UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

Análise de desempenho da rede neural artificial do tipo multilayer perceptron na era multicore

Francisco Ary Alves de Souza

Orientador: Prof. Dr. Samuel Xavier de Souza

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e de Computação da UFRN (área de concentração: Engenharia de Computação) como parte dos requisitos para obtenção do título de Mestre em Ciências.

Número de ordem PPgEEC: M364
Natal, RN, 07 de Agosto de 2012

Análise de desempenho da rede neural artificial do tipo multilayer perceptron na era multicore

Francisco Ary Alves de Souza

Orientador: Prof. Dr. Samuel Xavier de Souza

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e de Computação da UFRN (área de concentração: Engenharia de Computação) como parte dos requisitos para obtenção do título de Mestre em Ciências.

Natal, RN, 07 de Agosto de 2012

Seção de Informação e Referência

Catalogação da publicação na fonte. UFRN / Biblioteca Central Zila Mamede

Souza, Francisco Ary Alves de

Análise de desempenho da rede neural artificial do tipo multilayer perceptron na era multicore / Francisco Ary Alves de Souza. – Natal, RN, 2012.

60 f. : il.

Orientador: Samuel Xavier de Souza.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Norte. Centro de Tecnologia. Programa de Pós-Graduação em Engenharia Elétrica e de Computação.

1. Computação Paralela – Dissertação. 2. Multilaser Perceptron – Dissertação. 3. OpenMP – Dissertação. I. Souza, Samuel Xavier de. II. Universidade Federal do Rio Grande do Norte. III. Título.

RN/UF/BCZM CDU 004.032.24

Análise de desempenho da rede neural artificial do tipo multilayer perceptron na era multicore

Francisco Ary Alves de Souza

Dissertação de Mestrado aprovada em 07 de agosto de 2012 pela banca examinadora composta pelos seguintes membros:

Prof. Dr. Samuel Xavier de Souza (orientador) ..... DCA/UFRN

Prof. Dr. Danniel Cavalcante Lopes ..... UFERSA

Prof. Dr. Allan de Medeiros Martins ..... DEE/UFRN

Dedico aos meus Pais: Ivanaldo Alves e Fátima de Souza; ao meus filhos: Fernanda Luiza e Gustavo Henrique, a minha esposa: Ketty Alves; pela paciência, incentivo e ajuda incondicional; e principalmente por sonhar comigo este sonho. Dedico também aos demais familiares e amigos pela ajuda direta ou indireta dada durante a realização deste.

Agradecimentos

A Deus por mais esta graça alcançada.

Ao meu orientador, a quem agradeço profundamente pela paciência e incentivo que muitas vezes extrapolaram a orientação.

Agradeço também aos demais professores da UFRN pelos diversos ensinamentos.

Agradeço ao amigos do DCA/UFRN, em especial ao amigo Adelson Luiz de Lima pelo incentivo e pelas diversas ajudas, aos demais alunos e funcionários pela apoio direto ou indireto que com toda certeza ajudaram para a conclusão deste trabalho.

À minha família pelo apoio durante esta jornada.

À CAPES, pelo apoio financeiro.

Resumo

As redes neurais artificiais geralmente são aplicadas à solução de problemas complexos. Em problemas com maior complexidade, ao aumentar o número de camadas e de neurônios, é possível conseguir uma maior eficiência funcional, porém, isto acarreta em um maior esforço computacional. O tempo de resposta é um fator importante na decisão de usá-las em determinados sistemas. Muitos defendem que o maior custo computacional está na fase de treinamento. Porém, esta fase é realizada apenas uma única vez. Já treinada, é necessário usar os recursos computacionais existentes de forma eficiente. Diante da era multicore esse problema se resume à utilização eficiente de todos os núcleos de processamento disponíveis. No entanto, é necessário considerar a sobrecarga existente na computação paralela. Neste sentido, este trabalho propõe uma estrutura modular que é mais adequada para as implementações paralelas. Propõe-se paralelizar o processo *feed-forward* (passo para frente) de uma RNA do tipo MLP, implementada com o OpenMP em uma arquitetura computacional de memória compartilhada. A investigação dar-se-á com a realização de testes e análises dos tempos de execução. A aceleração, a eficiência e a escalabilidade são analisados. Na proposta apresentada é possível perceber que, ao diminuir o número de conexões entre os neurônios remotos, o tempo de resposta da rede diminui e por consequência diminui também o tempo total de execução. O tempo necessário para comunicação e sincronismo está diretamente ligado ao número de neurônios remotos da rede, sendo então, necessário observar sua melhor distribuição.

Palavras-chave: Computação Paralela, Multilayer Perceptron, OpenMP.

Abstract

Artificial neural networks are usually applied to solve complex problems. In problems with more complexity, by increasing the number of layers and neurons, it is possible to achieve greater functional efficiency. Nevertheless, this leads to a greater computational effort. The response time is an important factor in the decision to use neural networks in some systems. Many argue that the computational cost is higher in the training period. However, this phase is held only once. Once the network trained, it is necessary to use the existing computational resources efficiently. In the multicore era, the problem boils down to efficient use of all available processing cores. However, it is necessary to consider the overhead of parallel computing. In this sense, this paper proposes a modular structure that proved to be more suitable for parallel implementations. It is proposed to parallelize the feedforward process of an RNA-type MLP, implemented with OpenMP on a shared memory computer architecture. The research consistes on testing and analizing execution times. Speedup, efficiency and parallel scalability are analyzed. In the proposed approach, by reducing the number of connections between remote neurons, the response time of the network decreases and, consequently, so does the total execution time. The time required for communication and synchronization is directly linked to the number of remote neurons in the network, and so it is necessary to investigate which one is the best distribution of remote connections.

Keywords: Parallel Computing, Multilayer Perceptron, OpenMP.

Sumário

Lista de Figuras

Lista de Símbolos e Abreviaturas

RNA: Rede Neural Artificial

MLP: MultiLayer Perceptron

API: Application Programming Interface

FPGA: Field Programmable Gate Arrays

RBF: Radial Basis Function

BFGS: Broyden Fletcher Goldfarb Shannon

SVM: Support Vector Machine

SISD: Single Instruction Single Data

SIMD: Single Instruction Multiple Data

MISD: Multiple Instruction Single Data

MIMD: Multiple Instruction Multiple Data

OpenMP: Open Multi-Processing

MPI: Message Passing Interface

KLT: Kernel-Level Thread

ULT: User-Level Thread

I/O: Input/Output

SO: Sistema Operacional

SMP: Symmetric Multiprocessing

Capítulo 1

Introdução

A ciência e a sociedade em geral, ao logo do tempo, cada vez mais aumentam a necessidade de resolver problemas complexos. Além disto, estes precisam ser resolvidos de forma rápida, sendo cada vez mais necessário o desenvolvimento de ferramentas computacionais. Para atender esta demanda crescente em responder questões ainda desconhecidas. Seja pela falta de uma ferramenta inteligente ou pela demora em responder a determinadas questões, o estudo da computação paralela e das Redes Neurais Artificiais (RNAs) passa a ser uma solução promissora.

O desenvolvimento da computação paralela criou o que chamamos de era *multicore*, que surge para atender a crescente necessidade de processamento. Nesta era, o poder computacional é disponibilizado por meio de um sistema de processamento distribuído ou com processadores paralelos ou ainda com vários núcleos.

Na era *multicore*, o desenvolvimento de *software* muda radicalmente. O aumento de desempenho que antes era obtido por *hardware*, com o aumento de frequência de operação dos processadores, agora requer uma solução via *software*, com intervenção direta do programador.

As RNAs são técnicas computacionais inspiradas na estrutura neural humana. De forma análoga ao cérebro humano, são capazes de adquirir e de generalizar conhecimento. Uma das suas principais características é a capacidade de solucionar problemas de natureza não-linear [Haykin 2001, da Silva et al. 2010, Turchenko e Grandinetti 2009*a*]. Em muito casos, estas são usadas em substituição aos complexos sistemas matemáticos descritos por dezenas de equações diferenciais [Hunter e Wilamowski 2011].

As redes neurais artificiais podem ter centenas ou milhares de neurônios (unidades de processamento). Os neurônios são organizados em camadas e se comunicam através de ligações conhecidas como sinapses. A essas ligações são associados pesos, também conhecidos como pesos sinápticos. A inteligência da rede está na interação entre os neurônios, ou seja, nas sinapses [Haykin 2001, da Silva et al. 2010].

A capacidade das RNAs em resolver problemas complexos pode aumentar ao adicionar um número maior de camadas e de neurônios. Em problemas com maior complexidade, ao aumentar o número de camadas e de neurônios, pode-se conseguir uma maior eficiência funcional. No entanto, este aumento acarretará em um maior esforço computacional, e como consequência, um maior tempo para resolver um determinado problema.

O tempo que a rede resolve um problema é um fator importante na decisão de usá-la em alguns tipos de sistemas. Para se ter uma rede realmente eficiente, é desejável que esta seja rápida e que suas respostas estejam dentro do esperado. Com uso de sistemas paralelos, uma rede eficiente resume-se a um algoritmo escalável [Turchenko e Grandinetti 2009*b*].

A rede perceptron de múltiplas camadas (MLP, do inglês *MultiLayer Perceptron*) é um tipo de RNA comumente usada na solução de problemas não linearmente separáveis. A característica interna de uma rede do tipo MLP a princípio não é adequada à paralelização, pois realiza uma comunicação bastante intensa entre os neurônios. Neste ponto de vista, é necessário propor soluções que viabilizem seu uso em sistemas paralelos.

Como mostrado por Udo Seiffert em seu trabalho, é possível obter respostas mais rápidas da rede com uso da computação paralela [Seiffert 2002]. No entanto, é necessário desenvolver algoritmos na perspectiva da era multicore.

A necessidade de usar as redes neurais cresce à medida que a quantidade de dados a serem observados e classificados aumenta. Um exemplo prático disso, é o que acontece no centro de pesquisas nuclear europeu, onde seus experimentos geram milhares de dados, o que cria a necessidade de um pré-processamento para identificar os dados que são realmente relevantes. Desta forma uma RNA, pode ser aplicada para realizar a classificação dos dados em relevantes ou não relevantes [de Llano e Bosque 2010].

As redes do tipo MLP, se comparadas com outras de outro tipo, são mais complexas de serem codificadas em programas paralelos, pois naturalmente elas possuem um número muito grande de conexões entre os neurônios o que aumenta a necessidade de comunicação e sincronismo. Nas implementações paralelas, a existência de *overhead* de comunicação não pode ser ignorado e, se não for corretamente tratado, baixos desempenhos podem ser observados.

Atualmente, as RNAs, são comumente aplicadas em problemas de classificação de dados, ou ainda, na modelagem de sistemas complexos [de Llano e Bosque 2010]. Estas análises tornam-se muito complexas e demoradas quando está envolvido um grande volume de dados, fazendo com que seja necessário um grande poder computacional. Portanto, a computação paralela pode vir a viabilizar o uso das RNAs nesse tipo de situação, ou ainda, em sistemas de tempo real e outros que precisem de uma resposta rápida.

A computação paralela, bem como o nome sugere, é uma forma de se realizar vários cálculos simultâneos, valendo-se de múltiplos elementos de processamento a fim de resolver um problema de tal forma que cada unidade de processamento execute uma pequena fatia de um dado problema.

Pode-se então dizer que, o paralelismo é uma técnica de dividir uma tarefa grande e complexa em tarefas menores, que serão distribuídas e executadas simultaneamente em vários processadores. Este procedimento pode aumentar o desempenho de um sistema e diminuir o tempo gasto para a solução de um problema [Lin e Snyder 2009].

Fazer com que os atuais programas consigam aproveitar a capacidade total de processamento ainda é um desafio. A codificação de programas paralelos é mais difícil. Em geral, nem sempre um computador com N processadores trabalhando paralelamente atinge seu desempenho máximo.

Com o intuito de tirar proveito do desempenho conseguido com a computação paralela, muitas implementações paralelas da RNA foram propostas [Strey 2004, Turchenko et al. 2010, Turchenko e Grandinetti 2009*b*, Turchenko e Grandinetti 2009*a*, de Llano e Bosque 2010, Hunter e Wilamowski 2011, de S. Alves et al. 2002, Lopes et al. 2009]. Porém, estas geralmente levam em consideração apenas o processo de treinamento.

O desempenho de um programa paralelo é medido por sua aceleração (*speedup*) em relação ao melhor algoritmo serial. A partir disso, também é possível avaliar a eficiência e escalabilidade paralela.

Neste trabalho, as análises levam em consideração apenas o tempo em que a rede leva para gerar uma saída a partir de uma entrada. Desta forma, esta proposta não leva em consideração o processo de treinamento, mas, uma vez correto que esse processo envolve diversas avaliações da rede, é correto afirma que a fase de treinamento também está sujeita as mesmas conclusões deste trabalho.

As análises foram feitas em um sistema com 24 cores AMD com memória compartilhada em linguagem C com o uso das diretivas do OpenMP.

1.1 Motivação

O estudo das RNAs mostra-se muito atraente e promissor, principalmente ao observar as suas diversas aplicações importantes para solucionar diversos problemas.

Entre as aplicações mais comuns de uma RNA estão o processamento digital de imagens, previsões e reconhecimento de padrões, robótica, otimização, entre outros [Turchenko e Grandinetti 2009*a*, da Silva et al. 2010, Haykin 2001].

A computação paralela não é um conceito novo para a computação, porém, antes do ano de 2005 era pouco explorada. Isso provavelmente aconteceu por ser necessário criar um novo paradigma para o desenvolvimento de *software*. A partir de 2005, com a popularização dos computadores *multicore*, a computação paralela vem se tornando a única solução para se conseguir maior poder computacional.

A importância desse conceito é observada ao notar seu potencial e o quanto este pode diminuir o tempo necessário para se obter uma resposta. Pesquisas nessa área apontam para a continuidade e crescimento da chamada era *multicore*, provocando os pesquisadores sobre o assunto no sentido desses desenvolverem melhores soluções, tanto para o *hardware* quanto para o *software*, de modo que se possa alcançar melhores *speedups* bem como maior eficiência e escalabilidade.

O paralelismo entre unidades de processamento, ou entre computadores *(grid, cluster, cloud computing)*, passa a ser regra e não apenas uma opção.

1.2 Objetivos

Os objetivos gerais deste trabalho consistem no desenvolvimento, implementação e investigação de um algoritmo paralelo para a rede neural artificial do tipo *Multilayer Perceptron*. A partir deste, será possível analisar o *speedup*, eficiência e escalabilidade paralela desta proposta.

Para atingir os objetivos gerais faz-se necessário os seguintes objetivos específicos:

- estudo das arquiteturas e algoritmos paralelos;
- estudo das redes neurais em especial a rede do tipo MLP;
- criar uma estratégia para a implementação eficiente de uma MLP na era multicore;
- codificar o algoritmo desenvolvido com uma interface de programação de aplicações (API, do inglês Application Programming Interface) que possibilite a compilação para uma arquitetura computacional paralela;
- Realizar uma quantificação dos resultados de forma analítica entre as implementações do código serial e do código paralelo para o caso em estudo.

1.3 Organização do texto

Esta dissertação está organizada em seis capítulos, sendo que, o capítulo 1 apresenta uma breve introdução ao desenvolvimento deste trabalho e uma breve descrição sobre

redes neurais artificiais, computação paralela, motivação para o desenvolvimento desse trabalho e os objetivos desejados.

O capítulo 2 apresenta de forma resumida alguns trabalhos desenvolvidos com objetivos correlatos com este trabalho e o que os autores relataram de seus trabalhos, bem como os seus resultados.

O capítulo 3 apresenta uma breve descrição das principais características, evolução histórica e os principais tipos de rede neurais artificiais com uma atenção especial a rede *Multilayer Perceptron*, objetivo principal de estudo deste trabalho.

O capítulo 4 apresenta os conceitos fundamentais relativos a computação paralela seu funcionamento, tipos de computação paralela, conceituação de *thread*, como criar, gerenciar e destruir threads, API para codificação, com especial atenção as diretivas do OpenMP.

O capítulo 5 é apresenta a concepção e o desenvolvimento de uma estratégia para a codificação paralela de um rede neural do tipo multilayer perceptron, chamada de MLP Modular que, como poderá ser visto, o desempenho dessa estratégia está diretamente ligado ao número de conexões (sinapses) remotas.

E, finalmente, no capítulo 6, apresentaremos a conclusão deste trabalho, os pontos que achamos importantes e os possíveis trabalhos que poderão ser desenvolvidos no futuro.

Capítulo 2

Revisão Bibliográfica

A revisão bibliográfica foi realizada de forma analítica e crítica de algumas das publicações sobre o tema em estudo. Teve o objetivo de: verificar os textos publicados sobre o assunto; conhecer a forma como este assunto foi abordado; e de conhecer quais são as variáveis e problemas envolvidos neste estudo. A seguir, apresentamos resumos de alguns trabalhos que consideramos relevantes para a realização deste trabalho. Porém, será possível verificar que a maioria das pesquisas publicadas concentram-se em observar a paralelização da fase de treinamento da rede neural.

No trabalho realizado por Volodymyr Turchenko [Turchenko et al. 2010] os autores afirmam que as Redes Neurais Artificiais representam uma boa alternativa em substituição aos métodos matemáticos tradicionais, bem como, para a solução de problemas complexos em várias áreas, como por exemplo: processamento de imagem, previsões e reconhecimento de padrões, robótica, otimização, entre outras. Porém, estas requerem elevada carga computacional, especialmente na fase de treinamento, fase esta, que pode demorar várias horas ou dias.

No entanto, eles afirmam que o uso de computadores de alto desempenho com processadores paralelos, podem diminuir o tempo necessário para a fase de treinamento. Eles concluem que a simples paralelização do algoritmo de treinamento *backpropagation* não é escalável devido à grande sobrecarga de comunicação e sincronização entre os processadores paralelo.

Para sanar o problema de sobrecarga eles sugeriram usar o algoritmo de treinamento por lote, onde a atualização dos pesos sinápticos é realizada no final de cada época do treinamento, ou seja, após o processamento de todos os padrões de treinamento, em vez de atualizar os pesos após cada padrão apresentado, como acontece no modo de treino clássico do algoritmo *backpropagation*.

A implementação inicial do algoritmo proposto por eles foi implementado com uso do Open MPI, o que possibilitou observar uma aceleração (*speedup*) positiva. Os expe-

rimentos iniciais foram realizados em um computador com arquitetura computacional do tipo simétrica, este tipo tem como característica, ter o mesmo custo para acesso a memória entre as unidades de processamento. As eficiências obtidas de 74,3%, 43,5% e 22,1% para MLP com arquitetura 5-5-1 (36 conexões), 87,8%, 64,4% e 38,2% para MLP 10-10-1 (121 conexões) e 91,1%, 71,7% e 46,7% para MLP 15-15-1 (256 conexões), respectivamente, em 2, 4 e 8 processadores paralelos para um cenário de 200 padrões de treinamento.

As figuras 2.1 e 2.2 mostram os resultados obtidos com o seguinte cenário experimental de tamanho crescente, para as seguintes arquiteturas de rede: 5-5-1 (36 conexões), 10-10-1 (121 conexões), 15-15-1 (256 conexões), 20-20-1 (441 conexões), 30-30-1 (961 conexões), 40-40-1 (1681 conexões), 50-50-1 (2601 conexões) e 60-60-1 (3721 conexões).

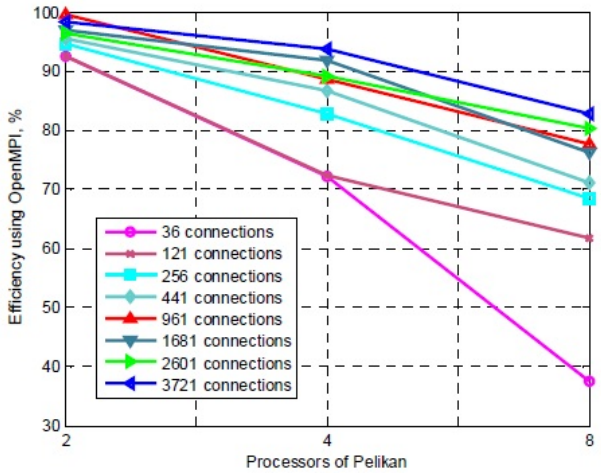

Figura 2.1: Gráfico de execuções com variado número de conexões sinápticas, em uma arquitetura computacional com acesso a memória de forma simétrica [Turchenko et al. 2010]

Já o autor Alfred Strey [Strey 2004], discute e compara várias possibilidades para a execução paralela de uma rede neural artificial do tipo função de bases radial (RBF, do inglês Radial Basis Function) realizadas em um ambiente computacional simétrico executadas em um computador SunFire 6800. As implementações paralelas são baseadas no OpenMP e MPI.

Neste trabalho são apresentados três métodos de particionamento para a implementação paralela da rede RBF, que são: soma parcial, método de re-cálculo e método simplificado. A rede RBF é um tipo de rede neural artificial tipicamente aplicada em problemas de aproximação de funções e classificação de padrões e é constituída por uma camada de entrada, uma camada oculta e uma camada de saída.

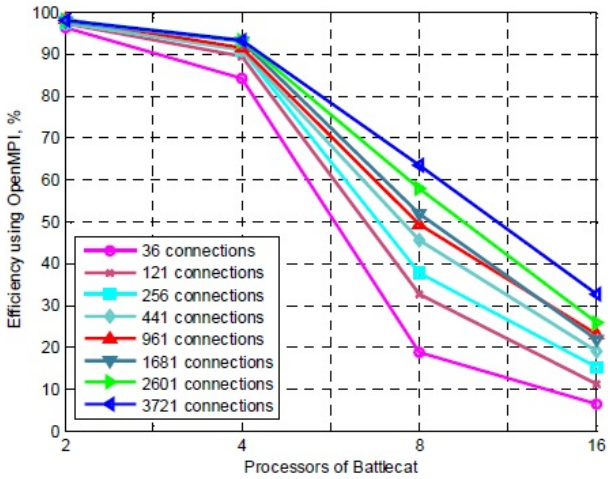

Figura 2.2: Gráfico de execuções com variado número de conexões sinápticas, em um cluster de computadores [Turchenko et al. 2010]

Uma breve explicação de cada método de particionamento proposto é apresentado abaixo, para maiores detalhes consulte o artigo "A comparison of OpenMP and MPI for neural network simulations on a SunFire 6800" [Strey 2004].

- Método soma parcial: para cada sinapse entre dois neurônios presente na mesma unidade de processamento é realizado o cálculo parcial e depois é realizada uma operação de redução ou uma soma geral no fim da operação específica;
- Método de re-cálculo: a rede é distribuída entre as unidade de processamento e no final realiza uma junção entre as divisões de trabalho; e
- Método simplificado: uma copia da rede é distribuída a cada unidade de processamento, sendo realizada apenas uma única comunicação no fim da operação.

Este autor optou por implementar os três métodos. O que possibilitou observar que o *speedup* depende fortemente da estratégia de particionamento selecionada. Para as implementações com o OpenMP, o método re-cálculo ofereceu mais desempenho do que o método simplificado. Apenas com uma ou duas unidades de processamento, o método re-cálculo é mais lento devido a sua maior comunicação e sincronismo. O método da soma parcial é extremamente lento. Porque o *speedup* permanece sempre abaixo de 1. Para as implementações com MPI, o método simplificado geralmente leva a melhores desempenhos, especialmente para grandes redes neurais onde é possível obter um *speedup* aproximadamente linear.

As figuras 2.3 e 2.4 mostram os resultados para implementações com o OpenMP e com o MPI em uma de rede do tipo RBF com arquitetura fixa e variação dos métodos.

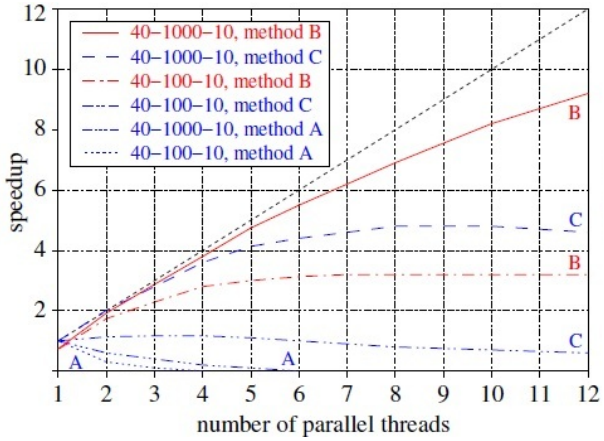

Figura 2.3: Resultados para implementação com o OpenMP dos métodos A, B, C com variação na camada oculta [Strey 2004]

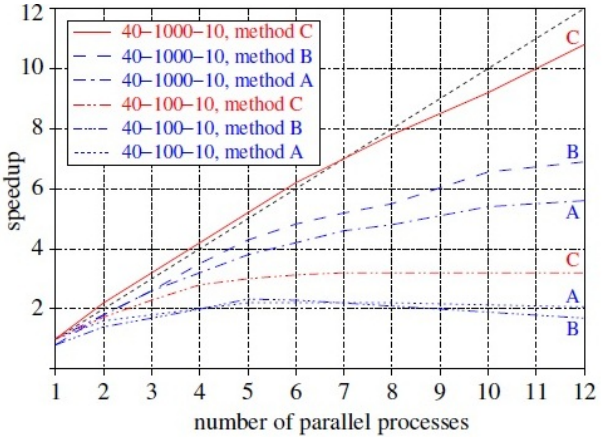

Figura 2.4: Resultados para implementação com o MPI dos métodos A, B, C com variação na camada oculta [Strey 2004]

Alfred Strey, conclui que as implementações com MPI revelaram-se ser ligeiramente mais rápido do que as implementações com OpenMP e que os desempenhos observados mostram que é necessário optar por um método correto de particionamento.

No trabalho de Danniel Lopes [Lopes et al. 2009] é proposto o desenvolvimento de uma aplicação com uso de uma rede do tipo máquina de comitê implementada em FPGA (do inglês, Field Programmable Gate Arrays) denominada como Modular Rede Neural, a ser aplicado para a observação e classificação de distúrbios na distribuição de uma concessionária de rede elétrica.

A estrutura desse tipo de rede neural usa o conceito de dividir para conquistar. De acordo com os autores, uma das vantagens deste tipo de rede é seu rápido processo de treinamento, pois é realizado de forma independente por cada especialista. Na figura 2.5,

é fácil ver, que a rede proposta, pode ser facilmente dividida, ou seja, cada especialista tem independência, e pode ser executado em diferentes tarefas, ou ainda distribuídos a diferentes processadores. Para maiores detalhes sobre máquina de comitê consultar Simon Haykin [Haykin 2001].

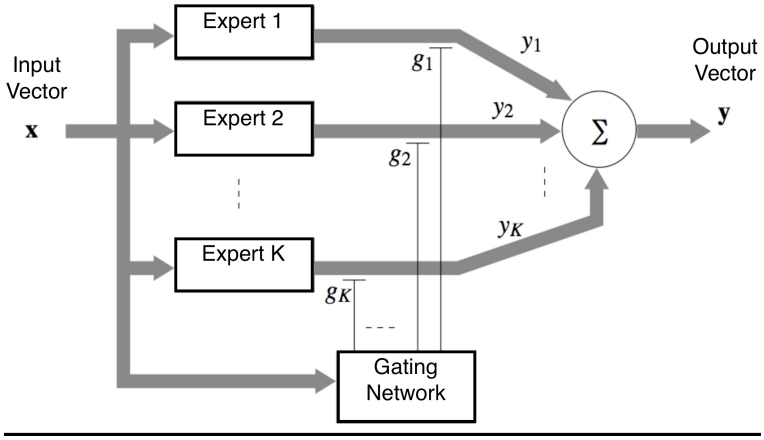

Figura 2.5: Diagrama de uma máquina de comitê baseada na Modular Rede Neural [Lopes et al. 2009]

Neste sentindo, basta criar uma tarefa para cada especialista que na concepção desse tipo de rede são processos independentes. Nesse caso de estudo a aplicação da Modular Rede Neural foi aplicada para a classificação de distúrbios elétricos que poderiam ter quatro possibilidades. Sua aplicação foi testada em um FPGA do tipo Nios II que é um processador programável desenvolvido pela Altera usado para diferentes aplicações. Os resultados foram obtidos usando um Nios II com 4 processadores.

Na figura 2.6, é possível observar um diagrama do algoritmo paralelo dividido em quatro pequenas tarefas e executado em quatro processadores. Sendo que cada especialista foi implementado em um processador diferente, bem como o *gating* (Rede de Passagem) e a função soma. Neste trabalho os autores decidiram implementar o *gating* e a função soma no mesmo processador, pois assim conseguiu-se uma menor comunicação entre os processos.

A proposta de Robinson Alves [de S. Alves et al. 2002] em seu trabalho foi o desenvolvimento de um algoritmo com o objetivo de diminuir o tempo necessário para o treinamento de uma rede neural do tipo *Multilayer Perceptron*, utilizando o algoritmo de treinamento *backpropagation*. Este também faz uma análise de diferentes estratégias utilizadas para a paralelização.

Nesse trabalho os autores observam cinco níveis de paralelismo que podem ser explorados no processo de treinamento proposto. Os níveis de paralelismo observados são a

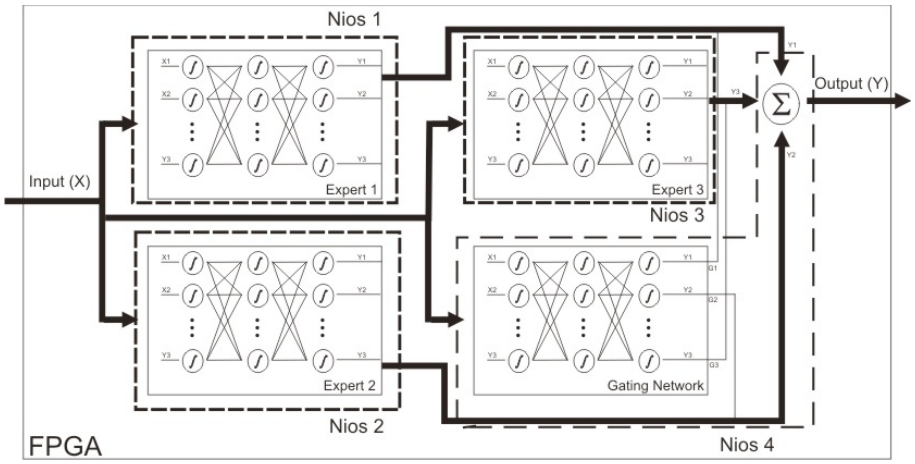

Figura 2.6: Diagrama da Modular Rede Neural executada em um Nios II com 4 processadores [Lopes et al. 2009]

nível: de rede, conjunto de treinamento, de camada, de neurônio, e de sinapse.

Eles observam que a paralelização da fase de treinamento proporcionará um menor tempo para conclusão desta fase. Além, de viabilizar a aplicação da rede na solução de problemas de maior complexidade. No entanto, as possibilidades demonstradas foram na paralelização pela:

- Decomposição conjunta do treinamento: consiste em dividir o treinamento em vários subconjuntos, onde cada um é usado para treinar uma cópia da rede. Todas as cópias são inicializadas com o mesmo peso sináptico. Em que cada cópia fica associada a uma unidade de processamento;
- Decomposição funcional ou por camada: consiste em associar cada camada da rede a uma correspondente unidade de processamento, como pode ser observado na figura 2.7;
- Decomposição por neurônio: no paralelismo por neurônio, cada unidade de processamento fica responsável pela execução de um conjunto de neurônios organizados em linha, como pode ser observado na figura 2.8. Porém, antes de inicializar o cálculo do próximo neurônio, é necessário esperar a saída do neurônio da camada anterior, sendo necessário uma maior comunicação entre as unidades de processamento, o que é indesejável, e que inviabiliza o uso desta estratégia.

Para testar a proposta os autores implementaram um algoritmo com o objetivo de compactar uma imagem e obtiveram resultados de *speedup* que chegaram a 64.

No trabalho de Turchenko [Turchenko e Grandinetti 2009*a*] foi realizada uma investigação para analisar a eficiência do BP em lote. Porém, devido a grande sobrecarga de

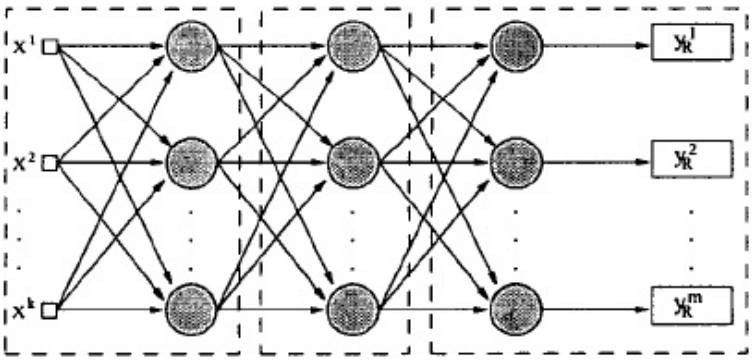

Figura 2.7: Decomposição por camada [de S. Alves et al. 2002]

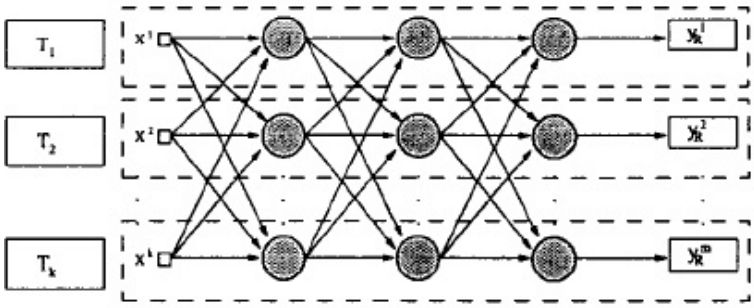

Figura 2.8: Decomposição por neurônio [de S. Alves et al. 2002]

comunicação, é necessário otimizar o algoritmo implementado com uso de funções de comunicação melhorada, proporcionando um paralelismo mais eficiente. Devido à alta comunicação e sincronismo entre os processadores paralelos, a simples paralelização do algoritmo sequencial BP não ocasionaria um melhor desempenho computacional. Com o objetivo de obter melhores resultados foi proposto um algoritmo de treinamento em lote paralelizado e testado em um sistema heterogêneo do tipo *grid*.

Os autores Rafael Menéndez de Llano e José Luis Bosque [de Llano e Bosque 2010] utilizaram o algoritmo de otimização quase newton (BFGS, do ingês Broyden Fletcher Goldfarb Shannon) divididos em BFGS-linear e BFGS-linear híbrido. Estes algoritmos são usados para encontrar valores que minimizem a função de erro para os pesos sinápticos. Por serem mais robustos, necessitam de maiores recursos computacionais. Os melhores speedup foram obtidos para redes neurais de grande porte, como por exemplo, as com 4 camadas divididas em: entrada com 16 neurônios, duas ocultas cada uma com 50 neurônios e um de saída com 1 neurônio, (16-50-50-1) e em alguns casos, até *speedup* superlinear (15-50-50-1). No entanto, para redes pequenas (16-10-10-1) os resultados não foram satisfatórios.

2.1 Considerações sobre o capítulo

No geral, os trabalhos demostram a viabilidade no estudo e paralelização das redes neurais, principalmente, as de médio e de grande porte, geralmente usadas para solucionar problemas que envolvam uma grande quantidade de variáveis. Já para redes de menor porte, os resultados não são bons, ou até mesmo inferiores aos executados em sistemas de apenas um unidade de processamento. Nesse caso, é necessário observar a aplicação a qual deseja-se uma solução.

Capítulo 3

Redes Neurais Artificiais

As Redes Neurais Artificiais (RNA) são técnicas computacionais inspiradas na estrutura neural de organismos inteligentes e que adquirem conhecimento através da experiência. Extrai seu poder computacional através de sua estrutura massivamente paralela e de sua habilidade de aprender e de generalizar seu aprendizado. A generalização se refere ao fato da rede produzir saídas adequadas para entradas que não estavam presentes durante o treinamento [Haykin 2001, da Silva et al. 2010].

As pesquisas das redes neurais artificiais iniciaram em 1943 com o trabalho pioneiro de McCulloch e Pitts. Esses descreveram o primeiro modelo matemático inspirado nos neurônios biológicos unificando os estudos da neurofisiologia e da lógica matemática.

Até o final da década de 1960 os estudos das redes neurais recebeu uma enorme atenção da comunidade científica, porém após a publicação de Marvin Minsky e Seymour Papert o interesse nesta área diminuiu. Eles demostraram de forma enfática a limitação das redes neurais artificiais, ou seja, a impossibilidade da rede em realizar a correta classificação de padrões não linearmente separáveis.

Os trabalhos de Hopfiled em 1982 e o de Rumelhart e McClelland em 1986 são considerados as publicações mais influentes e foram responsáveis pelo ressurgimento e interesse pelas redes neurais artificiais.

De acordo com Ivan Silva [da Silva et al. 2010] a retomada definitiva das pesquisa se deve entre outros fatores ao desenvolvimento de computadores com maior capacidade de processamento, a criação de algoritmos de otimização mais eficientes e robustos e as novas descobertas sobre o sistema nervoso biológico.

As RNAs tem como características principais:

- Capacidade de aprendizado;
- Capacidade de generalizar o conhecimento;
- Armazenar o conhecimento de forma distribuída; e
- Tolerância a falhas;

A estrutura da RNA foi concebida a partir de modelos conhecidos dos sistemas biológicos. Os neurônios artificiais são modelos bem simplificados dos neurônios biológicos. Paralelismo e alta conectividade, características principais das redes biológicas, também estão presentes nas RNAs.

As redes artificiais se comparam as redes biológicas pelas seguintes características:

- O conhecimento é adquirido por um processo de aprendizagem;
- O conhecimento é armazenado nas conexões existentes entre os neurônios as chamadas sinapses; e
- A capacidade de generalizar o conhecimento.

O funcionamento de uma RNA foi inicialmente proposto em 1943 por Warren McCulloch e Walter Pitts, e ainda é o mais utilizado nas diferentes arquiteturas da RNA, descrito ligeiramente a seguir:

- é dada a rede uma ou várias entradas $(x_1, x_2, \cdots, x_n)$;
- as entradas são ligadas ao neurônio através da sinapse que a ela é associado um peso w, onde em cada ligação se realiza uma operação de multiplicação;
- em seguida é realizada um somatório $(\sum)$ no neurônio para cada entrada a ele ligado. Nessa operação é produzida uma saída u; e
- essa u é submetido a uma função de ativação $f(u)$ que produz uma saída y.

Na equação 3.1 e figura 3.1 é possível observar o modelo de um neurônio artificial:

$$y = f(u), u = \sum_{i=1}^{n} w_i \cdot x_i - Bias, \tag{3.1}$$

sendo o Bias (θ) um limiar usado para a saída u do neurônio.

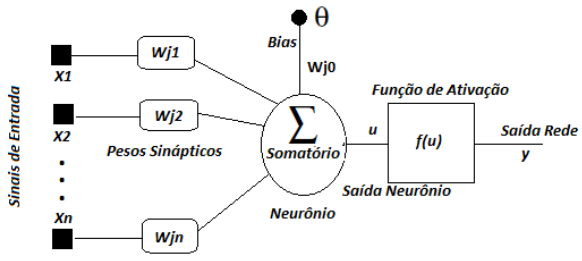

Figura 3.1: Neurônio artificial proposto por McCulloch e Pitts.

A utilização de uma RNA na solução de um determinado problema passa inicialmente por uma fase de treinamento, em que a rede extrai características relevantes de padrões de

informações apresentados para a mesma, criando dessa forma uma representação própria para o problema.

A etapa de treinamento consiste em um processo iterativo de ajuste de parâmetros da rede, ou seja, ajustes nos pesos sinápticos, responsáveis por guardar o conhecimento que a rede adquiriu do ambiente em que está operando.

A proposta de McCulloch e Pitts, não apresentava nenhuma técnica de treinamento. As técnicas de treinamento e aprendizado começaram a ser estudas por Donald Hebb em 1949, este mostrou em seu trabalho que a variação dos pesos sinápticos poderia tornar a rede artificial apta ao que chamamos de aprendizado.

Já em 1958, Frank Rosenblatt em fim propôs o primeiro modelo de uma RNA capaz de ser treinada a parti de um algoritmo de treinamento, e com capacidade de realizar o ajuste nos pesos sinápticos. No entanto, esse modelo possibilitava o aprendizado de padrões linearmente separáveis.

Em 1986 com o trabalho de Rumelhart, Hinton e Willians que proporão um algoritmo de treinamento nominado de *backpropagation*, foi possível realizar o treinamento de uma RNA de múltiplas camadas e pois fim a limitação das RNAs em resolver problemas que não fossem lineares.

A arquitetura de uma RNA define a forma como seus diversos neurônios estarão distribuídos. Em geral, essa pode ser dividida em três partes, camada de entrada, camada oculta e camada de saída. Respectivamente, capta os dados de entrada, processa os dados de entrada e na camada de saída é apresentado o resultado final da rede.

A partir das diversas arquiteturas, surgiram diversos tipos de RNAs. Nas seções 3.1 à 3.4, apresentamos alguns tipos com uma breve explicação.

Todos os tipos de redes neurais possuem algum processo de treinamento e aprendizado, que consiste em aprender com o ambiente e com isso melhorar seu desempenho ou respostas. Esta propriedade é a mais importante em uma RNA. Esse processo consiste em modificar os pesos sinápticos que são ajustados de acordo com os padrões de treinamento apresentados [Haykin 2001, da Silva et al. 2010].

O treinamento é realizado através de um processo iterativo de ajustes aplicado aos pesos sinápticos da rede. O aprendizado ocorre quando a rede atinge uma solução generalizada para uma classe de problemas.

O algoritmo de treinamento proposto por Hebb motivou os primeiros métodos de aprendizado para uma RNA. Este propõe que o peso de uma conexão sináptica deva ser ajustado, caso exista sincronismo entre os níveis de atividade da entrada e saída. No caso em que dois neurônios, em lados distintos da sinapse, são ativados sincronizadamente, teremos um fortalecimento dessa sinapse. Entretanto, se os neurônios forem ativados as-

sincronamente, a sinapse será enfraquecida ou mesmo eliminada. Em outras palavras, se o neurônio pré-sináptico tiver grande influência na ativação do neurônio pós-sináptico, a conexão entre eles deve ser reforçada.

Desta forma, um algoritmo de treinamento deve possuir um conjunto de regras bem definidas para a solução de um determinado problema. Para cada tipo de rede existe um algoritmo específico, estes algoritmos diferem um dos outros principalmente pelo modo como os pesos são modificados.

Outro fator importante é a maneira pela qual uma rede se relaciona com o ambiente. Nesse contexto existem os seguintes paradigmas de aprendizado:

- **Aprendizado supervisionado**: quando é utilizado um agente externo (professor) que indica se a resposta da rede é a desejada para o padrão de entrada;
- **Aprendizado não supervisionado** (auto-organização): quando não existe uma agente externo indicando a resposta desejada para os padrões de entrada. A rede atualiza seus pesos sem o uso de pares entrada/saídas desejadas e sem indicações sobre a adequação das saídas produzidas;
- **Aprendizado por reforço**: para cada entrada apresentada é produzida uma indicação (reforço) sobre a adequação das saídas correspondentes produzidas pela rede, ou seja, é quando um crítico externo avalia a resposta fornecida pela rede.

O algoritmo de treinamento possui um conjunto de procedimentos bem definidos para adaptar os parâmetros de uma RNA, afim de que a mesma possa aprender uma determinada função. Como dito aqui, exite vários algoritmos de aprendizado. No geral, temos um conjunto de ferramentas representadas por diversos algoritmos, cada qual com suas vantagens e desvantagens.

De forma resumida, o treinamento de uma rede neural é um processo iterativo e sua qualidade está diretamente relacionada com três parâmetros: ao tamanho da rede neural, ao número de padrões utilizados e ao número de épocas.

O tamanho da rede é medido de acordo com o número de camadas, de neurônios e de sinapses que deve ser adequado ao tamanho do problema a ser resolvido. Já o número de padrões apresentados a rede no processo de treinamento deve refletir adequadamente a situação a ser resolvida. O número de épocas também afeta a qualidade de resposta da rede. No caso em que a quantidade de épocas é pequena, a rede pode não aprender adequadamente, já um número elevado de épocas pode provocar o chamado *overtraining* ou sobre-ajuste da rede. Isso ocorre quando a rede memoriza os padrões fornecidos, tornando-se incapaz de generalizar.

A função de ativação serve para restringir a amplitude da saída de um neurônio. De

acordo com Haykin [Haykin 2001], essa função também pode ser referida com função restritiva já que limita o intervalo permissível de amplitude do sinal de saída a um valor finito. Este intervalo geralmente está entre 0 (zero) e 1 (um) ou entre -1 (menos um) e 1 (um). Entre as funções de ativação podemos citar:

Função de Limiar (Degrau), a saída do neurônio que emprega esta função é demostrada na equação 3.2 e figura 3.2:

$$f(u) = \begin{cases} 1 & \text{se } u \geq 0 \\ 0 & \text{se } u < 0 \end{cases} \tag{3.2}$$

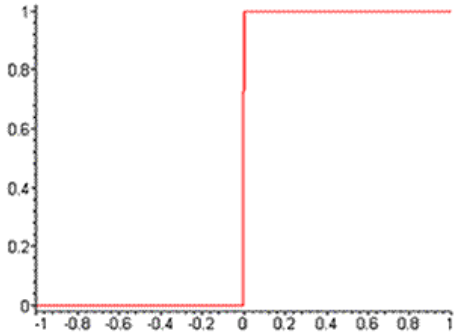

Figura 3.2: Gráfico da função de Limiar

Neste modelo, a saída de um neurônio assume o valor 1 (um), se o campo local induzido daquele neurônio é não negativo, e assume o valor 0 (zero) caso contrario.

Função linear, usada em problemas que estão separados linearmente. Também utilizada em redes com finalidade de aproximador universal de função. A representação gráfica dessa função é ilustrada na figura 3.3.

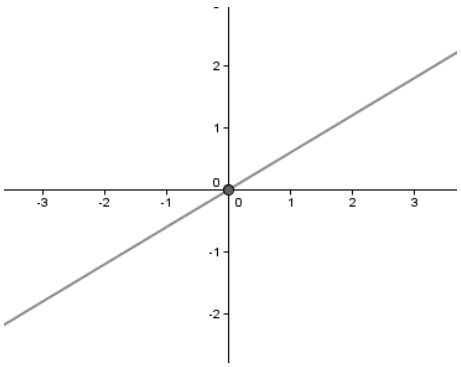

Figura 3.3: Função Linear

Função linear por partes, a saída do neurônio que emprega essa função é demostrada na equação 3.3:

$$f(u) = \begin{cases} 1, & u \geq +\frac{1}{2} \\ u, +\frac{1}{2} > & u > -\frac{1}{2} \\ 0, & u \leq -\frac{1}{2} \end{cases} \tag{3.3}$$

Nesse modelo, assume-se que o fator de amplificação dentro da região linear de operação é a unidade. Essa pode ser vista como uma aproximação de um amplificador não-linear.

Função Sigmoide, a saída do neurônio que emprega essa função tem a forma de um *S*. Essa é a função mais comumente aplicada para uma RNA. Assume comportamento linear e não-linear.

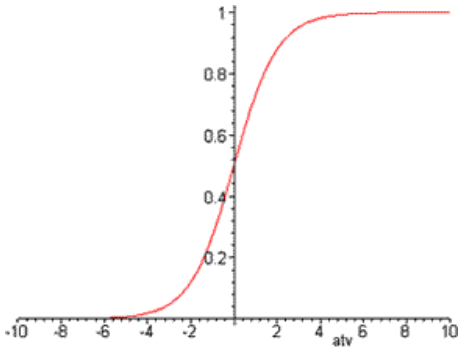

Figura 3.4: Função Sigmoide

Funções sigmoide comumente usadas:

Função logística, demostrada na equação 3.4 e figura 3.5:

$$f(u) = \frac{1}{1 + e^{-\beta \cdot u}}, \tag{3.4}$$

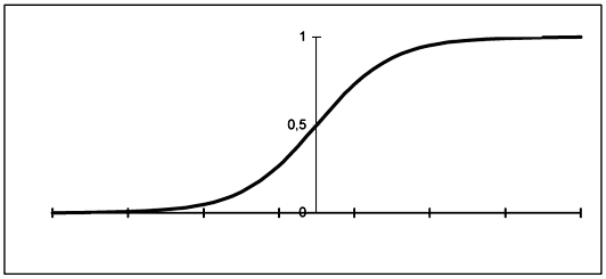

Figura 3.5: Função Logística

Função tangente hiperbólica, demostrada na equação 3.5 e figura 3.6:

$$f(u) = \frac{1 - e^{-\beta \cdot u}}{1 + e^{-\beta \cdot u}}, \tag{3.5}$$

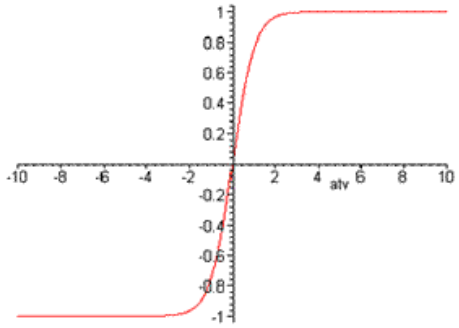

Figura 3.6: Função tangente hiperbólica com limitação superior $+1$ e inferior -1

As áreas potencias de aplicação das RNAs estão relacionadas com as engenharias e as ciências, alguns exemplos: aproximador universal de funções, controle de processos (robótica, aeronaves, satélites), reconhecimento ou classificador de padrões, agrupamento de dados (clusterização), sistemas de previsão, otimização de sistemas, memória associativa, etc..

Para maiores detalhes sobre RNA e suas aplicações consultar Simon Haykin e Ivan Silva [Haykin 2001, da Silva et al. 2010]:

3.1 Rede Perceptron

A rede Perceptron, figura 3.7, é a forma mais simples de uma RNA usada para classificação de padrões linearmente separáveis, ou seja, padrões que estão em lados opostos de um hiperplano. Consiste basicamente de um único neurônio com pesos sinápticos ajustáveis e um bias. O comportamento deste tipo de rede é descrito pela equação 3.6:

$$y = f(\sum_{i=1}^{n} w_i \cdot x_i - \theta), \tag{3.6}$$

onde f é uma função de ativação definida a partir do problema que se deseja resolver, que pode ser linear ou não linear.

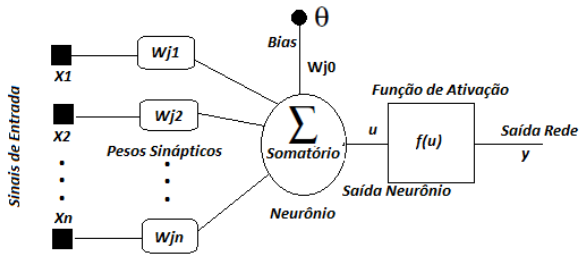

Figura 3.7: Rede perceptron com *n* entradas, um neurônio, bias e função de ativação

3.2 Perceptron de multicamadas

A rede Perceptron de Multicamadas (MLP, do inglês Multilayer Perceptron) é uma generalização do Perceptron, constituída de um conjunto de nós fonte, os quais formam a camada de entrada da rede, uma ou mais camadas ocultas de nós computacionais e uma camada de saída, também de nós computacionais. Com exceção da camada de entrada, todas as outras camadas realizam processamento.

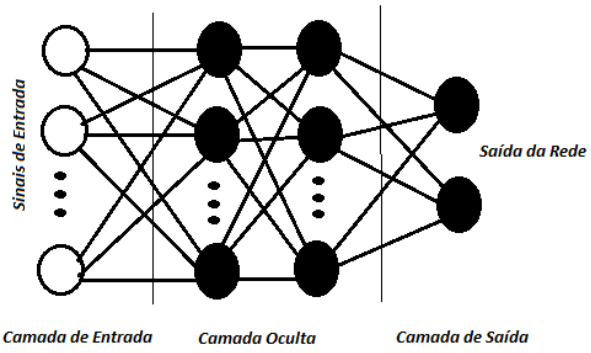

Figura 3.8: Rede MLP, com 1 camada de entrada com *n* neurônios fontes, 2 camadas ocultas com *n* neurônios computacionais, e 1 camada de saída com dois neurônios

Note que, essa é um tipo de rede neural unidirecional distribuída em camadas. Em cada camada é possível ter vários neurônios. As camadas se comunicam camada a camada até atingir a última camada.

Ao projetar uma MLP é necessário considerar dois aspectos importantes, não esquecendo que estes irão determinar o quanto a rede será eficiente para um determinado problema: determinar o número de camadas ocultas e o número de neurônios em cada camada. A determinação destes aspectos geralmente é subjetiva e vai depender da expertise de quem está projetando a rede.

A rede do tipo MLP pode ser classificada em dois grupos: parcialmente conectadas

e totalmente conectadas. Em uma MLP totalmente conectada, cada neurônio de uma camada é conectado a todos os neurônios da camada anterior e a todos os neurônios da camada posterior. Em uma MLP parcialmente conectada, algumas conexões entre neurônios não acontecem.

Entre a camada de entrada e a camada de saída, pode-se ter uma ou mais camadas ocultas. As camadas ocultas proporcionam complexidade e a possibilidade de resolver problemas não linearmente separáveis.

Uma das principais características de uma rede MLP é sua capacidade de resolver problemas não lineares, para isso, é necessário que a função de ativação dos neurônios pertencentes às camadas ocultas seja não-linear. Em geral, a função de ativação é sigmoide.

A MLP é uma rede progressista, ou seja, *feedforward*, essa característica é observada quando as saídas dos neurônios em qualquer particular camada conectam-se unicamente aos neurônios da camada seguinte, em outras palavras são entradas para os neurônios seguintes. Como consequência, a entrada se propaga através da rede, camada a camada, em um sentido progressivo.

De acordo com Simon Haykin, a rede MLP têm sido aplicada com sucesso na solução de diversos problemas difíceis, através do treinamento supervisionado com o algoritmo de retropropagação do erro (do inglês, *backpropagation*).

Basicamente, o treinamento por retropropagação do erro consiste de dois passos, um passo para frente e um passo para trás, respectivamente a propagação da entrada da rede as camadas posteriores e a retropropagação do erro que ocorre em sentido contrário da camada de saída as cadas ocultas. Neste processo é realizado o ajuste nos pesos sinápticos.

Na equação 3.7 e 3.8 a seguir é possível observar o cálculo para os ajustes nos pesos sinápticos do algoritmo *backpropagation*:

$$\rho_j(n) = d_j(n) - y_j(n), \tag{3.7}$$

onde $\rho_j(n)$ é a diferença entre o valor desejado para determinada entrada e saída gerada pela rede,

$$\varepsilon(n) = \frac{1}{2}\sum_{j \in C} \rho_j^2(n), \tag{3.8}$$

onde C é o conjunto de todos os neurônios que pertencem a camada de saída da rede e $\varepsilon(n)$ é o valor instantâneo do erro. A figura 3.9, ilustra o treinamento por retropropagação do erro.

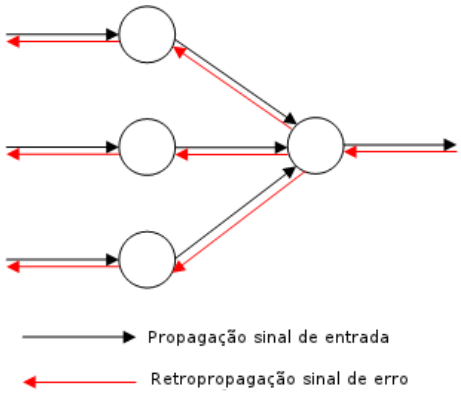

Figura 3.9: Propagação para frente e retropropagação de sinais de erro em uma rede MLP com backpropagation

Basicamente esse processo de aprendizagem consiste em encontrar um conjunto dos pesos W que minimize o custo da função de erro. A eficiência deste processo é observada por medição dos erros apresentados pela rede, realizada com novas entradas ainda não apresentadas a rede. Esse procedimento mede o desempenho de generalização da rede.

O algoritmo *backpropagation* é um paradigma de aprendizagem do tipo supervisionado e seu processo de aprendizado é iterativo, ou seja, a saída produzida pela rede é comparada a uma resposta desejada e melhorada a cada interação. De forma genérica, esse método possui uma função erro (professor) na saída da rede. Após seu cálculo, é realizada a propagação em sentido contrario (retropropagação). Nesse procedimento os pesos sinápticos das camadas ocultas são atualizados.

Por fim, esse processo consiste em ajustar os pesos sinápticos a fim de minimizar o erro entre a saída produzida pela rede em relação a respectiva saída desejada.
Uma vez a rede inicializada, os seguintes passos são realizados em cada iteração:

1. Apresentação dos padrões a rede e cálculo da resposta para cada padrão;
2. O sinal de erro é calculado de acordo com os pesos sinápticos iniciais dado a rede.
3. A minimização do sinal de erro é realizada em direção oposta ao gradiente em relação aos pesos de cada camada. A atualização começa na camada de saída e os erros são propagadas para trás, fornecendo os sinais de erro correspondentes à sinapses da camada anterior, para que ele possa atualizar seus pesos.

Nas duas seções seguintes é apresentada de forma resumida dois tipos de redes progressistas (*feedforward*), e que também são organizadas em camadas.

3.3 Função de bases radial

A rede de função de bases radial (RBF, do inglês Radial Basis Function) é uma rede neural que basicamente consistem de três camadas: uma camada de entrada, uma única camada oculta constituída de função de base radial, que aplica uma transformação não-linear do espaço de entrada para o espaço oculto, normalmente de alta dimensionalidade, uma camada de saída que normalmente é linear.

Essa é utilizada principalmente em problemas para aproximação de funções, pois é um aproximador universal, porém local.

Assim como a rede MLP, a redes RBF é uma rede multicamadas, com neurônios ocultos não-lineares. A principal diferença é que a rede RBF tem apenas uma única camada oculta, cujos neurônios possuem função de ativação gaussiana, em vez de sigmoide.

Esta também é uma rede alimentada adiante (feedforward), ou seja, redes cujas saídas dos neurônios de uma camada é entrada nos neurônios da camada posterior. Na equação 3.9 é possível visualizar a função de ativação do tipo gaussiana:

$$f(u) = e^{-\frac{(u-c)^2}{2\cdot\sigma^2}}. \tag{3.9}$$

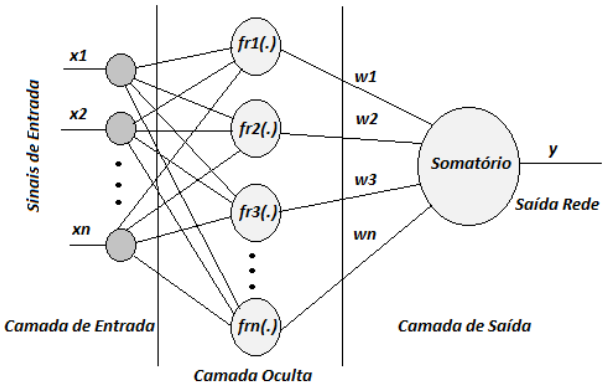

Figura 3.10: Rede RBF com n entradas, 1 camada oculta com n neurônios com funções de base radial e 1 camada de saída com um neurônio.

3.4 Máquina de vetores de suporte

A máquina de vetores de suporte (SVM, do inglês Support Vector Machine) é uma RNA com método de aprendizagem *feedforward* com uma única camada oculta. Esta também é uma rede alimentada adiante, ou seja, redes cujas saídas dos neurônios de uma

camada é entrada nos neurônios da camada posterior, porém não ocorrendo a realimentação.

Este tipo de rede foi inicialmente desenvolvida para classificação binária, que nesse caso, busca a construção de um hiperplano como superfície de decisão, de tal forma que a separação entre exemplos seja máxima. Isso considerando padrões linearmente separáveis. Já no caso de padrões não-linearmente separáveis, a rede busca uma função de mapeamento apropriada para tornar o conjunto mapeado linearmente separável.

Basicamente seu funcionamento pode ser descrito da seguinte forma: dadas duas classes e um conjunto de pontos que pertencem a essas classes, a SVM determinará o hiperplano que separa os pontos de forma a colocar o maior número de pontos da mesma classe em um mesmo lado, enquanto maximiza a distância de cada classe a esse hiperplano.

A SVM foi inicialmente proposta por Vapnik em 1995, com o intuito de resolver problemas de classificação de padrões e é fundamentada na teoria da aprendizagem estatística [Haykin 2001].

O hiperplano gerado pela SVM é determinado por um subconjunto dos pontos das duas classes, chamado vetores de suporte.

A figura 3.11 mostra um exemplo onde uma classe de funções pode ser utilizada para separar padrões linearmente separáveis.

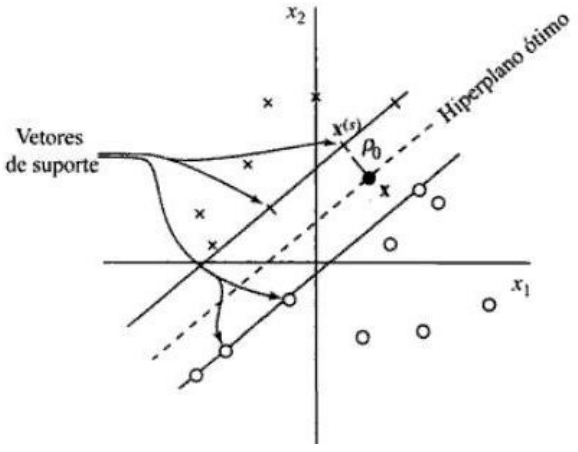

Figura 3.11: Classe de hiperplanos com um hiperplano ótimo [Haykin 2001]

A margem é obtida pela distância entre o hiperplano e os vetores que estão mais próximos a ele, sendo estes os vetores denominados de vetores suporte.

3.5 Considerações sobre o capítulo

Neste capítulo apresentamos quatro tipos de rede: Perceptron, MLP, RBF e SVM, que têm em comum, a propriedade de dividirem-se em camadas, onde o sinal do neurônio da camada anterior é usado como entrada para o neurônio da camada posterior (rede progressista). No entanto, optamos por estudar mais aprofundadamente e implementar a rede do tipo MLP, pois essas são comumente usadas na solução de variados tipos de problema.

Capítulo 4

Computação Paralela

Desde de meados da década de 2000, a disponibilidade no mercado de processadores *multicore* é cada vez mais comum. Entre as principais razões a ser destacada, está a limitação térmica, o que impossibilita o aumento de frequência dos atuais 3 Ghz, portanto, uma das saídas foi aumentar o número de unidade de processamento.

Para a computação, o paralelismo é a capacidade de um sistema ser executado de forma simultânea. Com isso, é possível executar várias instruções ao mesmo tempo.

Fazer com que os atuais programas consigam aproveitar a capacidade total de processamento ainda é um desafio. Em comparação com a programação de programas sequenciais, o desenvolvimento de programas paralelos é mais complexo, principalmente quando existe a necessidade de compartilhar recursos.

De toda forma, a computação paralela é capaz de usar de forma simultânea vários recursos computacionais, de tal forma, que, se bem programado um programa paralelo pode reduzir o tempo necessário para resolver um determinado problema.

Dentre os tipos de computação paralela poderíamos citar:

- Um único computador com múltiplos processadores;
- Um número arbitrário de computadores ligados em uma rede; ou ainda
- A combinação de ambos.

Os sistemas paralelos podem ser classificados, por exemplo, pelo fluxo de instruções e pelo fluxo de dados. Uma das metodologias mais conhecidas e utilizadas para classificar a arquitetura de um computador ou conjunto de computadores é a taxonomia de Flynn (1966) [Tanenbaum 2007]. Essa taxonomia classifica os computadores em quatro categorias:

SISD (do inglês, Single Instruction Single Data) - As instruções são executadas sequencialmente, mas podem ser sobrepostas nos seus estágios de execução (pipeline).

SIMD (do inglês, Single Instruction Multiple Data) - As instruções são executadas

em uma única instrução, através de uma unidade de controle, executa de forma síncrona em um conjunto de dados diferentes, distribuídos ao longo de processadores elementares.

MISD (do inglês, Multiple Instruction Single Data) - Diferentes instruções operam na mesma posição de memória ao mesmo tempo, executando instruções diferentes.

MIMD (do inglês, Multiple Instruction Multiple Data) - Cada processador executa seu próprio programa sobre seus próprios dados, de forma assíncrona.

Os últimos avanços na tecnologia de desenvolvimento de processadores, fazem com que, seja necessário mudar o padrão para o desenvolvimento de *software*. O que até pouco tempo resolvia-se de forma sequencial, precisa ser agora resolvido de forma paralela. O *software* precisa resolver seus problemas concorrentemente, executando vários cálculos simultâneos, de tal forma que, ao adicionar novos núcleos, o *software* seja executado mais rapidamente e de modo mais eficiente.

A computação paralela divide-se em três grupos: simétrica, distribuída, ou a combinação de ambas.

Na computação paralela simétrica, os computadores são multiprocessados: possuem múltiplos elementos de processamento em somente uma máquina. O acesso a memória neste tipo é simétrico (SMP, do inglês Symmetric Multiprocessing). Nessa arquitetura, várias CPUs e memórias estão ligadas por um barramento compartilhado ou ainda por uma interconexão rápida. Veja um exemplo na figura 4.1.

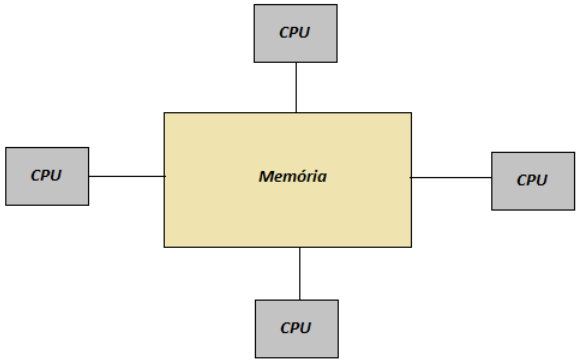

Figura 4.1: Modelo computacional simétrico com 4 processadores compartilhando o acesso a memória de forma simétrica

Já na computação paralela distribuída, os elementos de processamento estão distribuídos em computadores ligados por um sistema de interconexão, e em geral o acesso a memória é assimétrico, porém pode ser simétrico. Para maiores detalhes consultar Calvin Lin ou Andrew Tanenbaum [Lin e Snyder 2009, Tanenbaum 2007].

Na figura 4.2, é possível visualizar um exemplo do modelo computacional distribuído.

Já na figura 4.3 uma demostração híbrida, que faz uso dos modelos simétricos e assimétricos.

Figura 4.2: Modelo computacional distribuído com 4 computadores ligados por uma interconexão, cada um com sua própria memória

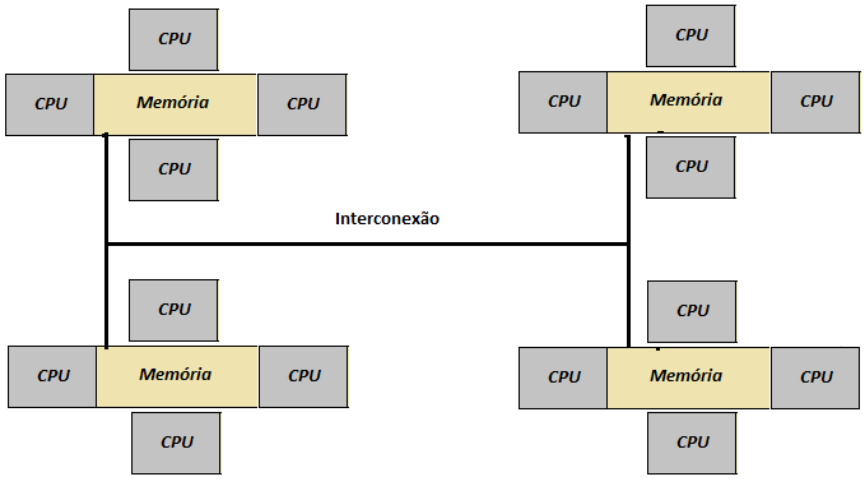

Figura 4.3: Modelo computacional híbrido com 4 computadores ligados por uma interconexão de rede, cada um com sua própria memória com acesso simétrico

Com a definição desses modelos criam-se dois paradigmas de comunicação entre os processadores: memória compartilhada e por troca de mensagens. Para o paradigma memória compartilhada, é necessário usar algum tipo de controle, esse terá como função evitar o acesso concorrente a uma região da memória por mais de uma unidade de processamento.

Geralmente uma comunicação exige uma sincronização. Entre as formas de sincronização destacam-se: exclusão mútua e sincronização condicional. Não vamos aqui estender tal explicação, porém, exclusão mútua tem como objetivo sincronizar processos de forma a garantir que determinadas partes do código sejam executadas por no máximo uma única unidade de processamento por vez, evitando que recursos compartilhados sejam acessados e manipulados simultaneamente por vários processos.

Entre os exemplos de controle para o processo de comunicação e sincronização, podemos destacar: o semáforo, o mutex, os monitores e os ferrolhos que são definidos por variáveis comuns, que tem como função servir de bloqueio a determinada variável. Para maiores detalhes consulte Andrew Tanenbaum [Tanenbaum 2007].

No entanto, um caso a destacar é o do mutex, que realiza uma operação atômico de test-and-set, permitindo que um processo consulte outro e faça o teste se este está disponível e em caso afirmativo mude o status para bloqueado em uma única operação.

Já para o paradigma troca de mensagem, geralmente é utilizada uma interconexão por uma rede de computadores, que possibilite a comunicação entre as unidades de processamento. Nesse paradigma, geralmente cada processador tem sua própria memória acessada apenas por si mesmo.

Apesar das arquiteturas paralelas serem atualmente uma realidade, a programação de programas paralelos continua a ser uma tarefa complexa. Diante dos novos desafios para o desenvolvimento de programas paralelos, que não estão presentes no desenvolvimento de programas sequenciais, destacam-se:

- Concorrência;
- Comunicação;
- Sincronização;
- Balanceamento de carga.

A programação de programas paralelos é mais difícil. Em geral, nem sempre um computador com *N* processadores trabalhando paralelamente atinge seu desempenho máximo. Além dos desafios citados anteriormente, é necessário desenvolver algoritmos eficientes sem desconsiderar a complexidade imposta pela era multicore [Lin e Snyder 2009]. Diante desse contexto, a simples paralelização (codificação/compilação) do código não é suficiente, sendo necessário desenvolver novos algoritmos, novas estratégias e meios matemáticos para conseguir melhores desempenhos.

Em compensação, dentre os principais motivos para utilizar programas paralelos, destacam-se:

- Reduzir o tempo necessário para solucionar um problema;
- Resolver problemas com maior complexidade e de maior dimensão; e
- Obter mais respostas no mesmo tempo.

Dentre as ferramentas disponíveis para a codificação de algoritmos paralelos, poderíamos citar duas, OpenMP (do inglês Open Multi-Processing) e MPI (do ingles Message Passing Interface).

O OpenMP é um modelo portável e otimizado que dá ao programador uma interface simples e flexível para o desenvolvimento de suas aplicações. Nas seções que se seguem maiores detalhes.

Já o MPI, é um modelo baseado no conceito de envio e recebimento de mensagens entre os processos ou entre computadores, de tal forma que, um processo em execução se comunique com outros. Essa ferramenta é usada para o desenvolvimento de programas com memória distribuída, máquinas paralelas, e em "clusters".

O MPI tornou-se um padrão para o desenvolvimento de aplicações paralelas baseado no paradigma de passagem de mensagens. A sua utilização permite a geração de um código, flexível portátil e otimizado para todas as arquiteturas paralelas.

Atualmente o MPI encontra-se na versão 2.2 e é desenvolvido em conjunto, por fabricantes de software, universidades, usuários e cientistas. Entre os objetivos dos desenvolvedores está, estabelecer um padrão portátil, eficiente e flexível. Maiores detalhes em [MPI documents 2012].

4.1 Avaliação de desempenho e eficiência paralela

A avaliação de um sistema paralelo é dada principalmente com a observação do *speedup* e da eficiência paralela.

O *speedup* é uma mediada que se refere a quanto um algoritmo paralelo é mais rápido do que um algoritmo sequencial correspondente [Lin e Snyder 2009]. Em outras palavras, o *speedup* é uma medida que se refere ao ganho em relação ao tempo entre um processo executado com um único processador, e um executado com N processadores.

Para calcular o *speedup* é usada a equação 4.1:

$$S_p = \frac{T_s}{T_p}, \tag{4.1}$$

onde p é o numero de processadores, T_s é o tempo de execução serial e T_p é o tempo de execução paralela com p processadores.
Os resultados possíveis para o *speedup*(S) são:
S < 1, *slowdown*, situação indesejável;
S < p, *sublinear*, comportamento comum;
S = p, *linear*, ideal e onde não existe sobrecarga;
S > p, *superlinear*, situação possível.

O *speedup* linear ocorre raramente pois a maioria das soluções paralelas introduzem alguma sobrecarga.

A eficiência é uma medida de quanto o processamento está realmente sendo usado para fazer cálculos e de quanto é gasto com comunicação e sincronismo [Lin e Snyder 2009].

Para calcular a eficiência paralela é usada a equação 4.2:

$$E = \frac{T_s}{p \cdot T_p}, \tag{4.2}$$

onde p é o número de processadores da versão paralela e T_s e T_p são os tempos serial e paralelo, respectivamente. Observe que, se tivermos $T_p = T_s/p$ para p processadores, teríamos todos os processadores fazendo algum trabalho útil, o tempo gasto com comunicação seria nulo e a eficiência seria 1, ou seja 100%. Os resultados mais comuns estão entre 0 e 1, sendo 1 o caso de maior eficiência.

Não podendo desconsiderar a lei de Amdahl's, que refere-se a existência em programas paralelos de uma fração do algoritmo que obrigatoriamente será sequencial. Na fração sequencial, o *speedup* é nulo ou inexistente, em quanto que a fração paralela será distribuída entre as diversas unidades de processamento.

Para o calculo da lei de Amdahl's é usada a equação 4.3:

$$T_p = \frac{1}{S} \cdot T_s + (1 - \frac{1}{S}) \cdot \frac{T_s}{P}, \tag{4.3}$$

para maiores detalhes consultar Calvin Lin [Lin e Snyder 2009].

A escalabilidade é uma característica desejável em todos os programas. Na computação paralela, ela indica a capacidade de um programa paralelo em manter uniforme sua eficiência E ao aumentar o número de unidades de processamento e o tamanho do problema.

Também é possível observar a escalabilidade em um programa paralelo se ao aumentar o tamanho do problema a eficiência melhorar para o mesmo número de processadores.

4.2 Codificação com OpenMP

O OpenMP é um conjunto de bibliotecas ou uma API para processamento paralelo baseado em memória compartilhada, com suporte a multi-plataforma, programação

em C, C++ e Fortran para vários ambientes, incluindo Unix (linux) e Microsoft Windows [OpenMP.org 2012].

A sua especificação foi criada por um grupo de grandes fabricantes de *hardware* e *software* com o objetivo de desenvolver uma API que seja portável e escalável. Entre as empresas que criaram e mantém o OpenMP estão inclusas a Intel, HP, SGI, IBM, Sun, Compaq, KAI, PGI, PSR, APR, Absoft, ANSYS, Fluent, Oxford Molecular, NAG, DOE, ASCI, Dash e Livermore Software.

A programação paralela com OpenMP foi projetada para computadores paralelos com memória compartilhada, para dois tipos de arquitetura, memória compartilhada centralizada e memória compartilhada distribuída, além de incorpora o conceito de *threads* [OpenMP.org 2012].

Uma *thread*, ou em português, linha de execução é uma pequena fatia de um processo, esta pode compartilhar o mesmo espaço de memória com outras *threads*. O seu uso permite dividir um processo em tarefas que podem ser executadas paralelamente. A comunicação é através de uma área compartilhada e é coordenada pela *thread* master. A *thread* permite, por exemplo, que o usuário de um programa utilize uma funcionalidade do ambiente enquanto outras *threads* realizam outros cálculos e operações.

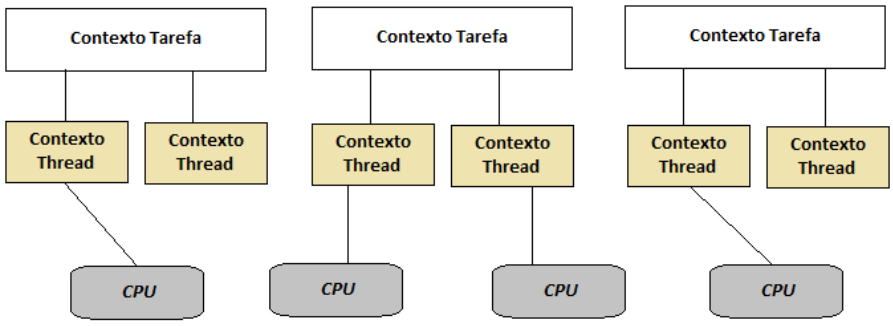

Figura 4.4: A figura mostra a relação entre os processadores, threads e tarefas

Cada thread tem o mesmo contexto de software e compartilha o mesmo espaço de memória (endereçado ao processo master), porém o contexto de hardware é diferente. Sendo assim o overhead causado pelo escalonamento de uma thread é muito menor do que o escalonamento de um processo [Abraham 2010].

O suporte a threads é fornecido pelo próprio sistema operacional (SO) e essas são classificadas em dois níveis. As threads ao nível do núcleo (KLT, do inglês, Kernel-Level Thread) são criadas e gerenciadas pelo próprio SO. Já as threads a nível de usuário (ULT, do inglês, User-Level Thread) são implementada (criadas e gerenciadas) através de uma biblioteca de uma determinada linguagem [Abraham 2010].

As threads da categoria (ULT) são implementadas pela aplicação, sem conhecimento do sistema operacional e geralmente são adicionadas por pacotes de rotinas (códigos para criar, terminar, escalonamento e armazenar contexto) fornecidas por uma determinada biblioteca de uma linguagem.

As ULT suportam as mesmas operações que as KLT (criar, sincronizar, duplicar e abortar). Essas são escalonadas pelo programador, tendo a grande vantagem de cada processo usar um algoritmo de escalonamento que melhor se adapte a situação, o sistema operacional neste tipo de thread não faz o escalonamento, em geral ele não sabe que elas existem.

As KLT são escalonadas diretamente pelo sistema operacional, comumente são mais lentas que as threads ULT pois a cada chamada elas necessitam consultar o sistema, exigindo assim a mudança total de contexto do processador, memória e outros níveis necessários para alternar um processo [Abraham 2010].

Em hardwares equipados com um único processador/core, cada *thread* é processada de forma aparentemente simultânea, pois a mudança entre uma thread e outra é feita de forma tão rápida que para o usuário isso está acontecendo paralelamente. Em hardwares com múltiplos processadores ou multi-cores, as *threads* são executadas pelos processadores realmente de forma simultânea;

Um dos benefícios do uso das *threads* que pode ser citado é o fato do processo poder ser dividido em mais de uma tarefa, por exemplo, quando uma *thread* está esperando determinado dispositivo de entrada e saída (I/O, do inglês Input/Output) ou qualquer outro recurso do sistema, o processo como um todo não fica parado, pois quando uma *thread* entra no estado de bloqueio uma outra *thread* aguarda na fila de prontos para executar.

Basicamente uma thread pode assumir os seguintes estados:

- **Criação**, neste estado, o processo pai está criando a thread que é levada a fila de prontos;
- **Execução**, este estado a thread está usando o processador;
- **Pronto**, neste estado a thread avisa ao processador que esta pronta para entrar no estado de execução e entra na fila de prontos;
- **Bloqueado**, neste estado, por algum motivo, o processador bloqueia a thread, geralmente enquanto aguarda algum dispositivo de I/O;e
- **Término**, neste estado, são desativados o contexto de hardware e a pilha é desalocada.

As regiões paralelas em um programa codificado com o OpenMP são definidas com uma linha de código especial chamadas de diretivas, por exemplo, em C/C++ "`#pragma`

`omp`".

Os programas começam a ser executados com uma única *thread*, chamada master, quando encontram no código a primeira região paralela cria outras *threads* em um modelo *fork/Join*, ou seja, dividir em uma execução paralela e depois juntar ou regredir a execução em um único processo.

As *threads* executam a parte do programa que está dentro da região paralela e no fim a thread master recebe a conclusão das *threads* filhas para continuar a execução do programa até encontrar outra sentença ou diretiva paralela.

Dentro da região paralela os dados podem ser privados ou compartilhados. Nos dados definidos como compartilhados, o acesso as variáveis é comum para todas as *threads*, ou seja, todas as *threads* podem ler e escrever. Já nos dados privados cada *thread* tem sua própria cópia dos dados, ou seja, são inacessíveis às outras *threads*.

Em muitos programas os laços são as principais partes paralelizáveis. Se as interações deste laços forem independentes, sua execução acontece em qualquer ordem, pois não existe dependência entre os dados.

Já nos casos em que haja dependência entre os dados, é necessário garantir que quando a *thread* 2 for pegar os dados da thread 1, a thread 1 já tenha concluído suas operações.

No OpenMP, por padrão, as variáveis são definidas como compartilhadas [OpenMP.org 2012].

Um exemplo de uso do OpenMP dentro do código em **C/C++** pode ser visto a seguir:

```
#pragma omp parallel
{
bloco de código (região paralela)
}
```

Dentro do OpenMP, existem funções úteis como por exemplo para saber em qual thread está a execução atual, "`# int omp_get_thread_num(void)`", ou ainda, para saber a quantidade de threads em execução, "`# int omp_get_num_threads(void)`".

As diretivas do OpenMP permitem o uso das chamadas cláusulas, que especificam informações adicionais. Por exemplo: "`# pragma omp parallel [cláusula(s)]`".

As cláusulas mais comuns são: "`shared(lista de variável compartilhadas)`" e "private(lita de variáveis privadas)", os dados são definidos respectivamente como compartilhados e privados para uma determinada região paralela.

Além dessas, existem diversas outras cláusulas como a "`firstprivate(variável)`", usada em variáveis do tipo compartilhada mas de uso privado. Um exemplo é uma variável usada como índice de um `for`. Outra cláusura muito utilizada em operações de adição, subtração e multiplicação é a "`reduction(op:list)`", após as operações reali-

zadas pelas threads filhas a thread master realiza uma operação de redução para então ter o resultado da operação.

Em resumo, o OpenMP é uma API para máquinas multi-threads com memória compartilhada. Seu uso torna explícito o paralelismo existente nos programas, projetado para ser escalável e portável. Também adiciona uma camada de abstração acima do nível das threads, deixando o programador livre para a tarefa de criar, gerenciar e destruir threads [OpenMP.org 2012]. Para maiores detalhes consultar o **apêndice A**.

4.3 Considerações sobre o capítulo

Por mais que esteja disponível uma grande quantidade de unidades de processamento, ou ainda, por mais poderoso o sistema paralelo, é necessário que o programador desenvolva um programa capaz de aproveitar corretamente os recursos disponíveis. Caso contrário, não conseguirá tirar corretamente proveitos da era multicore. A consequência de uma solução incorreta será um pior desempenho, ou ainda, uma situação em que, a velocidade de execução máxima será limitada, sendo praticamente irrelevante aumentar o número de processadores.

Capítulo 5

Desenvolvimento

A paralelização da rede MLP apresentada neste trabalho incorpora um novo algoritmo que explora de forma inovadora os novos recursos disponibilizados pela era multicore. A MLP proposta é baseada na rede *multilayer perceptron* tradicional. No entanto, não é meramente uma adaptação do algoritmo sequencial. Além disso, essa proposta usa algumas heurísticas para melhorar a eficiência e escalabilidade paralela.

A implementação e os resultados desse algoritmo levam em consideração o passo à frente desse tipo de rede, ou seja, a execução de uma entrada até a resposta da rede. Esse procedimento foi usado nas implementações da MLP tradicional e da MLP proposta.

A codificação do algoritmo foi realizada com uma biblioteca do OpenMP em linguagem "C"e as observações e resultados foram feitas em um sistema operacional linux instalado em um servidor com 24 cores AMD. Esse ambiente computacional é do tipo multiprocessamento simétrico ou SMP.

De forma simulada, foi realizada uma analise do *speedup* e eficiência paralela para a arquitetura computacional com acesso assimétrico à memória (memória distribuída).

Os testes foram realizados variando o número de neurônios, de módulos e de conexões remotas, todos eles com duas camadas ocultas e um único nerônio na camada de saída.

Os resultados demostram que a MLP proposta é mais adequada para as implementações paralelas, principalmente para um número menor de conexões entre os neurônios remotos, reduzindo a sobrecarga de comunicação e sincronismo.

5.1 Implementação Paralela

A rede MLP Modular, proposta neste trabalho, é composta de módulos formados por camadas e neurônios organizados de tal forma que, dentro dos módulos os neurônios são totalmente conectados. Já as conexões entre os neurônios que são remotos, ou seja, estão em outros módulos, é limitada, fazendo com que a comunicação entre os módulos seja

menor.

O número de conexões (sinapses) que conectam os neurônios remotos influencia no desempenho do sistema, ou seja, quanto maior for esse número, maior será a comunicação entre os módulos, o que pode provocar uma maior sobrecarga de comunicação, que não é desejado para as implementações paralelas.

A MLP clássica é definida pela expressão 5.1:

$$h_i^l = f^l(b_i^l + \sum_{j=1}^{n^{l-1}} w_{i,j}^l h_j^{l-1}), \tag{5.1}$$

onde f^l é a função de ativação da camada l-ésima, b_i^l é o bias para o neurônio i-ésima na camada l-ésima, para $i = 1,2,\cdots,n^l$. O valor $n^l - 1$ é o número de neurônios na camada $l-1$ e $w_{i,j}^l$ é o peso sináptico entre os neurônios de cada camada subsequente. Os valores $h_i^1 = u_i$ e $\mathrm{h}_j^L = \mathrm{y}_j$ para i = 1,2,...,I e j = 1,2,...,O, definem a entrada e saída da rede, com I e O, sendo o tamanho da entrada e saída, respectivamente. O valor de L, representa o número de camadas.

A MLP modular foi definida com a expressão 5.2 originada da MLP clássica:

$$h_i^l = f^l(b_i^l + \sum_{j \in \Lambda_m^l} w_{i,j}^l h_j^{l-1} + \sum_{k \in \Gamma_m^l} w_{i,k}^l h_k^{l-1}),\ \forall\ i \in \Lambda_m^l, \tag{5.2}$$

onde h_i^l, é a saída do neurônio na camada l, com modulo m em $i \in \Lambda_m^l$. O conjunto Λ_m^l, é um subconjunto de todos os índices dos neurônios h^l, que são atribuídos ao módulo m, na camada l, ou seja, o módulo local. A união de todos Λ^l é equivalente a todos os índices h^l:

$$\bigcup_{m=1}^{M} \Lambda_m^l = \{1,2,\cdots,n_l\}, \tag{5.3}$$

onde M é o número total de módulos. O subconjunto Γ_m^l contêm os índices dos neurônios que não estão conectados localmente ao módulo m, ou seja, os neurônios dos módulos remotos.

Para uma rede MLP totalmente conectada, a união do conjunto Γ_m^l com o conjunto Λ_m^l é o que equivale ao conjunto de todos os índices de h^l, ou seja, todos os neurônios conectados.

$$\Gamma_m^l \cup \Lambda_m^l = \{1,\cdots,n^l\},\ l > 1,\ \forall\ m = 1,2,\cdots,M. \tag{5.4}$$

O valor de l em 5.4 deve ser maior que 1, pois a entrada é compartilhada entre todos

os módulos, sem prejuízo para o desempenho em paralelo, consequentemente, a segunda camada não têm ligações remotas a partir da camada de entrada.

Para uma rede totalmente conectada, temos:

$$\left|\Gamma_m^l\right| \leq n^l - \left|\Lambda_m^l\right|. \tag{5.5}$$

Uma ilustração de uma MLP modular com 2 módulos é apresentado na Figura 5.1. Neste exemplo, os conjuntos de índices usados para calcular a terceira camada h_1^3 são $\Lambda_1^2 = \{1,2\}$ e $\Gamma_1^2 = \{3\}$, para h_2^3 são $\Lambda_1^2 = \{1,2\}$ e $\Gamma_1^2 = \emptyset$, para h_3^3 são $\Lambda_2^2 = \{3,4\}$ e $\Gamma_2^2 = \emptyset$, e para h_4^3 são $\Lambda_2^2 = \{3,4\}$ e $\Gamma_2^2 = \{1\}$. Para os conjuntos da camada de saída, ou camada 4, $y_1 \equiv h_1^4$ são $\Lambda_1^3 = \{1,2\}$ e $\Gamma_1^3 = \{4\}$, e para $y_2 \equiv h_2^4$ são $\Lambda_2^3 = \{3,4\}$ e $\Gamma_2^3 = \{2\}$.

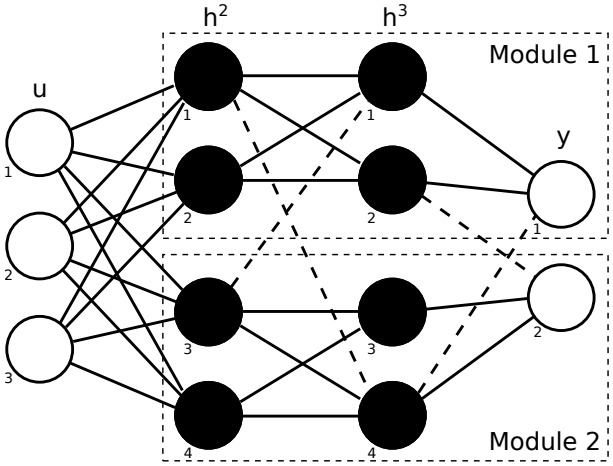

Figura 5.1: MLP Proposta com 2 camadas ocultas, 1 camada de saída com 2 neurônios, executada por 2 módulos.

5.2 Implementação com OpenMP

Para um maior entendimento, observe que, para a implementação com OpenMP, os módulos da MLP Modular são definidos por threads. A *thread* permite a divisão de um processo em diversas tarefas. Nesse contexto, a rede divide-se em várias outras tarefas que trabalham concorrentemente, de tal forma que, cada módulo da rede fica associado a uma *thread*.

Para possibilitar um compartilhamento correto de dados, a comunicação entre as *threads* é controlada por uma *"flag"*. Em cada neurônio existe uma sinalização tipo *"flag"* que o habilita para ser usado nos neurônios de outras threads, ou seja, se já está pronto para ser uma entrada em um neurônio da próxima camada em um outro módulo.

Cada módulo realiza a seguinte sequência de forma independente e em paralelo:

1) Calcula as entradas locais diretamente conectada aos neurônios;
2) Modifica a *flag* para "liberada"após a conclusão em cada neurônio local;
3) Nos neurônios locais são realizadas as seguintes sequências:
   - 3.1) Calcule os neurônios conectados localmente;
   - 3.2) Verifica se há conclusão de neurônios conectados remotamente; caso positivo
   - 3.3) calcule os neurônios com as conexões remotas.
4) Calcula a função de ativação;
5) Acione a *flag* após a conclusão dos neurônios locais.

Algoritmo:

for all $i \in \Lambda_m^l$ **do**
  Calcule h_i^2;
  Modifique a *flag* do neurônio h_i^2 para liberada;
end for
for all $l < L$ **do**
  for all $i \in \Lambda_m^l$ **do**
    Calcule o b_i^l bias para o h_i^l;
    for all $j \in \Lambda_m^l$ **do**
      Calcule $w_{i,j}^l h_j^{l-1}$ para o peso sináptico de h_i^l;
    end for
    for all $k \in \Gamma_m^l$ **do**
      Aguarde até que a *flag* do neurônio h_k^{l-1} esteja liberada;
      Calcule $w_{i,k}^l h_k^{l-1}$ para o peso sináptico de h_i^l;
    end for
    Calcule a função de ativação f^l para o neurônio h_i^l;
    Modifique a *flag* do neurônio h_i^l;
  end for
end for

5.2.1 Resultados

Os resultados mostram que a implementação da MLP Modular assim como esperado obteve resultados melhores que a simples paralelização da MLP clássica. Em todos os casos a métrica utilizada é o tempo de resposta. A partir desse tempo foi calculado o *speedup* e a eficiência paralela para cada experimento.

Os experimentos foram feitos variando o número total de neurônios e de conexões remotas por módulos. Também foram analisadas redes parcialmente e totalmente conectadas.

Os testes foram realizados com a seguinte metodologia:

- fixação de arquitetura da rede neural;
- distribuição dos neurônios e camadas por módulos; e
- variação do número de conexões remotas.

As variáveis que foram consideras em nossos experimentos foram:

M o número de módulos;
W o número total de sinapses;
R o número de conexões remotas por módulo.

Para que os experimentos fossem computacionalmente justos, consideramos em todos os experimentos um número equivalente de neurônios para a rede proposta a da rede MLP clássica. A seguinte fórmula 5.6 foi utilizada para calcular o número de neurônios por módulo na rede proposta, de modo que o seu número de neurônios fosse aproximadamente equivalente ao da MLP clássica.

$$k = \frac{(i+1+l) + \sqrt{(i+1+l)^2 - 4 \cdot (l \cdot m) \cdot (\frac{-wf}{m})}}{2 \cdot (l \cdot m)}, \tag{5.6}$$

onde k é número desejado de neurônios por módulo; i é o valor de entrada; l é o número de camadas; m o número de módulos; e wf é o número total de sinapses.

MLP Modular com um número equivalente de sinapses de uma rede MLP clássica totalmente conectada

Para realizar os testes de desempenho de uma rede modular com equivalência a uma rede clássica totalmente conectada foi necessário calcular o número de neurônios de acordo com o número de módulos M, o que proporcionou um número de conexões sinápticas equivalente.

Os resultados demostram que as melhores avaliações paralelas foram para um número maior de W e de M, ou seja, para rede de grandes dimensões ou de maior complexidade. Ver figura 5.2.

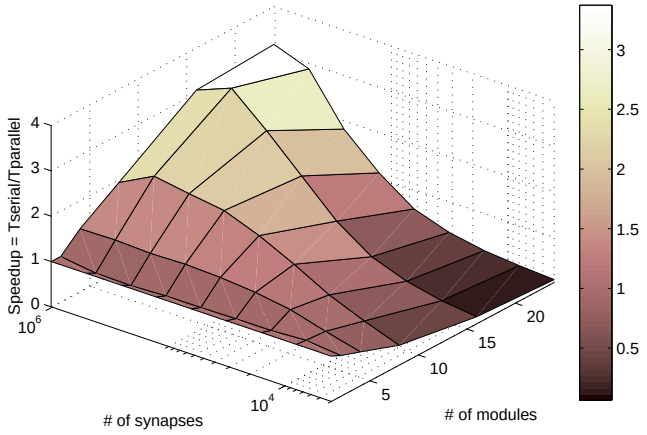

Figura 5.2: Speedup para MLP modular totalmente conectada, com variação do W e M.

Com o intuito de observar a diferença entre uma arquitetura computacional de memória compartilhada e uma de memória distribuída, foi introduzido um atraso de 100 NOP de instrução para acesso aos neurônios remotos. O NOP é uma instrução que não executa nenhuma operação. Será útil apenas para inserir atrasos em ciclos de *clock*. O resultado pode ser visto na figura 5.3.

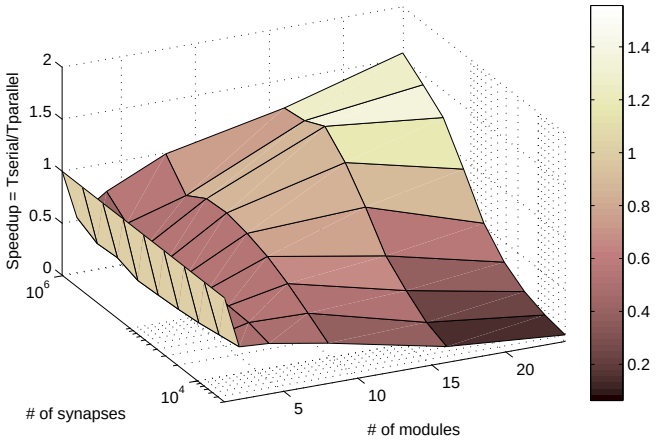

Figura 5.3: Speedup para MLP modular totalmente conectada, com variação do número W e de M, com um atraso de 100 NOP de instrução

Na figura 5.4 é possível observar os experimentos realizados com latência de 100

NOP, com speedup de 1,5, já sem a latência o speedup é aproximadamente de 3,5.

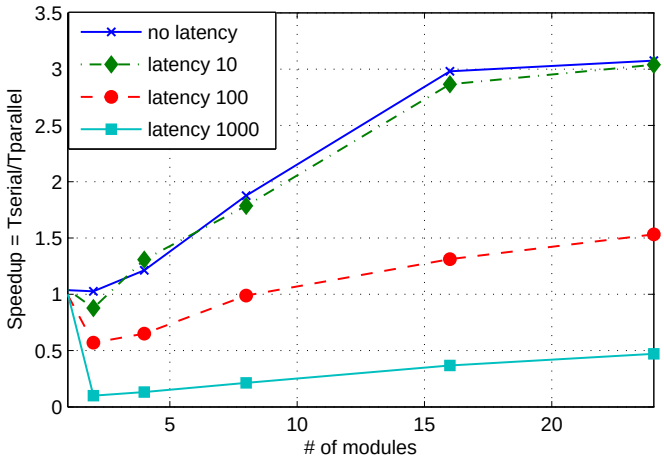

Figura 5.4: Adição de diferentes NOP para uma rede com aproximadamente 10^6 conexões no geral.

MLP Modular com um controlado número de conexões remotas

As análises foram realizadas com variação do número de conexões remotas entre os módulos, porém, mantendo o número de neurônios equivalente a uma rede totalmente conectada. Também foi considerado um atraso de 100 NOP para acessar os neurônios remotos.

Na figura 5.5 é possível visualizar os resultados com variação do número de conexões remotas para uma rede com cerca de 10^6 conexões. A velocidade de execução da rede já não ocorre juntamente com o número de processadores, é possível concluir que um número menor de conexões remotas proporciona um melhor desempenho e torna a aplicação escalável.

Na figura 5.6 é possível observar que para a rede totalmente conectada a eficiência E após certo número de conexões tende a estabilizar. Já com a rede proposta, observa-se que, a eficiência continua crescendo, o que leva a concluir que esse tipo de rede é mais escalável e mais adequada a era multicore.

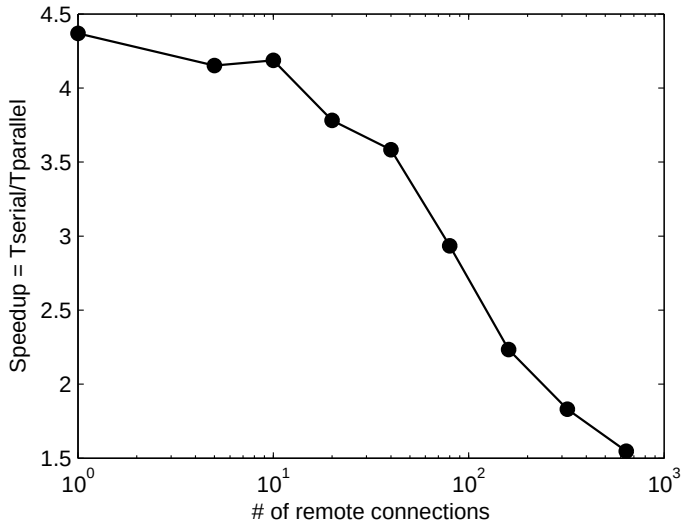

Figura 5.5: Speedup para variados números de conexões remotas, aproximadamente 10^6.

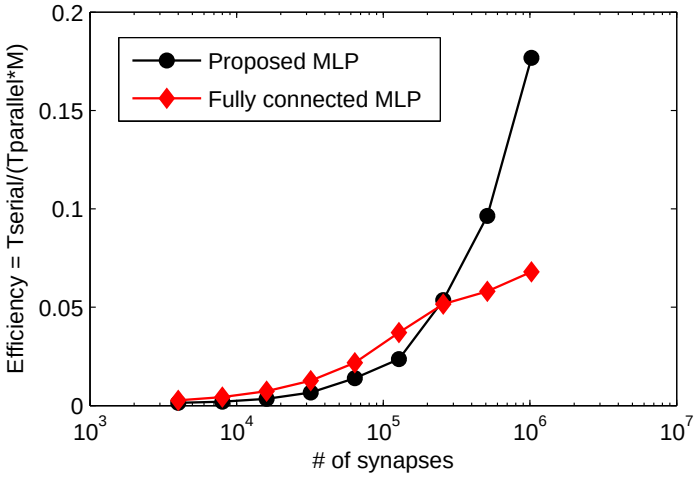

Figura 5.6: Eficiência, MLP Modular proposta vs. MLP totalmente conectada

5.3 Considerações sobre o capítulo

Os resultados demostram que os melhores *speedups* são observados para redes de médio e de grande porte. No entanto, foi observado para a implementação proposta uma melhor escalabilidade quando comparada com a MLP totalmente conectada. Nos casos em que a rede é de menor porte, a excessiva sobrecarga de comunicação foi a causa para não ser observado melhores resultados ou mesmo piores desempenhos.

Capítulo 6

Conclusões

Neste trabalho, apresentamos um estudo e uma implementação inicial de uma rede MLP Modular. Esta proposta, como mostrada os resultados, melhorou a velocidade de execução da rede MLP. Se for levado em consideração a execução, poderíamos concluir que nossa proposta foi mais rápida, principalmente para um número menor de conexões remotas, além de escalável para arquitetura paralela analisada.

Nos resultados apresentados é possível ainda perceber que para um número pequeno de conexões remotas a proposta é mais adaptada a era *multicore* se comparada a simples paralelização da MLP clássica. Percebe-se que a MLP Modular mostrou-se mais adequada, principalmente, se observado seu potencial de escalabilidade.

Também é possível concluir que, a rede MLP Modular com um número grande de conexões remotas é fracamente escalável para a arquitetura paralela com acesso distribuído à memória. Por outro lado, a proposta modular apresentou-se bastante escalável, especialmente para o número pequeno de conexões remotas e para arquitetura paralela com acesso simétrico a memória.

De toda forma, é possível concluir que os resultados demonstraram uma redução no tempo total de resposta da rede. No entanto, a relação entre o tempo de resposta da rede e o número de neurônios remotos é inversamente proporcional, pois ainda é inevitável o *overhead* de comunicação e a sincronização entre as unidades de processamento.

Os resultados iniciais foram positivos para a grande maioria dos experimentos, porém, em um outro trabalho é necessário avaliar a eficiência funcional da rede. Experimentos limitados indicam que a rede com algumas conexões remotas, mas, com um número semelhante de sinapses, tem uma eficiência funcional equivalente a uma rede MLP clássica totalmente conectada.

Considerando trabalhos futuros e não tomando a proposta como absoluta e completa, espera-se que este trabalho estimule a discussão sobre o tema, bem como, maiores discussões e melhorias ao modelo proposto.

Referências Bibliográficas

Abraham, Silberschatz (2010), *Fundamentos de Sistemas Operacionais*, 8 edi, Ltc.

da Silva, Ivan Nunes, Danilo Hernane Spatti e Rogério Andrade Flauzino (2010), *Rede Neurais Artificiais: Para Engenharia e Ciências Aplicadas*, Ed. Artliber, São Paulo.

de Llano, Rafael Menéndez e José Luis Bosque (2010), 'Study of neural net training methods in parallel and distributed architectures', *Future Generation Computer Systems* **26**, 267–275.

de S. Alves, Robinson L., Jorge D. de Melo, Adrião D. Doria Neto e Ana Claudia M. L. Albuquerque (2002), New parallel algorithms for back-propagation learning, *em* 'World Congress on Computacional Intelligence (WCCI 2002)'.

Haykin, Simon (2001), *Redes Neurais: Princípios e Práticas*, 2 edi, BOOKMAN.

Hunter, David e Bogdan Wilamowski (2011), Parallel multi-layer neural network architecture with improved efficiency, *em* 'International Conference on Human System Interaction (HSI 2011)', Yokohama, Japan.

Lin, Calvin e Larry Snyder (2009), *Principles of Parallel Programming*, Addison-Wesley - 02/26/2008.

Lopes, Danniel C., Rafael M. Magalhães, Jorge D. Melo e Adrião D. Dória Neto (2009), Implementation and evaluation of modular neural networks in a multiple processor system on chip to classify electric disturbance, *em* 'International Conference on Computational Science and Engineering'.

MPI documents, janeiro de 2012 (2012), Mpi documents, janeiro de 2012, Mpi documents.
URL: *http://www.mpi-forum.org/docs/*

OpenMP.org, janeiro de 2012 (2012), Openmp.org, janeiro de 2012, Openmo.org.
URL: *http://www.openmp.org/*

Seiffert, Udo (2002), Artificial neural networks on massively parallel computer hardware, *em* 'ESANN 2002 proceedings - European Symposium on Artificial Neural Networks', d-side publi., Bruges (Belgium), pp. 319–330.

Strey, Alfred (2004), 'A comparison of openmp and mpi for neural network simulations on a sunfire 6800', *Parallel Computing: Software Technology, Algorithms, Architectures Applications, G.R. Joubert et al. (Editors), Elsevier* .

Tanenbaum, Andrew S. (2007), *Organização Estruturada de Computadores*, 5 edi, Person Prentice Hall Brasil.

Turchenko, Volodymyr e Lucio Grandinetti (2009*a*), Efficiency research of batch and single pattern mlp parallel training algorithms, *em* J. C.A. Loureiro, ed., 'IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications', Rende (Cosenza), Italy.

Turchenko, Volodymyr e Lucio Grandinetti (2009*b*), Investigation of computational cost model of mlp parallel batch training algorithm, *em* 'IEEE Symposium on Industrial Electronics and Applications (ISIEA 2009)', Kuala Lumpur, Malaysia.

Turchenko, Volodymyr, Lucio Grandinettia, George Bosilcab e Jack J. Dongarrab (2010), Improvement of parallelization efficiency of batch pattern bp training algorithm using open mpi, *em* 'International Conference on Computational Science, ICCS 2010', Elsevier Ltd., Ouro Preto, MG, pp. 525–533.

Apêndice A

Diretivas OpenMP

- **#pragma omp parallel for reduction (Operador:variável(is)):** A diretiva reduction é usada para especificar um operador a uma ou mais variáveis. Para cada variável, é criada uma cópia privada em uma tarefa implícita. Após o término desta região, a variável original é atualizada com os valores das cópias privadas usando o operador especificado.
- **#pragma omp parallel for private(variável(is)) shared(variável(is)):** As variáveis marcadas como private não são compartilhadas entre threads. Já as marcadas como shared serve para explicitar que uma variável deve ser compartilhada entre as threads.
- **#pragma omp parallel firstprivate (variável(is)):** A diretiva firstprivate é usada quando é necessário que a variável privativa mantenha o valor que existia antes de iniciar o código paralelo, ou seja, se uma variável x declarada antes da região paralela tivesse valor $x = 3$ ao entrar na região paralela esta ficaria com $x = 0$, se necessário manter o valor $x = 3$ use a diretiva firstprivate, usada apenas com parallel for.
- **#pragma omp barrier:** Esta diretiva é de sincronização exigindo que todas as threads cheguem a um determinado ponto da execução para continuar, ou seja, é um barreira explicita no código, faz com que seja necessário sincronizar todas as threads.
- **#pragma omp master:** Esta diretiva faz com que apenas a thread principal (master) executar um trecho de código. Sendo este trecho ignorado pelas demais threads.
- **#pragma omp single:** Esta diretiva faz com que apenas uma thread execute um trecho de código. Ao final deste trecho, implicitamente existe uma diretiva barrier que exige que todas as outras threads cheguem aquele ponto para continuar.
- **#pragma omp critical:** Esta diretiva determina uma seção crítica do código, onde somente uma thread pode executar aquele código por vez.

- **#pragma omp parallel for schedule ((static, dynamic, guided), chunksize):** A iteração, ou as iterações, na construção de partilha são atribuídas para as threads de acordo com o método programado definido nesta cláusula.
  - **static:** Todas as threads são atribuídas as iteração antes de executar o laço de repetição. As interações são divididas igualmente entre os threads por padrão. No entanto, especificando um inteiro para um parâmetro "chunk"fixará um número *"chunk"* de iterações sequenciadas a uma determinada lista de threads.
  - **dynamic:** Algumas iterações são atribuídas a um número menor de threads. Após o término da iteração atribuída a um thread em particular, ele retorna para buscar uma das iterações restantes. O parâmetro *chunk* define o número de iterações sequenciais que são atribuídas a uma thread por vez.
  - **guided:** Um grande *chunk* de iterações sequenciadas são atribuídos a cada thread dinamicamente (*dynamic*). O tamanho do *chunk* diminui exponencialmente com cada atribuição sucessiva até um tamanho mínimo especificado no parâmetro chunk.