        catch( Exception e ) {
            e.printStackTrace();
        }
    }

    public void setNeuralNet( BackpropagationNet net ) {
        neuralNet = net;
    }
    public void setTrainingSet( TrainingSet ts ) {
        trainingSet = ts;
    }
    public void setError( double err ) {
        error = err;
    }
    public void setEpochs( int ep ) {
        epochs = ep;
    }

    public void addTrainingClient( String clientName ) throws
java.rmi.RemoteException {

        if( running() )
            return;

        try {
            clientsVector.addElement( Naming.lookup( clientName ) );
            System.out.println("cliente conectado: "+clientName );
        }
        catch ( Exception e ) {
            e.printStackTrace();
        }
    }

    public void stop() {
        running = false;
    }

    public void run() {

        running = true;
```

```
        configureClients();

        deltaWeights = (NetWeights)neuralNet.getNetWeights().clone();
        oldDeltaWeights = (NetWeights)neuralNet.getNetWeights().clone();
        oldDeltaWeights.clean();

        readable = true;
        neuralNet.training = true;

        initTime = System.currentTimeMillis();

        startThreads();


    }

    private void configureClients() {

         // calcula o numero de clientes 'nClients'
        nClients = clientsVector.size();

         // inicia o vetor de threads
        threadsVector = new Vector();

        bufferWeights = new NetWeights[nClients];

        for( int i = 0; i<nClients; i++ ) {

            try {
                  //para cada cliente:
                TrainingClient client =
(TrainingClient)clientsVector.elementAt(i);

                  //define a rede neural do cliente
                client.setNet( neuralNet );

                  //define o conjunto de treinamento do cliente
                client.setTrainingSet( trainingSet.getPart(i+1,
nClients) );

                  //cria uma thread para o cliente
                TrainingThread thread =  new TrainingThread( i, client );
                threadsVector.addElement( thread );
            }
            catch( Exception e ) {
                e.printStackTrace();
            }
        }
    }


      // inicia as threads
    private void startThreads() {

        for( int i = 0; i< threadsVector.size(); i++ ) {
            ((TrainingThread)threadsVector.elementAt( i )).start();
        }
    }

     // método usado pelas threads para coletar
```

```
 // os pesos atuais do Servidor
protected synchronized NetWeights getWeights()
{
    while( !readable ) {

        try {
            wait();
        }
        catch( InterruptedException e ) {
            e.printStackTrace();
        }
    }

    readed++;
    if( readed == nClients ) {
        readable = false;
        writable = true;
        readed = 0;

        notifyAll();
    }

    return neuralNet.getNetWeights();
}

  // método usado pelas threads para fornecer
  // ao Servidor os pesos recebidos dos Clientes
protected synchronized void setWeights( NetWeights weights, int
index )
{
    while( !writable ) {

        try {
            wait();
        }
        catch( InterruptedException e ) {
            e.printStackTrace();
        }
    }
    bufferWeights[index] = weights;

    writed++;
    if( writed == nClients ) {

        updateWeights();

        writable = false;
        readable = true;
        writed = 0;
        notifyAll();
    }
}

private void updateWeights()
{

    deltaWeights.clean();
    double actualError = 0;
    for( int i = 0; i<nClients; i++ )
    {
```

```
            deltaWeights.sum( bufferWeights[i] );
            actualError+= bufferWeights[i].getError();
        }

        neuralNet.getNetWeights().sum(deltaWeights);
        oldDeltaWeights.multiplicate( neuralNet.momentun );
        neuralNet.getNetWeights().sum(oldDeltaWeights);
        oldDeltaWeights = (NetWeights)deltaWeights.clone();

        actualError = actualError/trainingSet.size();  //// ts.size()-
1 ???
        actualEpoch++;

        neuralNet.netEpoch = actualEpoch;
        neuralNet.netError = actualError;

        if( actualError < error || actualEpoch == epochs )
        {
            finishTime = System.currentTimeMillis();
            time = finishTime-initTime;
            System.out.println( timeToString( time ) );
            running = false;
        }
    }


    public boolean running() {
        return running;
    }

    public double getTrainingError() {
        return trainingError;
    }
    public int getTrainingEpoch() {
        return actualEpoch;
    }

    public String getTrainingLog() {
        StringBuffer buffer = new StringBuffer();
        buffer.append("\n\n---------------------\n");
        buffer.append("Log de treiamento\n\n");
        buffer.append("Numero de clientes = "+nClients+"\n");
        if(!running){
            buffer.append("Tempo de treinamento =
"+timeToString(time)+"\n");
        }
        else{
            time = System.currentTimeMillis()-initTime;
            buffer.append("Tempo de treinamento =
"+timeToString(time)+"\n");
        }
        buffer.append("Epocas = "+getTrainingEpoch()+"\n");
        buffer.append("Erro = "+getTrainingError());

        return buffer.toString();
    }

    public String timeToString( long millis ) {
        long sec = (millis/1000)%60;
        long min = (millis/60/1000)%60;
```

```java
        long hrs = (millis/60/60/1000);

        String s = (String.valueOf(hrs) + "h:" + String.valueOf(min) +
"m:" + String.valueOf(sec) + "s" );
        return s;
    }

     // classe interna que define a thread de treinamento
    class TrainingThread extends Thread {

        NetWeights weights, clientDeltaWeights;
        int myIndex;
        TrainingClient client;

        public TrainingThread( int index, TrainingClient client )
        {
            myIndex = index;
            this.client = client;
        }

        public void run()
        {
            while( true ) {

                if(!running()){
                    return;
                }

                weights = getWeights();

                try {
                    clientDeltaWeights = client.getDeltaWeights(weights);
                }
                catch( RemoteException e ) {
                    e.printStackTrace();
                    return;
                }
                setWeights( clientDeltaWeights, myIndex );
            }
        }
    }
}
```

```java
/*
 * TrainingClient.java
 */

package remote;

import classes.*;

/**
 *
 * @author josé
 */
public interface TrainingClient extends java.rmi.Remote {
```

```
    public void setNet( BackpropagationNet net ) throws
java.rmi.RemoteException;

    public void setTrainingSet( TrainingSet ts ) throws
java.rmi.RemoteException;

    public NetWeights getDeltaWeights( NetWeights weights ) throws
java.rmi.RemoteException;

}
```

```
/*
 * TrainingClientImpl.java
 * classe que implementa TrainingClient
 */

package remote;

import classes.*;

/**
 *
 * @author josé
 */
public interface TrainingClient extends java.rmi.Remote {

    public void setNet( BackpropagationNet net ) throws
java.rmi.RemoteException;

    public void setTrainingSet( TrainingSet ts ) throws
java.rmi.RemoteException;

    public NetWeights getDeltaWeights( NetWeights weights ) throws
java.rmi.RemoteException;

}
```

```
/* Backpropagation.java
 *
 * A classe BackpropagationNet corresponde a uma rede neural feedforward.
 * Um objeto da classe BackpropagationNet possui um objeto NetWeights
 * e um vetor contendo objetos do tipo NeuronLayer.
 * Esta classe possui métodos para a manipulação da rede neural que
permitem
 * inserir camadas na rede, calcular as saídas da rede, e realizar o
treinamento
 * da rede utilizando os algoritmos backpropagation padrão e
batch,apartir
 * de um conjunto de treinamento TrainingSet
 */
```

```
package classes;

import classes.actfunc.ActivationFunction;
import java.util.Vector;

/**
 *
 * @author josé
 */
public class BackpropagationNet implements java.io.Serializable  {

    NetWeights netWeights;

    Vector layersVector;
    int inputs, lastLayerSize;

    public double learningRate = 0.02;
    public double momentun = 0.0;

    public boolean training = false;

    public int netEpoch;
    public double netError;

    /** Cria um objeto BackpropagationNet
     * com um numero de entradas igual a inputs **/
    public BackpropagationNet( int inputs ) {
        this.inputs = inputs;
        lastLayerSize = inputs;
        layersVector = new Vector();
        netWeights = new NetWeights();
    }
    /*adicina uma camada de neuronios de tamanho size com funçoes de
ativação af
     * parametro size é o tamanho da camada(numero de neuronios)
     * parametro af é um objeto que implementa as funçoes de ativação
     */
    public void addLayer( int size, ActivationFunction af ) {
        layersVector.addElement( new NeuronLayer( size, af ) );
        netWeights.addWeightsMatrix( lastLayerSize + 1, size ); ///
adiciona o bias
        lastLayerSize = size;
    }

    public NeuronLayer getLayer( int index ) {
        return (NeuronLayer)layersVector.elementAt( index );
    }

    public int nLayers() {
        return layersVector.size();
    }

    public NetWeights getNetWeights() {
        return netWeights;
    }
    public void setNetWeights( NetWeights nw ) {
        netWeights = nw;
    }
```

```
    /*calcula um vetor double[] contendo as saídas da rede
     *respectivas ao vetor de entrada inputs[]
     */
    public double[] computeNetOutput( double inputs[] ) {

        activateLayer( 0, inputs );

        for( int index = 1; index< nLayers(); index++ )
        {
            activateLayer( index, getLayer(index-1).getLayerOutputs() );
        }

        return getLayer( nLayers() -1 ).getLayerOutputs();
    }

    private void activateLayer( int layerIndex, double inputs[] ) {

        NeuronLayer nl = getLayer(layerIndex);

        int i;
        int j;

        for( j = 0; j< nl.size(); j++ )
        {
            nl.getLayerInputs()[j] = 0;      //limpa a entrada do
neuronio

            for( i = 0; i<inputs.length; i++ ) //calcula o valor de
entrada nos neuronios
            {
                nl.getLayerInputs()[j] +=
netWeights.getWeightsMatrix(layerIndex)[i][j] * inputs[i];
            }
            nl.getLayerInputs()[j] +=
netWeights.getWeightsMatrix(layerIndex)[i][j]; // BIAS
        }

        nl.activateNeurons();     //ativa os neuronios da camada
    }


    public void backpropagationLearn( TrainingSet ts, int epochs, double
error ) {

        NetWeights deltaWeights = (NetWeights)netWeights.clone();
        deltaWeights.clean();
        NetWeights oldDeltaWeights = (NetWeights)deltaWeights.clone();

        training = true;

        int actualEpoch = 0;
        double actualError;

        while( training )
        {
            actualError = 0.0;

            //para cada exemplo do conjunto de treinamento
            for( int i = 0; i < ts.size(); i++ )
            {
```

```
                //calcula as saidas dos neuronios da rede
                computeNetOutput( ts.getInput(i) );

                //calcula os termos de erro dos neuronios da rede
                computeNetError( ts.getTarget(i) );

                //calcula as variações de peso e atualiza os pesos da
rede
                computeDeltaWeights(ts.getInput(i), deltaWeights);

                actualError += errorFunction( ts.getTarget(i),
getLayer( nLayers() - 1).getLayerOutputs() );


                //atualiza os pesos da rede
                netWeights.sum(deltaWeights);
                oldDeltaWeights.multiplicate( momentun );
                netWeights.sum(oldDeltaWeights);
                oldDeltaWeights = (NetWeights)deltaWeights.clone();
            }


            actualEpoch++;
            actualError = actualError/ (ts.size()); /// **??** ts.size()
- 1

            System.out.println("* epoch=" + actualEpoch + "  error="
+actualError);
            if( actualEpoch == epochs || actualError <= error )
            {
                System.out.println("->epoch=" + actualEpoch + "  error="
+actualError);
                training = false;
            }
        }
    }

    public void backpropagationBatchLearn( TrainingSet ts, int epochs,
double error ) {

        //preenche os pesos com valores aleatórios
//        netWeights.randomize();

        NetWeights deltaWeights = (NetWeights)netWeights.clone();
        deltaWeights.clean();

        NetWeights oldDeltaWeights = (NetWeights)deltaWeights.clone();

        NetWeights bufferDeltaWeights = (NetWeights)deltaWeights.clone();

        training = true;

        int actualEpoch = 0;
        double actualError;

        while( training )
        {
            actualError = 0.0;
```

```
            //para cada exemplo do conjunto de treinamento
            for( int i = 0; i < ts.size(); i++ )
            {
                //calcula as saidas dos neuronios da rede
                computeNetOutput( ts.getInput(i) );

                //calcula os termos de erro dos neuronios da rede
                computeNetError( ts.getTarget(i) );

                //calcula as variações de peso
                computeDeltaWeights(ts.getInput(i), deltaWeights);

                bufferDeltaWeights.sum( deltaWeights );

                actualError += errorFunction( ts.getTarget(i),
getLayer( nLayers() - 1).getLayerOutputs() );
            }

            //atualiza os pesos da rede
            netWeights.sum( bufferDeltaWeights );
            oldDeltaWeights.multiplicate( momentun );
            netWeights.sum(oldDeltaWeights);
            oldDeltaWeights = (NetWeights)bufferDeltaWeights.clone();
            bufferDeltaWeights.clean();


            actualEpoch++;
            actualError = actualError/ (ts.size()); /// **??** ts.size()
- 1

            System.out.println("* epoch=" + actualEpoch + "  error="
+actualError);

            netEpoch = actualEpoch;
            netError = actualError;

            if( actualEpoch == epochs || actualError <= error )
            {
                System.out.println("->epoch=" + actualEpoch + "  error="
+actualError);
                training = false;

            }
        }
    }

    /*
     * este metodo retorna o erro referente a um elemento do conjunto de
treinamento */
    public double errorFunction( double target[], double outputs[] ) {

        double sum = 0.0;

        for( int i = 0; i< target.length; i++)
        {
            sum +=Math.pow( ( target[i] - outputs[i] ), 2 );
        }

        return sum;
    }
```

```
    public void computeNetError( double desiredOutputs[] ) {

        double d;
        double f,v;
        double sum;
        int i,j,k;

        NeuronLayer layer = getLayer( nLayers() - 1 );
        NeuronLayer nextLayer;

        //calcula o erro na camada de saída
        for( i = 0; i < layer.size(); i++ )
        {
            d = desiredOutputs[i];
            f = layer.getLayerOutputs()[i];
            v = layer.getLayerInputs()[i];

            layer.getLayerError()[i] = layer.derivativeAF(v) * (d-f);
        }

        //calcula o erro nas demais camadas
        for( k = nLayers()- 2; k >= 0; k--)
        {
            layer = getLayer( k );
            nextLayer = getLayer( k+1 );

            for( i = 0; i < layer.size(); i++ )
            {
                sum = 0.0;
                for( j = 0; j< nextLayer.size(); j++ )
                {
                    sum += nextLayer.getLayerError()[j] *
netWeights.getWeightsMatrix(k+1)[i][j];
                }

                f = layer.getLayerOutputs()[i];
                v = layer.getLayerInputs()[i];

                layer.getLayerError()[i] = layer.derivativeAF(v) * sum;
            }
        }

    }

    public void computeDeltaWeights( double inputs[], NetWeights nw ) {

        computeLayerDeltaWeights( 0, inputs, nw.getWeightsMatrix(0) );

        for( int k = 1; k<nLayers(); k++ )
        {
            computeLayerDeltaWeights( k, getLayer(k-1).getLayerOutputs(),
nw.getWeightsMatrix(k) );
        }
    }

    private void computeLayerDeltaWeights( int layerIndex, double
activation[], double deltaWeights[][] ) {

        double err=0;
```

```
        double o;
        int i, j;

        for( j = 0; j < getLayer(layerIndex).size(); j++ )
        {
            for( i = 0; i < activation.length; i++ )
            {
                err = getLayer(layerIndex).getLayerError()[j];
                o = activation[i];
                deltaWeights[i][j] = learningRate * err * o;
            }
            //err = getLayer(layerIndex).getLayerError()[j];
            deltaWeights[i][j] = learningRate * err;
        }
    }

    public boolean training() {
        return training;
    }

    public String testNet( TrainingSet ts ) {

        double in[];
        double targ[];
        double out[];
        double t;
        double o;
        StringBuffer buffer = new
StringBuffer("\ntarget\toutput\t|error|\n");
        double error= 0;
        for( int tsIndex = 0; tsIndex<ts.size(); tsIndex++ )
        {
            in = ts.getInput(tsIndex);
            targ = ts.getTarget(tsIndex);
            out = computeNetOutput( in );
            for( int outputIndex = 0; outputIndex<out.length;
outputIndex++ )
            {
                t = targ[outputIndex];
                o = out[outputIndex];

                error=Math.abs(t - o);

                buffer.append("(t["+outputIndex+"]=\t"+t);
                buffer.append("\to["+outputIndex+"]=\t"+o);
                buffer.append("\t|e|=\t"+error);
            }
            buffer.append("\n");
        }
        return buffer.toString();

    }
}


/*
 * NetWeights.java
 * classe que define as matrizes de pesos da rede
 */
```

```
package classes;

import java.util.Vector;

public class NetWeights extends Object implements Cloneable,
java.io.Serializable{

    private Vector weights;
    private double error;

    /** Creates new NetWeights */
    public NetWeights() {
        weights = new Vector();
    }

    public void addWeightsMatrix( int in, int out ) {

        double weightsMatrix[][] = new double[in][out]; // in+1 para os
bias
        weights.addElement( weightsMatrix );
    }

    public double[][] getWeightsMatrix( int index ) {
        return (double[][])weights.elementAt(index);
    }

    public void setError( double err ) {
        error = err;
    }
    public double getError() {
        return error;
    }

    public void clean() {
        for( int i = 0; i<weights.size(); i++ )
        {
            cleanArray( getWeightsMatrix( i ) );
        }
    }

    private void cleanArray( double array[][] ) {
        for(int i = 0; i< array.length; i++ )
        {
            for( int j = 0; j< array[0].length; j++ )
            {
                array[i][j] = 0.0;
            }
        }
    }

    public void randomize() {
        for( int i = 0; i<weights.size(); i++ )
        {
            randomizeArray( getWeightsMatrix( i ) );
        }
    }


    private void randomizeArray( double array[][] ) {
        for(int i = 0; i< array.length; i++ )
```

```
        {
            for( int j = 0; j< array[0].length; j++ )
            {
                //array[i][j] = 0.1;
                //array[i][j] = 0.2 * Math.pow( -1, i+j );

                array[i][j] = Math.random() - 0.5;
            }
        }
    }

    public Object clone() {

        NetWeights nw = new NetWeights();
        Vector v = new Vector();
        nw.weights = v;
        for( int i = 0; i<weights.size(); i++) {


v.addElement( array2dCopy( (double[][])weights.elementAt(i)) );
        }
        return nw;

    }

     //retorna uma copia de array
    private double[][] array2dCopy( double array[][] ) {

        double result[][] = new double[array.length][array[0].length];
        for( int i = 0; i< array.length; i++ )
        {
            for( int j = 0; j< array[0].length; j++ )
            {
                result[i][j] = array[i][j];
            }
        }
        return result;
    }


    public void sum( NetWeights nw ) {

        for( int w = 0; w< weights.size(); w++ ) {
            sumArrays( nw.getWeightsMatrix(w), this.getWeightsMatrix(w),
this.getWeightsMatrix(w) );
        }
    }
    private void sumArrays( double a1[][], double a2[][], double sum[][] )
{

        try{

            for( int i = 0; i<sum.length; i++ ) {
                for( int j = 0; j<sum[0].length; j++ ) {
                    sum[i][j] = a1[i][j] + a2[i][j];
                }
            }
        }
        catch( IndexOutOfBoundsException e ) {
            e.printStackTrace();
```

```
        }

    }

    public void multiplicate( double value ) {
        for( int i = 0; i<weights.size(); i++) {
            multiplicateArray( getWeightsMatrix(i), value );
        }
    }

    public void multiplicateArray( double array[][], double value ) {
        for( int i = 0; i<array.length; i++ ) {
            multiplicateArray( array[i], value );
        }
    }
    public void multiplicateArray( double array[], double value ) {
        for( int i = 0; i<array.length; i++ ) {
            array[i] = array[i] * value;
        }
    }

    public String toString() {

        StringBuffer sb = new StringBuffer();
        sb.append("\nNet Weights");
        for( int i = 0; i<weights.size(); i++ )
        {
            sb.append("\nMatrix "+i);
            sb.append( arrayToString( getWeightsMatrix(i) ) );
        }

        return new String(sb);
    }
    public String arrayToString( double array[] ) {

        StringBuffer sb = new StringBuffer();

        sb.append("[");
        for( int i = 0; i< array.length; i++ )
        {
            sb.append(array[i]);
            sb.append(", ");
        }
        sb.append("]");
        return new String(sb);
    }
    public String arrayToString( double array[][] ) {
        StringBuffer sb = new StringBuffer();

        for( int i = 0; i< array.length; i++ )
        {
            sb.append("\n");
            sb.append( arrayToString( array[i] ) );
        }
        return new String(sb);
    }

}
```

```
/* NeuronLayer.java
 *
 * A classe NeuronLayer corresponde a uma camada de neuronios de uma
 * rede neural(BackpropagationNet).
 * Guarda informações dos valores de entrada, dos valores de saída
 * e dos valores de erro de cada neuronio da camada.
 * Possui tambem uma referencia a uma implementação da classe
 * ActivationFunction que corresponde as funçoes de ativação dos
neuronios
 * da camada
 */

package classes;

import classes.actfunc.ActivationFunction;

/**
 *
 * @author josé
 */
public class NeuronLayer extends Object implements java.io.Serializable {


    private double[] layerError;
    private double[] inputs;
    private double[] outputs;

    private ActivationFunction af;

    /** Creates new NeuronLayer */
    public NeuronLayer( int size, ActivationFunction af) {

        inputs = new double[size];
        outputs = new double[size];
        layerError = new double[size];
        this.af=af;
    }

    public int size() {
        return inputs.length;
    }

    public void activateNeurons() {
        for( int i = 0; i< size(); i++ )
        {
            outputs[i] = actFunc( inputs[i] );
        }
    }

    public double[] getLayerOutputs() {
        return outputs;
    }

    public double[] getLayerInputs() {
        return inputs;
    }

    public double[] getLayerError() {
```

```
        return layerError;
    }

    public double actFunc( double v ) {
        return af.actFunc(v);
    }

    public double derivativeAF( double v ) {
        return af.derivActFunc(v);
    }
}




/*TrainingSet.java
 *
 * A classe TrainingSet define um conjunto de treinamento
 * Possui duas matrizes bidimensionais de mesmo tamanho
 * contendo as entradas e saidas desejadas do conjunto de treinamento.
 * Cada linha das matrizes correspondem respectivamente ao vetor de
entrada
 * e o vetor de saída desejada de um par de treinamentos.
 */

package classes;
/**
 *
 * @author josé
 */
public class TrainingSet implements java.io.Serializable {

    private double inputMatrix[][], targetMatrix[][];
    private int size;

    /** Creates a new instance of TrainingSet */
    public TrainingSet(double inputs[][], double targets[][]) {
        inputMatrix = inputs;
        targetMatrix = targets;
        size = inputs.length;
    }

    public int size() {
        return size;
    }

    public double[] getInput( int index ) {
        return inputMatrix[index];
    }

    public double[] getTarget( int index ) {
        return targetMatrix[index];
    }

    public TrainingSet getPart( int part, int nParts ) {

        int partialSize = size / nParts;
        int first = (part-1)*partialSize;
        int last = first + partialSize - 1;
```

```
        return new TrainingSet( partialCopy( inputMatrix, first,
last ),partialCopy( targetMatrix, first, last ) );

    }

    private double[][] partialCopy( double array[][], int initialIndex,
int finalIndex ) {

        int lines = finalIndex-initialIndex+1;
        int columns = array[0].length;

        double resultArray[][] = new double[lines][columns];
        for( int i = 0; i< lines; i++ )
        {
            for( int j = 0; j< columns; j++ )
            {
                resultArray[i][j] = array[ i+initialIndex ][ j ] ;
            }
        }
        return resultArray;
    }

    @Override
    public String toString() {
        StringBuffer sb = new StringBuffer();
        sb.append(this.inputMatrix[0].length+"; ");
        sb.append(this.targetMatrix[0].length+"; ");
        sb.append(this.size+"\n");
        for( int i = 0; i<this.size(); i++ )
        {
            sb.append( arrayToString(inputMatrix[i]) );
            sb.append( arrayToString(targetMatrix[i]) );
            sb.append("\n");
        }
        return new String(sb);
    }

    private String arrayToString( double array[] ) {
        StringBuffer sb = new StringBuffer();
        sb.append("[");
        for( int i = 0; i<array.length; i++ )
        {
            sb.append( String.valueOf( array[i] ) );
            sb.append(", ");
        }
        sb.append("]");

        return new String(sb);
    }
     // método que normaliza as saídas do conjunto de treinameto
     // entre -1 e 1
    public void normalizeBipolar() {

        double upperBound[] = new double[targetMatrix[0].length];
        double lowerBound[] = new double[targetMatrix[0].length];
        double midRange[] = new double[targetMatrix[0].length];
        double range[] = new double[targetMatrix[0].length];

        for(int i = 0; i<targetMatrix[0].length; i++)
        {
```

```
            upperBound[i] = Double.NEGATIVE_INFINITY;
            lowerBound[i] = Double.POSITIVE_INFINITY;
        }

        for( int t=0;t<targetMatrix.length;t++)
        {
            for( int i = 0; i<targetMatrix[0].length;i++)
            {
                if( upperBound[i] < targetMatrix[t][i])
                {
                    upperBound[i] = targetMatrix[t][i];
                }
                if( lowerBound[i] > targetMatrix[t][i])
                {
                    lowerBound[i] = targetMatrix[t][i];
                }
            }

        }

        for( int i=0; i<targetMatrix[0].length;i++)
        {
            midRange[i]=(upperBound[i]+lowerBound[i])/2;
            range[i]=upperBound[i]-lowerBound[i];
        }

        for( int l=0; l<targetMatrix.length; l++)
        {
            for( int c=0; c<targetMatrix[0].length; c++)
            {
                targetMatrix[l][c]=(targetMatrix[l][c]-
midRange[c])/(range[c]/2);
            }
        }
    }

}
```

```
/*
 * ActivationFunction.java
 *  interface que define os metodos para a função de ativação
 * e sua derivada
 */

package classes.actfunc;

/**
 *
 * @author josé
 */
public interface ActivationFunction {

    public double actFunc( double x );

    public double derivActFunc( double x );
```

```
}


/* ActFuncBipolarSigmoidal.java
 *
 * A classe ActFuncBipolarSigmoidal implementa a interface
ActivationFunction
 * e contem os metodos para o cálculo da função de ativação sigmoide
bipolar
 * e sua derivada.
 * a função de ativação sigmoide bipolar corresponde a:
 * f(x) = 2/(1 + exp(-x)) + 1 e retorna valores entre -1 e 1.
 * a derivada da função corresponde a:
 * f'(x) = 1/2(1-f(x))
 */

package classes.actfunc;

/**
 *
 * @author josé
 */
public class ActFuncBipolarSigmoidal implements java.io.Serializable,
ActivationFunction{

    public ActFuncBipolarSigmoidal() {
    }

    /*retorna o valor da função de ativação sigmoidal bipolar para x*/
    @Override
    public double actFunc(double x) {
        return 2d/(1d+Math.exp(-1d*x))-1d;
    }

    /*retorna o valor da derivada da função de ativação sigmoidal bipolar
para x*/
    @Override
    public double derivActFunc(double x) {
        return 0.5d*(1d-Math.pow(actFunc(x), 2d));
    }

}



/* ActFuncBinarySigmoidal.java
 * A classe ActFuncBinarySigmoidal implementa a interface
ActivationFunction
 * e contem os metodos para o cálculo da função de ativação sigmoide
binária
 * e sua derivada.
 * a função de ativação sigmoide binária corresponde a:
 * f(x) = 1/(1 + exp(-x)) e retorna valores entre 0 e 1.
 * a derivada da função corresponde a:
 * f'(x) = (1-f(x))f(x)
 */

package classes.actfunc;
```

```java
/**
 *
 * @author josé
 */
public class ActFuncBinarySigmoidal  implements java.io.Serializable,
ActivationFunction{

    public ActFuncBinarySigmoidal() {

    }

    /*retorna o valor da função de ativação sigmoidal binaria para x*/
    @Override
    public double actFunc(double x) {
        return 1/(1+Math.exp(x));
    }

    /*retorna o valor da derivada da função de ativação sigmoidal binaria
para x*/
    @Override
    public double derivActFunc(double x) {
        double o = actFunc(x);
        return (1d-o)*o;
    }

}
```