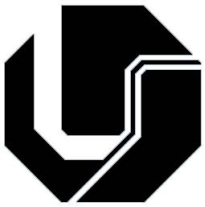
UNIVERSIDADE FEDERAL DE UBERLÂNDIA

FACULDADE DE ENGENHARIA ELÉTRICA

PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA



ESTUDO DO USO DA COMPUTAÇÃO PARALELA NO APRIMORAMENTO DO TREINAMENTO DE REDES NEURAIS ARTIFICIAIS FEEDFORWARD MULTICAMADA

WILL ROGER PEREIRA

MESTRADO

UBERLÂNDIA

2012

WILL ROGER PEREIRA

ESTUDO DO USO DA COMPUTAÇÃO PARALELA NO APRIMORAMENTO DO TREINAMENTO DE REDES NEURAIS ARTIFICIAIS FEEDFORWARD MULTICAMADA

Dissertação apresentada ao Departamento de Pós-Graduação da Faculdade de Engenharia Elétrica da Universidade Federal de Uberlândia (UFU) como parte dos requisitos para a obtenção do grau de Mestre em Engenharia Elétrica na área de Inteligência Artificial.

Banca Examinadora:

Luciano Vieira Lima, Dr. Orientador

Antônio Eduardo Costa Pereira, Dr. UFU

Keiji Yamanaka, Ph.D. UFU

José Lopes de Siqueira Neto, Dr. UFMG

UBERLÂNDIA

FEVEREIRO, 2012

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

FACULDADE DE ENGENHARIA ELÉTRICA

PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

WILL ROGER PEREIRA

ESTUDO DO USO DA COMPUTAÇÃO PARALELA NO APRIMORAMENTO DO TREINAMENTO DE REDES NEURAIS ARTIFICIAIS FEEDFORWARD MULTICAMADA

Dissertação apresentada ao Departamento de Pós-Graduação da Faculdade de Engenharia Elétrica da Universidade Federal de Uberlândia (UFU) como parte dos requisitos para a obtenção do grau de Mestre em Engenharia Elétrica na área de Inteligência Artificial.

Luciano Vieira Lima, Dr.	Alexandre Cardoso, Dr.
Orientador	Coordenador do Programa de Pós-Graduação em Engenharia Elétrica

UBERLÂNDIA

FEVEREIRO, 2012

Dedico este trabalho a Deus e a todos que me ajudaram a fazer acontecer.

Agradecimentos

Primeiramente agradeço a Deus por ter me guiado pelo caminho até a realização do mestrado. Aos meus pais Mara e Clayton pelo incentivo e pelo investimento na concretização desta etapa. Ao meu pai Hélio *(in memorian)* por me guiar e acompanhar do plano espiritual.

Ao meu orientador, prof. Dr. Luciano Vieira Lima, pela confiança depositada e por me aceitar como seu orientando. Mais este ano sobre sua orientação, agora como aluno de pós-graduação, me proporcionou uma oportunidade única de aprendizado e crescimento profissional. Seus ensinamentos e as conversas que tivemos ajudaram no caminho que trilhei para a obtenção deste título. Palavra dada é palavra cumprida.

Ao prof. Dr. Eduardo Costa, pelo vasto conhecimento em diversas áreas, principalmente em Inteligência Artificial, onde contribuiu de forma significativa em minha formação como mestre. Suas idéias, as conversas que tivemos foram muito influentes em minha formação na pós-graduação. A oportunidade ímpar de ser professor assistente de Inteligência Artificial permitiu um amadurecimento na área de docência e a certeza de que essa é a carreira que eu quero seguir. Gostaria também de agradecer o aceite para fazer parte da banca para defesa deste trabalho.

Ao prof. PhD Keiji Yamanaka, por aceitar ser parte da banca examinadora deste trabalho e pelo conhecimento passado nas disciplinas de Redes Neurais Artificiais e Algoritmos Genéticos. Os conhecimentos iniciais foram cruciais para que este trabalho abordasse este tema. Também agradeço pelos conselhos dados durante a pós-graduação.

Ao prof. Dr José Lopes Siqueira Neto, por aceitar viajar em virtude de sua participação em minha defesa. Tenho certeza que irá contribuir neste trabalho.

Aos meus amigos e amigas da UFU e do laboratório de Inteligência Artificial. Principalmente Matheus e Rubens pela companhia e ajuda durante esta etapa. Além disso à todos os professores e colegas que me incentivaram.

Ao pessoal da COPEL, principalmente a Cinara, por me ajudar a trilhar o caminho da burocracia de maneira rápida e eficiente.

À CAPES, pelo apoio financeiro e por nunca deixar a bolsa atrasar.

Meu muito obrigado a todos que se sintam parte desta conquista em minha vida, mesmo aqueles que não foram mencionados neste trabalho, além de todos aqueles que vibraram positivamente para que este trabalho fosse real.

Valeu galera!!!

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Lista de Abreviaturas

ADALINE	Adaptative Linear Neuron
AIP	Ajuste Independente dos Pesos
BP	Backpropagation
LISP	List Programing
FTHREAD	Fair Thread
MADALINE	Multiple Adaline
MRIII	MADALINE Rule III
POSIX	Portable Operating System Interface
RNA	Rede Neural Artificial

Resumo

Este trabalho compara redes neurais feedforward clássicas com um algoritmo que permite explorar o paralelismo presente em diversas máquinas disponíveis. O esquema de treinamento discutido aqui não depende da topologia da RNA, portanto pode ser uma melhor representação das redes neurais naturais produzidas ao longo da evolução animal. Porém, por ser uma estratégia que não utiliza derivada da função de ativação, é mais rústica, o que pode resultar em uma convergência mais lenta quando comparados com algoritmos que dependem de uma topologia, como o Error Backpropagation. É esperada uma melhora na velocidade da convergência com a utilização da computação paralela. No entanto, em trabalhos futuros, o paralelismo em máquinas com muitos núcleos de processamento pode mais do que compensar esta baixa velocidade, uma vez que os pesos são ajustados de forma independente. Serão analisados e comparados os comportamentos do método mais popular de treinamento de redes neurais artificiais feedforward na atualidade, o Error Backpropagation, e o algoritmo abordado neste trabalho, tanto no modo sequencial quanto paralelo.

Palavras-chave: Redes Neurais Artificiais, Gradiente Descendente, Perturbação do Peso, Scheme, paralelismo.

Abstract

This work compares classical feedforward neural networks with an algorithm that permit exploit parallelism present in widely available machines. The training scheme discussed here do not depend on the neural network topology, therefore may be a better representation of the natural neural networks produced along animal evolution. Although, for being a strategy that does not use derivative of the activation function, is more rustic, that may result in a slower convergence when compared to topology dependent algorithm, like Error Backpropagation. An improvement in the convergence speed is expected with the use of parallel computing. However, in future works, parallelism in machines with many cores may more than compensate this lower speed, once the weights are ajusted independently. The behavior of the most popular training method nowadays, the Error Backpropagation, and the algorithm discussed in this work, in either sequencial and parallel mode, will be analised and compared.

Keywords: Artificial Neural Networks, Gradient Descent, Weight Perturbation, Scheme, paralelism.

Capítulo 1

Introdução

Nos tempos modernos, a Inteligência Artificial procurou por inspiração em sistemas biológicos. Como se imagina, as redes neurais dos cérebros de animais modernos forneceram um dos primeiros modelos de comportamento inteligente que a comunidade científica tentou simular. Uma vez que os neurônios são os componentes principais para o funcionamento do cérebro, pesquisadores da área tentaram aprender como estas intrigantes células funcionam.

A ciência moderna alega que a evolução é o mecanismo que organiza o cérebro e faz a inteligência ser possível. Assim, nos últimos anos, cientistas e pesquisadores que trabalham com inteligência artificial empreenderam grandes esforços na pesquisa da organização neural, de modo a tentar entender o processo evolucionário que é produzido pela grandiosa máquina, o cérebro. Este trabalho apresenta um sistema para ler e processar dados e atributos biológicos, de modo a produzir e treinar estruturas computacionais que imitam as redes neurais capazes de representar a informação fornecida.

Para explorar máquinas de múltiplos núcleos de processamento e outras tecnologias que estão disponíveis, é necessário fazer bom uso da computação paralela, seja utilizando processos independentes, threads ou comunicação assíncrona. Também são necessárias linguagens e ferramentas de programação que não fiquem obsoletas antes que a tecnologia fique madura, ou os recursos financeiros disponíveis para implementação e desenvolvimento.

Diversas são as aplicações em que são empregadas as redes neurais artificiais. Vão desde o reconhecimento de padrões, imagens e sinais, passando pela tomada de decisões sequenciais, aproximação de funções e regressão simbólica, até aplicações na robótica e em jogos iterativos. São utilizadas para "aprender"e "generalizar"um comportamento através de exemplos, desde uma sequência de números até estratégias para vencer um jogo.

No desenvolvimento deste trabalho, o hardware é bastante importante, já que o tempo de execução será uma das grandezas medidas e comparadas. Então é importante detalhar o hardware que será utilizado, de forma a permitir a comparação entre os resultados obtidos neste trabalho e testes feitos em outras máquinas. A máquina utilizada para realizar todos os testes é dotada da seguinte configuração:

```
Máquina utilizada:
  - Processador: Intel core i7 (8 núcleos - 4 reais e 4 virtuais)
  - Memória RAM: 3GB
  - Sistema Operacional: Linux Ubuntu 10.04

Compilador utilizado:
  - Linguagem: Bigloo
  - Versão: 3.6a
```

Este trabalho será dividido nos seguintes capítulos:

- No capítulo 2 será discutida a linguagem de programação utilizada neste trabalho, o Scheme Bigloo, abordando suas vantagens, propriedades e áreas de atuação. Além disso serão abordadas a compilação e a otimização, sendo esta última responsável por garantir uma velocidade impressionante para quase todas as aplicações;

- Já no capítulo 3 será mostrado um breve histórico sobre as Redes Neurais, abordando alguns fatos importantes desde a descoberta do neurônio, passando pela invenção do neurônio artificial, até a invenção do método de treinamento mais popular atualmente;

- O capítulo 4 aborda o método discutido neste trabalho, tanto no aspecto teórico quanto práticos, de modo a demonstrar suas inspirações, vantagens, desvantagens e

implementação. Também consta uma pequena demonstração, visando representar a porta lógica ou-exclusivo;

- Por sua vez, o capítulo 5 mostra como utilizar o método abordado no capítulo 4 em um exemplo mais complexo, além de compará-lo com o método mais popular de treinamento de RNAs atualmente, o error backpropagation;

- No capítulo 6 será mostrado o que é e como utilizar a computação paralela, de modo a permitir que fragmentos de código sejam executados simultaneamente por máquinas com múltiplos núcleos de processamento. Também serão abordadas estratégias para permitir este feito, além de ser construída uma aplicação para demonstrar as vantagens e dificuldades deste paradigma;

- O capítulo 7 tem como objetivo aplicar os conhecimentos apresentados no capítulo 6 para treinamento de redes neurais artificiais, além de realizar comparações entre uma aplicação sequencial e uma que utiliza computação paralela;

- Já no capítulo 8 poderão ser vistos os trabalhos futuros que poderão ser feitos para avançar nesta pesquisa, além da conclusão do trabalho, baseada nos resultados obtidos.

Neste trabalho pretende-se mostrar como um método já existente para treinamento de redes neurais artificias, independente de topologia, pode ser aprimorado com o uso de estratégias de computação paralela, além de mostrar as vantagens e desvantagens deste aprimoramento.

Capítulo 2

Sobre a Linguagem Utilizada

Neste trabalho, todos os programas serão codificados em Scheme Bigloo [BIGLOO 2011]. Como todo Scheme, foi influenciado pelo LISP, a *lingua franca* da Inteligência Artificial, de tal maneira que mantém diversas características desta linguagem. Dentre elas, destacam-se o cálculo lambda, Read-Eval-Print-Loop (REPL) e a notação infixa. Sua principal característica, sem dúvidas, é sua velocidade. Utilizando otimização, o desempenho de um programa aumenta de maneira impressionante.

O Bigloo tenta fazer com que o Scheme seja prático, oferecendo recursos que normalmente estão presentes em linguagens tradicionais, mas não são oferecidos pelo Scheme ou por linguagens de paradigma funcional. Ele se propõe a entregar códigos binários pequenos e rápidos, que podem ser executados diretamente. Além disso, ele permite uma conexão total com C, Java e C#.

Utilizando bibliotecas, é possível utilizar interface gráfica. TK, GTK e JAPI são exemplos de toolkits que possuem bibliotecas prontas para se 6utilizar com Bigloo.

É possível utilizá-lo em sistemas operacionais Windows, Linux e Mac OS. Possui diversas funcionalidades nativas: Interface com arquivos, programação cliente-servidor, processos, threads[1], SQL, Unicode, etc. É possível gerar código C, Java e C#, o que facilita a integração de programas feitos em Bigloo com ferramentas largamente utilizadas, como Matlab.

[1]Não disponível em Windows.

Uma característica muito poderosa do Scheme é o sistema de macros, que permite ao usuário ampliar a sintaxe básica. Isto é possível e fácil porque dados e programas possuem a mesma representação: ambos são listas. Se o código fonte contém além da aplicação, os macros necessários para processar a sintaxe utilizada na escrita do código-fonte, a aplicação se torna praticamente imune a obsolescência.

A motivação para utilização desta linguagem é o fato de ela gerar códigos binários pequenos e rápidos, além de ser simples, objetiva, de modo a aumentar a produtividade.

Um arquivo que contém um código em Bigloo possui o sufixo *.scm*. Para compilá-lo, o seguinte comando deve ser utilizado.

```
bigloo nomedoarquivo.scm -o nomedoprograma
```

Para utilizar a otimização, basta apenas utilizar o prefixo *-On*, com n pertencendo a [1..6]. No entanto, a otimização máxima, a nível de benchmark, é alcançada quando um código é compilado com o prefixo *-Obench*. É importante ressaltar que nem todos os programas podem ser otimizados no nível máximo. Ao longo da experiência do autor com esta linguagem de programação, alguns programas não funcionaram com esta otimização. Mas isto vai depender de quais funções foram utilizadas e da forma que foi codificado o programa. Para utilizar a otimização *-Obench*, compile o programa com o seguinte comando.

```
bigloo -Obench nomedoarquivo.scm -o nomedoprograma
```

Capítulo 3

Breve Histórico das Redes Neurais

Neste capítulo serão citados alguns dos principais trabalhos na área de redes neurais, principalmente no treinamento de redes neurais artificiais feedforward, foco deste trabalho.

O cientista espanhol Don Santiago Ramon y Cajal, ganhador do prêmio nobel de medicina de 1906, estudou a fisiologia do sistema nervoso e concluiu que ele era orientado por células que ele chamou de neurônios [RAMON y CAJAL 1906]. Atualmente, a comunidade científica acredita que um dado neurônio recebe impulsos de outros neurônios por conexões, chamadas sinapses.

Tipicamente, um neurônio coleta sinais de outros através de estruturas chamadas dendritos. Se a intensidade dos sinais recebidos passa de um limite, o neurônio dispara e manda estímulos elétricos através de uma haste, chamada axônio. Ela se parte em múltiplos dendritos, cujas sinapses vão ativar outros neurônios. Então, para juntar tudo, no fim de cada dendrito, uma sinapse converte a atividade neural em estímulo elétrico que inibe ou excita outros neurônios.

A era moderna das redes neurais artificiais começou com o trabalho pioneiro de Warren McCulloch e Walter Pitts, em 1943. Eles formalizaram o primeiro modelo de redes neurais artificiais. Este modelo descreve a lógica de cálculo das redes neurais, unindo conhecimentos da matemática e da neurofisiologia [HAYKIN 1999].

Em 1949, Donald Hebb introduz a capacidade de aprendizado em um neurônio, onde apresenta o primeiro algoritmo de treinamento. Ele é baseado na seguinte frase: "Células

que disparam juntas, permanecem conectadas"[HEBB 1949]. Ele assumia que a aprendizagem do conhecimento era alcançado pelo fortalecimento das conexões entre os neurônios.

Já em 1958, Frank Rosenblatt criou o Perceptron, um modelo de RNA composto por uma camada de neurônios, capaz de aprender tudo o que pudesse representar. Ele também demonstrou que, se as sinapses fossem ajustáveis, o Perceptron poderia ser treinado para aprender a classificar padrões em classes linearmente separáveis [ROSENBLATT 1958]. Foi o primeiro modelo de treinamento a se basear no erro.

Bernard Widrow e seu orientado Ted Hoff criaram, em 1960, o ADALINE. Trata-se de uma rede neural de uma camada cujo treinamento é baseado na regra delta, que utiliza o gradiente descendente do erro para ajustar os pesos. A diferença para o Perceptron é que além de utilizar a regra delta, os pesos são ajustados de acordo com a saída direta dos neurônios, ou seja, a soma do produto entre os pesos e as entradas [WIDROW e HOFF 1960].

David Rumelhart, Geoffrey Hinton e Ronald Williams aperfeiçoaram as idéias do Perceptron e do Adaline, em 1986, originando o algoritmo Backpropagation. Trata-se de uma generalização da regra delta, também baseada no gradiente descendente do erro. Neste algoritmo, é calculado o erro na camada de saída, e este erro é retro-propagado para as camadas anteriores, permitindo o ajuste dos pesos dos neurônios de todas as camadas, inclusive de camadas escondidas [RUMELHART et al. 1986].

Além destas técnicas de treinamento, existem diversas outras com o intuito de fazer com que uma RNA aprenda o que é desejado. Também existem diversas outras arquiteturas, com os mais variados processos de treinamento [FAUSSETT 1993] e [HAYKIN 1999]. Porém, como não é de interesse deste trabalho, não serão abordados.

Atualmente, o error BP é um dos métodos mais populares, senão o mais popular, para treinamento de redes neurais feedforward[1]. O método abordado neste trabalho será comparado com ele.

[1]Redes neurais artificiais com entrada e saída sem loops em suas conexões e sem competição.

Capítulo 4

O Método AIP

Redes neurais artificiais são capazes de aprender, representar e generalizar, baseados em exemplos, diversos padrões e comportamentos, sejam eles funções, imagens, decisões, ações, etc. Em RNAs feedforward, este treinamento ocorre ajustando-se os pesos das conexões sinápticas diversas vezes até que o aprendizado seja concluído, ou seja, o erro apresentado pela RNA seja razoável.

Este ajuste dos pesos funciona baseado em dois momentos: Primeiramente, o erro da RNA é calculado baseado nas saídas reais e ideais. Posteriormente, baseado neste erro, os pesos da RNA são ajustados de forma que a saída calculada se aproxime da saída desejada [FAUSSETT 1993].

O método de treinamento em questão, aqui chamado de AIP (Ajuste Independente dos Pesos), é baseado em técnicas existentes para encontrar o ajuste dos pesos de uma RNA feedforward. Utilizando o gradiente descendente com aproximação direta é a maneira mais simples de se realizar implementações paralelas [JABRI e FLOWER 1992]. Por envolver um esforço computacional maior, RNAs menores são preferidas pois tanto a computação quanto o treinamento são mais rápidos, uma vez que um número menor de pesos são ajustados. Mas nada impede que seja utilizada uma RNA maior. Neste trabalho será discutida uma abordagem que visa aprimorar o treinamento de uma RNA por este método.

4.1 Obtenção da Saída da RNA

No perceptron, os sinais de entrada multiplicados por um peso (com um peso diferente para cada sinal de entrada), são somados. Se o resultado for maior do que um limite, ele produz um sinal excitatório. Este processo foi aperfeiçoado mudando a função de ativação de degrau para uma função como a sigmoide[1], e pegando o valor desta função como saída do perceptron. A função sigmoide binária pode ser definida como:

$$\varsigma_{bin}(x) = \frac{1}{1 + e^{-x}}$$

Em Bigloo, o código desta função ficaria como no código 4.1.

Código 4.1: Função sigmoide binaria.

```
1 (define (sigma x) (/ 1 (+ 1 (exp (- x)))))
```

É possível treinar o perceptron para reconhecer parâmetros de entrada. De fato, dados exemplos suficientes, um algoritmo irá modificar os pesos da RNA, e então ela conseguirá distinguir entre dois ou mais padrões. O algoritmo de aprendizado de uma RNA deve ter um conjunto de exemplos para treinamento. A priori, será adotado como exemplo a porta lógica XOR, um conjunto não-linearmente separável. No código 4.2 seguem os exemplos:

Código 4.2: Função Exemplos de treinamento porta lógica XOR.

```
1 (define inputs '((0 0) (0 1) (1 0) (1 1)))
2 (define targets '((0) (1) (1) (0)))
```

O perceptron possui um procedimento, que usa a fórmula $\varsigma(\sum w_i x_i)$ para fornecer o valor de saída, dados conjuntos de entrada. Neste trabalho, este tipo de procedimento será construído utilizando uma closure.

Os termos *let over lambda*, *ambiente léxico salvo* ou *closure*, se referem a um bloco de código que pode ser passado como argumento para uma função. Basicamente, pode ser entendido como uma dualidade função/ambiente, onde uma outra função retorna outra função como resultado [HOYTE 2008]. Isto permite a geração de múltiplas funções com

[1] Existem outras funções que se adequam às necessidades de uma função de ativação.

características semelhantes utilizando apenas uma função. Muito simples de se implementar em linguagens onde uma função pode ser fornecida como argumento para outra função, como o Bigloo.

Neurônios artificiais são entidades com características semelhantes, porém com atributos diferentes. Com estas propriedades, eles são perfeitos para a utilização de closure. Em Bigloo, a função que cria um neurônio com esta funcionalidade pode ser codificado como no código 4.3.

Código 4.3: Neurônio artificial.

```
(define (newneuron weights indexes)
   (lambda(inputs)
      (let loop[(i indexes) (x (cons 1.0 inputs)) (acc 0.0)]
         (cond
            [(or (null? i) (null? x))
               (sigma acc)]
            [else
               (loop (cdr i) (cdr x) (+ acc (* (vector-ref weights (car i))
                  (car x))))])))))
```

Quando a função *newneuron* é executada, dado um vetor de pesos e uma lista de índices do vetor de pesos, uma função do neurônio será criada e retornada. Isto pode ser utilizado para criar todos os neurônios da RNA. Como closures do Bigloo são objetos de primeira classe, elas podem ser passadas como argumento para outra função, criadas dinamicamente ou guardadas em uma variável, assim como se faz com uma string ou um número inteiro. Se a closure do neurônio gerado for executada, dada uma lista de entrada, ela irá retornar a saída do neurônio.

Assim como o neurônio, é possível utilizar uma estrutura semelhante para criar a closure de uma RNA, como na figura 4.1. Isto é possível através de uma codificação tal como no código 4.4.

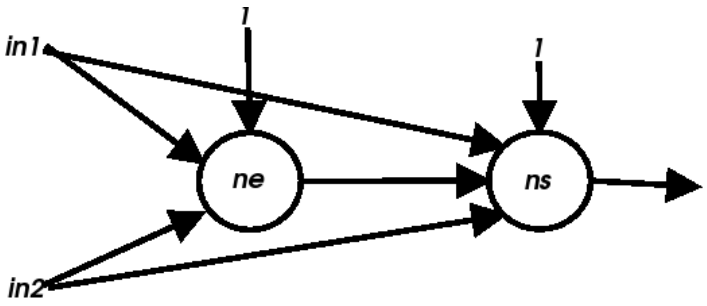


Figura 4.1: Esquema da RNA gerada pela função *gate*

Código 4.4: Rede neural artificial.

```
(define (gate weights)
   (let [(ne (newneuron weights '(0 1 2)))
         (ns (newneuron weights '(3 4 5 6)))]
        (lambda(in)
          (cond
             [(null? in)
                weights]
             [else
                (let [(outc1 (list (ne in)))]
                   (list
                      (ns (append in outc1))
                   )
                )])))))
```

Observe que na figura 4.1, assim como no código 4.4, há conexões entre a entrada e o neurônio que fornece a saída. Isto pode acontecer, pois o treinamento pelo método AIP independe de topologia, ou seja, não é necessário organizar a RNA em camadas.

Quando a função *gate* é executada, dado um vetor de pesos como argumento, ela irá retornar uma função que representa a RNA. Quando esta função retornada é executada, dada uma lista contendo as entradas, o resultado final da RNA será retornado. Os pesos são guardados no vetor passado como argumento para a função *gate*, que cria a RNA. Os verdadeiros pesos devem ser encontrados através de um algoritmo de treinamento.

4.2 Cálculo do Erro Quadrático Total

O algoritmo de aprendizagem é baseado no cálculo e correção do erro da RNA. O erro de um conjunto de exemplos é calculado pela expressão abaixo, onde e é um exemplo, v_c

o objetivo e v a saída real.

$$E = \sum_{i=0}^{n} (v_c(e)_i - v(e)_i)^2$$

Em Bigloo, o cálculo deste erro é realizado pela função que consta no código 4.5.

Código 4.5: Erro quadrático total.

```
(define (errsum targets reals)
   (define err 0.0)
   (map (lambda(x1 x2) (map (lambda(y1 y2) (set! err (+ err (* (- y1 y2) (-
       y1 y2))))) x1 x2)) targets reals)
   err)
```

4.3 Atualização dos Pesos Sinápticos

A função *updateweights* modifica os pesos de forma a minimizar os erros ao longo dos ciclos. O gradiente descendente é um algoritmo de otimização que encontra um mínimo local de uma função, com passos proporcionais ao negativo do gradiente descendente da função no ponto atual. A cada passo, o vetor de pesos é atualizado de acordo com a seguinte fórmula:

$$\omega_{n+1} = \omega_n + adj_n$$

$$adj_n \equiv \triangle\omega_n$$

Dotado do vetor de pesos e do vetor de ajustes, o código 4.6 serve para atualizar os pesos da RNA, em Bigloo.

Código 4.6: Atualização dos pesos.

```
(define (updateweights weights adjusts)
   (define tam (vector-length weights))
      (let loop[(i 0)]
         (cond
            [(< i tam)
               (vector-set! weights i (+ (vector-ref weights i) (vector-ref
                   adjusts i)))
               (loop (+ i 1))])))
```

4.4 Cálculo dos Ajustes dos Pesos

Diferente do método BP, todos os ajustes dos pesos são baseados no erro final da RNA, sem estimativas. Além disso, trata-se de um método sem diferenciação, ou seja, não é necessário utilizar a derivada da função de ativação.

Mas assim como o BP, o AIP (Ajuste Independente dos Pesos) é um método baseado no gradiente descendente. Porém este último se baseia no gradiente descendente com aproximação direta, utilizando o método das diferenças finitas [CAUSON e MINGHAM 2010].

Em 1988, a regra MADALINE III (MRIII) foi desenvolvida por Andes et al. [WIDROW e LEHR 1990 com o objetivo de treinar RNAs feedforward multicamada, utilizando como função de ativação a sigmoide. Em 1990, Widrow e Lehr provaram que esta regra era matematicamente equivalente ao método error backpropagation. No entanto, apesar de ser uma regra mais simples, cada ajuste de peso utilizando MRIII envolve um esforço consideravelmente maior do que no BP.

A MRIII pode ser considerada como uma implementação do gradiente utilizando "perturbação neural", de acordo com a seguinte equação.

$$adj_{ij} = -\alpha \frac{\triangle E}{\triangle net_i} x_j$$

De maneira genealizada, o ajuste de um peso sináptico de uma RNA é o gradiente descendente do erro ($-\nabla \mathbf{E}(\omega)$). Porém, para permitir o aprendizado de uma RNA, estudiosos acrescentaram uma taxa de aprendizagem (α), para facilitar a convergência do erro para um mínimo. Assim, o ajuste de um peso sináptico i pode ser descrito da seguinte maneira:

$$adj_i = -\alpha \nabla \mathbf{E}(\omega_i) = -\alpha \frac{\partial E}{\partial \omega_i}$$

Baseada no MRIII, Marwan Jabri e Barry Flower criaram, em 1992, uma técnica chamada "perturbação do peso" [JABRI e FLOWER 1992]. Com ela, é possível calcular o gradiente descendente do erro com aproximação direta. Para tanto, é utilizado o método das diferenças finitas. Uma perturbação ($\delta\omega$) é somada a um peso, e levando-se em

consideração o erro da RNA com e sem perturbação, $E(\omega + \delta\omega)$ e $E(\omega)$ respectivamente, é possível calcular o ajuste de um peso i.

Caso a perturbação $\delta\omega$ seja um número suficientemente pequeno, utilizando o método das diferenças finitas, do tipo forward e de primeira ordem em ω, ter-se-á a seguinte equação:

$$adj_i = -\alpha \frac{\partial E}{\partial \omega_i} = -\alpha \frac{E(\omega_i + \delta\omega) - E(\omega_i)}{\delta\omega}$$

Nesta fórmula, se $\alpha > 0$ é um número suficientemente pequeno, $E(\omega_{n+1}) < E(\omega_n)$. Considerando um ω_0, a sequencia $\omega_0, \omega_1, \omega_2 \ldots$ irá convergir para um mínimo. O mínimo será alcançado quando o gradiente for zero, ou perto de zero, em situações práticas.

Imagine que há um terreno com montanhas e vales. Uma posição neste terreno é dado pelas coordenadas ω_1 e ω_2, análogas a latitude e longitude. A altura neste terreno representa o erro. O algoritmo de treinamento move a rede de uma posição inicial para um vale, ou seja, para uma posição onde o erro alcance um mínimo local.

A situação ideal acontece quando a rede chega a um mínimo global. Todavia, métodos baseados em gradiente ficam presos em mínimos locais com certa frequência. Isto acontece porque quando a RNA alcança um mínimo, local ou global, seu gradiente fica próximo de zero. De fato, o algoritmo sabe se um mínimo foi alcançado quando o valor do gradiente fica próximo de zero. O problema é como saber se o mínimo é local ou global. Da mesma forma que o BP, este método também tem problemas com mínimos locais.

O algoritmo para calcular o ajuste de um peso é descrito a seguir:

```
Defina: erro = erro quadrático da RNA
        w = vetor de pesos da RNA
        newrna = uma nova RNA, com pesos w
        alfa = taxa de aprendizagem
        dw = perturbação
Adicione dw ao peso desejado em newrna
Calcule o erro quadrático da newrna
Retorne - alfa * [(newerro - erro)/dx]
```

A função *AIPadj* irá encontrar o ajuste para determinado peso da RNA, e pode ser

codificada como no código 4.7.

Código 4.7: Ajuste de um peso da RNA.

```
(define (AIPadj i weights erro inputs targets alfa dx)
   (define tam (vector-length weights))
   (define newrna (gate (copy-vector weights tam)))
   (vector-set! (newrna '()) i (+ (vector-ref (newrna '()) i) dx))
   (let [(newerro (errsum targets (map (lambda(x) (newrna x)) inputs)))]
      (- (* alfa (/ (- newerro erro) dx)))))
```

Para calcular o ajuste de todos os pesos da RNA, o algoritmo é descrito logo abaixo:

```
Defina: erro = erro quadrático da RNA
        adj = vetor de ajustes
        tam = tamanho de adj
Loop i=0
  Se i<tam
    Encontre o ajuste do peso i e armazene em adj[i]
    Retorne para Loop
  Caso contrário
    Ajuste terminado
Fim Loop
```

Já a função *AIP* irá aplicar a função *AIPadj* em todos os pesos da RNA, de modo a encontrar o vetor de ajustes, utilizado pela função *updateweights*. O código 4.8 mostra como programar esta função.

Código 4.8: Encontrando o vetor de ajustes.

```
(define (AIP adj weights erro inputs targets alfa dx)
   (define tam (vector-length weights))
   (let loop[(i 0)]
      (cond
         [(< i tam)
            (vector-set! adj i (AIPadj i weights erro inputs targets alfa
               dx))
            (loop (+ i 1))])))
```

4.5 Treinamento de uma RNA pelo método AIP

O treinamento de uma RNA feedforward (cálculo do vetor de ajustes) e a atualização do vetor de pesos ocorre até que uma de duas coisas aconteça: O erro quadrático total

fique abaixo de um valor estimado ou o número de ciclos gastos ultrapasse um limite. O algoritmo para treinar qualquer RNA pelo método AIP é descrito a seguir:

```
Defina: erro = erro quadrático da RNA
        ciclos = ciclos gastos
        errormax = erro quadrático máximo permitido
        cyclesmax = número máximo permitido de ciclos
Loop
  Se (erro < errormax) ou (ciclos > cyclesmax)
    Treinamento terminado
  Caso contrário
    Encontre o vetor de ajustes
    Atualize o vetor de pesos baseado no vetor de ajustes
    Retorne para Loop
Fim Loop
```

No código 4.9 é mostrado como fazer uma função para treinar uma RNA pelo método AIP. Ela já utiliza as funções criadas em códigos anteriores para encontrar o vetor de ajustes e atualizar o vetor de pesos.

Código 4.9: Treinamento da RNA.

```
1  (define (train rna inputs targets cicmax errmax alfa dx)
2     (define adj (make-vector (vector-length (ann '())) 0))
3     (let loop[(ciclos 0) (erro (errsum targets (map (lambda(x) (rna x))
          inputs)))]
4        (cond
5           [(or (>= ciclos cicmax) (< erro errmax))
6              erro]
7           [else
8              (AIP adj (rna '()) erro inputs targets alfa dx)
9              (updateweights (rna '()) adj)
10             (loop (+ ciclos 1) (errsum targets (map (lambda(x) (rna x))
                  inputs)))])))
```

O programa completo encontra-se no Anexo A, código A.1. Caso ele seja compilado, executado e a RNA for testada, serão obtidos resultados corretos como a seguir:

```
Treinamento concluído:
Ciclos gastos: 1020
Erro quadrático total: 0.0099946758369243
Tempo gasto: 100 milisegundos.
0 0 = (0.042143253872158)
0 1 = (0.94853664705752)
1 0 = (0.94853706165411)
1 1 = (0.05405285626788)
```

Neste caso, utilizando somente a otimização com prefixo *-Obench*, vide capítulo 2, sem mudar nada no código, a performance é 2.5 vezes melhor. Veja os resultados obtidos, com a otimização, a seguir:

```
Treinamento concluído:
Ciclos gastos: 1035
Erro quadrático total: 0.0099974848745369
Tempo gasto: 40 milisegundos.
0 0 = (0.04214946180328)
0 1 = (0.94852937785363)
1 0 = (0.94853002247702)
1 1 = (0.054060375635775)
```

Como uma porta lógica ou-exclusivo é um exemplo demasiadamente simples, será confeccionada uma aplicação mais complexa, de modo a demonstrar melhor as características e o desempenho deste método diante de um desafio maior.

Capítulo 5

Aplicação do Método AIP

O software para demonstrar uma aplicação do método AIP, abordado neste trabalho, compara uma RNA que será treinada tanto pelo método BP [RUMELHART et al. 1986], como pelo método AIP. O caso de estudo é baseado em uma base de dados, não-linearmente separável, que contém atributos da flor Iris [FRANK e ASUNCION 2010]. Os dados da base de dados se encontram no Anexo C, código C.1. O objetivo é reconhecer a que classe pertence uma flor, baseando-se no comprimento e largura de pétalas e sépalas.

5.1 Metodologia

A base de dados é composta de 150 amostras divididas em três classes, dispostas da seguinte maneira:

- Iris-setosa: 50 amostras;
- Iris-versicolor: 50 amostras;
- Iris-virginica: 50 amostras.

As amostras foram divididas em dois grupos menores: O grupo de treinamento e o grupo de teste ou grupo real. Cada grupo tem a mesma proporção de indivíduos. O primeiro é utilizado para treinar a RNA e para testar seu comportamento, enquanto o segundo é somente para ser utilizado como referência.

Na aplicação, os dados são extraídos, normalizados e devidamente divididos. Na RNA existem três neurônios de saída, cada um responsável por ativar quando uma classe é reconhecida. As classes foram divididas de acordo com as seguintes saídas ideais:

1. Primeiro neurônio ativo (1 -1 -1): Iris-setosa;

2. Segundo neurônio ativo (-1 1 -1): Iris-versicolor;

3. Terceiro neurônio ativo (-1 -1 1): Iris-virginica;

Um neurônio que fornece a saída final dispara (saída 1) somente se sua saída for maior ou igual a 0.7, e inibe (saída -1) somente se sua saída for menor ou igual a -0.7. Caso contrário, é atribuído o valor 0, identificando a saída como não reconhecida. Isto foi feito para melhorar a certeza que a RNA terá ao identificar as classes.

Além deste tratamento da saída, toda a base de dados foi catalogada, de modo a saber quando uma entrada é classificada como uma flor errada. Isto visa melhorar a qualidade dos dados estatísticos. São considerados erros as seguintes situações:

- A saída de algum neurônio que fornece a saída final tiver valor 0;

- Nenhum neurônio, ou mais de um, fornecer saída final ativa, ou seja, 1;

- A RNA fornecer como saída uma classe diferente da catalogada para aquela entrada.

Como foram utilizados valores dentro da escala bipolar, a função de ativação usada foi a sigmoide bipolar, que pode ser codificada, em Bigloo, como no código 5.1:

$$\varsigma_{bip}(x) = \frac{2}{1 + e^{-x}} - 1$$

Código 5.1: Função Sigmoide Bipolar

```
1 (define (bipsig x) (- (/ 2 (+ 1 (exp (- x)))) 1))
```

A RNA utilizada foi obtida empiricamente e pode ser codificada como no código 5.2[1]. Experimentos posteriores irão modificar a estrutura da RNA, mas a priore ela conta com os seguintes atributos:

- 4(treze) atributos de entrada;
- 10(cinco) neurônios na camada escondida;
- 3(três) neurônios na camada de saída;
- Um total de 83(oitenta e três) pesos sinápticos sendo:
  - : 40(quarenta) entre a entrada e a camada escondida;
  - : 30(trinta) entre a camada escondida e a camada de saída;
  - : 13(treze) para os neurônios (bias);

Código 5.2: Código da RNA utilizada na aplicação

```
1  (define (gate weights)
2     (let [(n10 (newneuron weights (map (lambda(x) (+ x (* 5 0))) (
          list-tabulate 5 values))))
3           (n11 (newneuron weights (map (lambda(x) (+ x (* 5 1))) (
              list-tabulate 5 values))))
4           (n12 (newneuron weights (map (lambda(x) (+ x (* 5 2))) (
              list-tabulate 5 values))))
5           (n13 (newneuron weights (map (lambda(x) (+ x (* 5 3))) (
              list-tabulate 5 values))))
6           (n14 (newneuron weights (map (lambda(x) (+ x (* 5 4))) (
              list-tabulate 5 values))))
7           (n15 (newneuron weights (map (lambda(x) (+ x (* 5 5))) (
              list-tabulate 5 values))))
8           (n16 (newneuron weights (map (lambda(x) (+ x (* 5 6))) (
              list-tabulate 5 values))))
9           (n17 (newneuron weights (map (lambda(x) (+ x (* 5 7))) (
              list-tabulate 5 values))))
10          (n18 (newneuron weights (map (lambda(x) (+ x (* 5 8))) (
              list-tabulate 5 values))))
11          (n19 (newneuron weights (map (lambda(x) (+ x (* 5 9))) (
              list-tabulate 5 values))))
12          (n20 (newneuron weights (map (lambda(x) (+ x (* 11 0) (* 5 10))) (
              list-tabulate 11 values))))
13          (n21 (newneuron weights (map (lambda(x) (+ x (* 11 1) (* 5 10))) (
              list-tabulate 11 values))))
```

[1]A parte comentada faz-se necessária quando a RNA é treinada pelo método BP, pois a saída da camada escondida faz parte do algoritmo de treinamento.

```
14          (n22 (newneuron weights (map (lambda(x) (+ x (* 11 2) (* 5 10))) (
                list-tabulate 11 values))))]
15      (lambda(in)
16          (cond
17              [(null? in)
18                  weights]
19              [else
20                  (let [(outc1 (list (n10 in) (n11 in) (n12 in) (n13 in) (n14
                        in) (n15 in) (n16 in) (n17 in) (n18 in) (n19 in)))]
21                      ;(list
22                      ;    outc1
23                          (list
24                              (n20 outc1)
25                              (n21 outc1)
26                              (n22 outc1)
27                          )
28                      ;)
29                  )])))))
```

5.2 Resultados

O código completo deste trabalho se encontra no Anexo D. A execução e os testes foram feitos utilizando parâmetros de entrada obtidos empiricamente, como na figura 5.1.

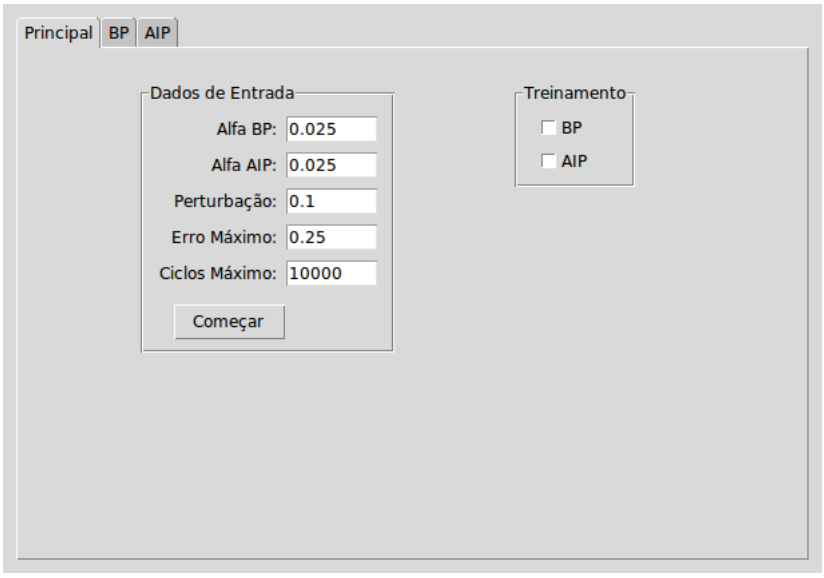


Figura 5.1: Janela Principal da Aplicação

Para comparar os métodos BP e AIP, foi utilizada a mesma taxa de aprendizagem[2]. Além disso, a RNA utilizada será a mesma para os dois métodos de treinamento, com

[2]O método BP suporta uma taxa de aprendizagem maior.

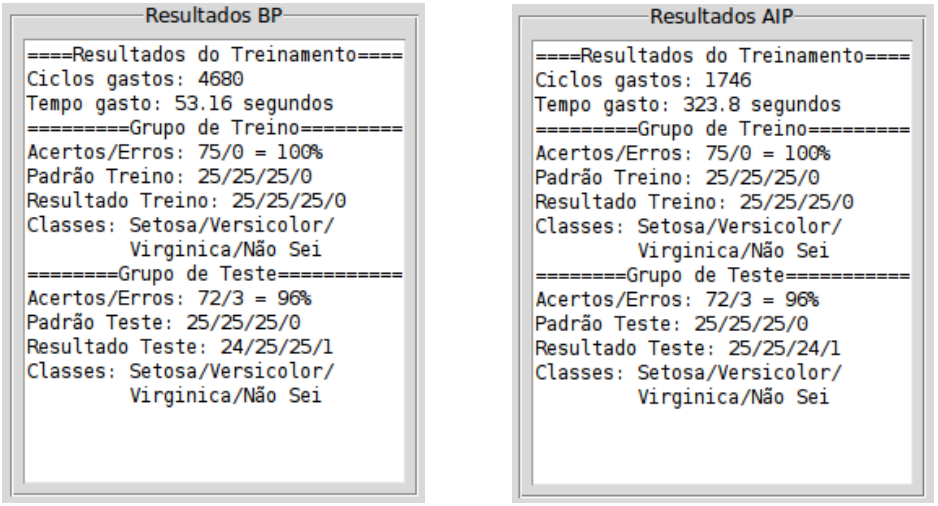
topologia multicamada[3]. Os pesos iniciais serão aleatórios (entre -0.5 e 0.5), porém os mesmos para ambos os métodos.

Visando a qualidade dos resultados, o vetor de pesos iniciais será sorteado uma única vez. Ele será utilizado para realizar todos os testes de treinamento de RNAs ao longo do trabalho. No código 5.3 é registrado todos os seus 83(oitenta e três) pesos.

Código 5.3: Vetor de pesos utilizado para testes.

```
1 #(0.499 0.262 0.181 -0.079 -0.361 -0.162 -0.153 0.392 0.435 0.113 0.212
    -0.174 0.493 0.194 -0.169 0.198 -0.095 0.278 -0.463 0.01 -0.454 0.24
    -0.339 -0.121 0.131 -0.206 -0.404 0.358 -0.434 0.189 -0.295 -0.125
    -0.049 -0.305 -0.394 0.4 0.344 -0.047 -0.399 0.279 -0.124 -0.378 -0.086
    -0.131 0.317 -0.445 -0.434 -0.278 0.334 -0.086 0.232 -0.31 -0.347 -0.297
     0.38 0.285 0.308 0.476 0.143 -0.316 0.475 -0.342 0.06 -0.075 -0.146
    0.166 0.135 -0.303 -0.071 0.237 -0.214 0.306 0.36 0.201 -0.325 0.487
    0.256 0.052 0.018 0.4 0.466 0.06 -0.411)
```

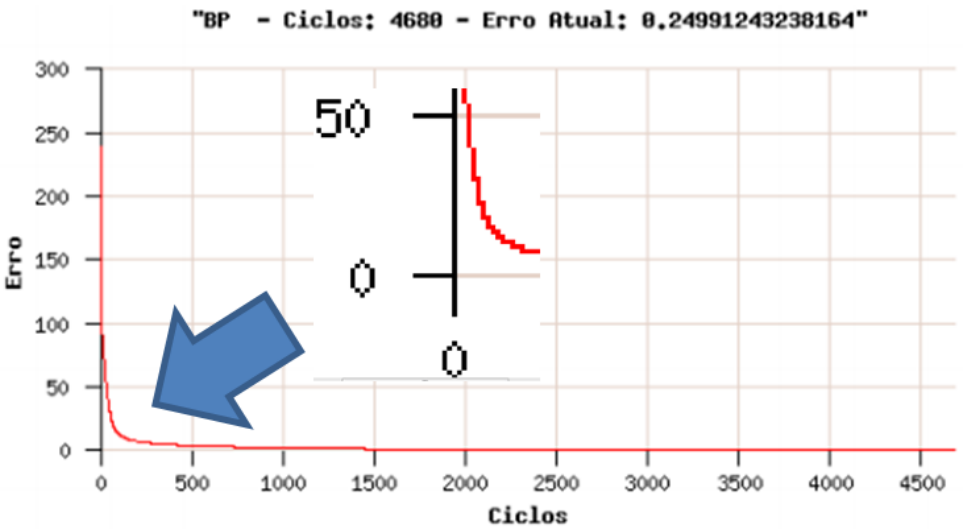
Os resultados do treinamento pelo método BP e AIP podem ser observados na figura 5.2. O comportamento do erro quadrático total por ciclo pode ser analisado através do gráfico da figura 5.3. O tempo gasto está dentro da média de diversos testes realizados.



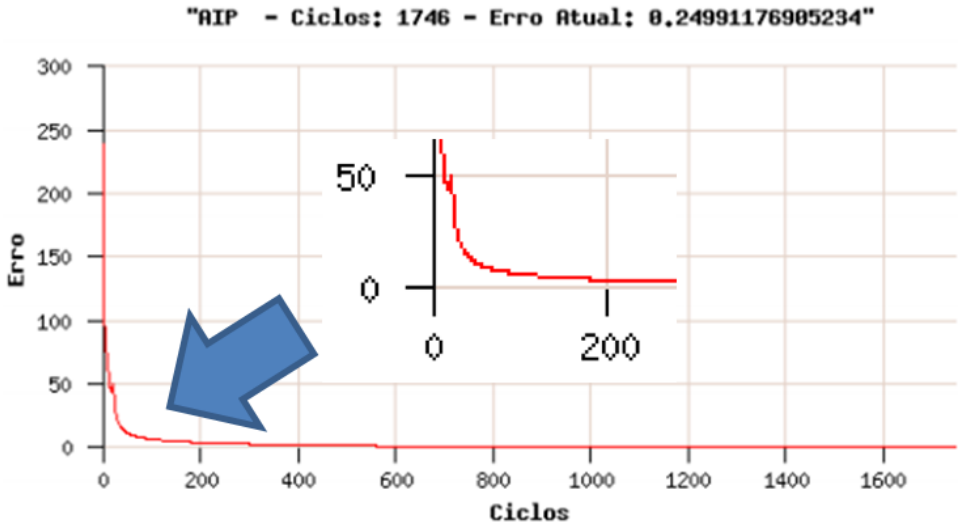
(a) Método BP (b) Método AIP

Figura 5.2: Resultados da RNA treinada pelos métodos BP e AIP

[3]O método AIP suporta uma arquitetura sem topologia.



(a) Método BP

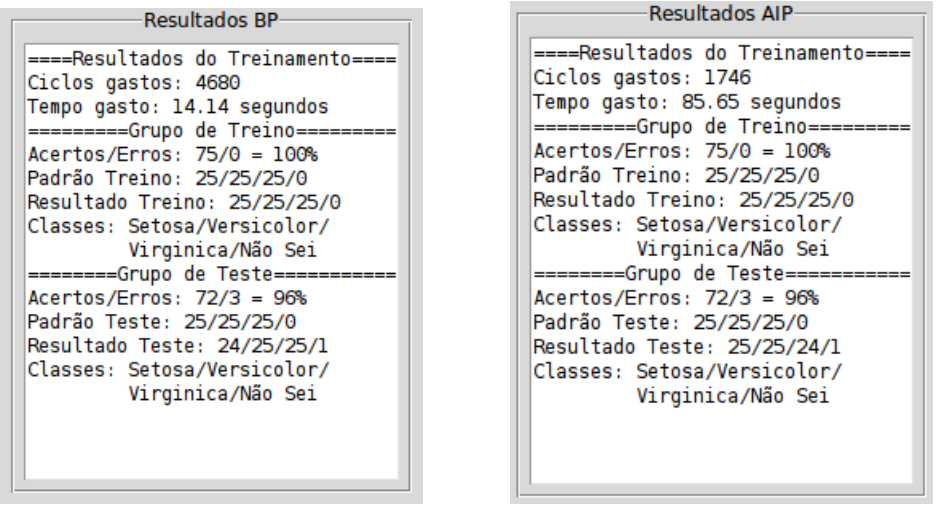


(b) Método AIP

Figura 5.3: Gráficos do erro quadrático total BP e AIP

Para melhorar a performance, pode-se compilar o programa com otimização *-Obench*, vide capítulo 2. Isto foi feito, e os resultados com o uso deste recurso podem ser conferidos na figura 5.4. Não foi mostrado o gráfico do erro quadrático pois é o mesmo da figura 5.3, uma vez que os pesos iniciais são os mesmos.

É importante ressaltar que o tempo de execução pode mudar a cada execução, uma vez que a performance instantânea de uma máquina depende de diversos fatores, tais como quantidade de processos executados pelo sistema, quantidade de memória disponível, etc. A performance também varia de máquina para máquina. Além disso, os valores dos pesos



(a) Método BP (b) Método AIP

Figura 5.4: Resultados da RNA treinada pelos métodos BP e AIP com otimização

iniciais também influenciam no desempenho do programa. Se eles são mudados, o erro inicial muda, assim como todo o comportamento do treinamento.

No resultado apresentado, os pesos iniciais são os mesmos para ambos os métodos. Através destas estatísticas, é possível ver que ambos os métodos de treinamento atingiram o principal objetivo: Classificar os dados com boa precisão.

Mesmo o método BP necessitando de mais ciclos, ele gastou menos tempo do que o método AIP. Porém este último gastou menos ciclos para finalizar o treinamento. Assim, é possível concluir que o ajuste dos pesos por ciclo no método AIP é maior do que no outro método. Porém, como já era esperado, seu custo computacional também é maior, já que gasta muito mais tempo para completar um ciclo de treinamento.

Isto implica uma maior influência da taxa de aprendizagem no comportamento do treinamento. Em certas situações, uma taxa de aprendizagem muito alta faz com que o treinamento pelo método AIP fracasse, enquanto ele é completado pelo outro método. Esta é a razão pela qual uma taxa de aprendizagem tão pequena foi utilizada.

Assim como o método BP, o método AIP também pode ficar preso em mínimos locais. No entanto, técnicas de minimização, como algoritmos genéticos, podem ser utilizadas com sucesso para contornar este problema.

Outro fato importante foi o ganho de performance com a otimização *-Obench*. Os

programas ficaram aproximadamente 3.84 vezes mais rápidos para ambos os métodos, o que demonstra a eficiência da otimização da linguagem Bigloo.

O AIP é um método que permite o ajuste independente dos pesos da RNA, isto é, para se ajustar um peso não é necessário o resultado de nenhum outro. Assim, além da otimização com *-Obench*, é possível ganhar mais performance utilizando uma abordagem diferenciada: Ajustar os pesos tão simultaneamente quanto possível. Será demonstrado também que, quanto maior a RNA, maior a melhora da eficiência através dessa abordagem diferenciada. No próximo capítulo será detalhado como obter sucesso na utilização da computação paralela.

Capítulo 6

Computação Paralela

Computação paralela é uma forma de computação em que há processamento de dados, instruções ou tarefas distintas simultaneamente, por diferentes núcleos de processamento [ALMASI e GOTTLIEB 1989]. É operado sob o princípio de que grandes problemas geralmente podem ser divididos em problemas menores, que então são resolvidos concorrentemente, ou seja, em paralelo [ADVE et al. 2008].

Entre 1980 e 2004, o aumento da frequência foi o principal recurso para melhorar o desempenho dos computadores. Com uma maior frequência de processamento, gasta-se menos tempo para processar cada instrução, aumentando assim a performance do computador [HENNESSY e PATTERSON 2002]. Devido a diminuição da razão entre o aumento da performance e o custo, além da crescente preocupação do consumo de energia por parte dos computadores, máquinas dotadas de múltiplos núcleos de processamento tornaram-se dominantes no mercado a partir de 2004. Com isso, a computação paralela passou a ser cada vez mais popular.

O ganho de performance depende muito da estratégia utilizada e da porcentagem do programa que é paralelizável. Quanto maior for a parte do programa possível de ser paralelizada, maior o ganho de performance por este tipo de estratégia [AMDAHL 1967] e [GUSTAFSON 1988]. Porém, a computação paralela também apresenta desvantagens. Para usar múltiplos núcleos de processamento ou computadores simultaneamente existem os custos para criar e manter as entidades requeridas, além de realizar a comunicação

entre elas. Dependendo da situação pode não compensar pagar este custo.

Deve ser levado em consideração que, para permitir o processamento simultâneo de diversos fragmentos, devem ser criadas entidades, que serão estudadas posteriormente. Para manter estas unidades é preciso memória. Além disso, para que informações sejam trocadas entre estas entidades há o custo da comunicação [GRAMA et al. 2003].

Além dos custos computacionais, programas que utilizam o recurso da computação paralela são mais difíceis de programar que programas sequenciais. Isto acontece, pois a concorrência introduz diversas novas variáveis ao contexto, como a condição de corrida, além de potenciais defeitos, como deadlocks[1] e starvation[2] [PATTERSON e HENNESSY 1998].

Neste trabalho não foram feitas investigações minuciosas a respeito do paralelismo, já que este não é o foco deste trabalho. A computação paralela será tratada somente como uma ferramenta, que possibilitará aprimorar o treinamento de redes neurais artificiais. Testes serão utilizados como parâmetros para medir quando há mais vantagens ou desvantagens no uso do paralelismo.

6.1 Condições Para Uso de Computação Paralela

Além da arquitetura do hardware, também é necessário analisar o programa em que se deseja utilizar a computação paralela. Segundo [BERNSTEIN 1966], quando os dois fragmentos de um programa satisfazem certas condições, chamadas de condições de Bernstein, eles são independentes e podem ser executados em paralelo. De maneira simplificada, as condições são as seguintes:

- Independência de fluxo: Um fragmento não pode produzir um resultado que será utilizado pelo outro;
- Independência de variáveis: Um fragmento não pode utilizar ou sobrescrever uma variável que será utilizada por outro fragmento;

[1] Situação desastrosa, onde um processo de um programa precisa de um determinado recurso que está com outro, e este outro precisa do recurso do primeiro para continuar, travando assim a execução.

[2] Quando um processo nunca é executado, pois processos de prioridade maior sempre o impedem de ser executado.

- Independência de saída: As saídas de diferentes fragmentos devem ser independentes. Porém, quando duas saídas escrevem em um mesmo local, a saída final deve vir do segundo fragmento.

Existem diversos tipos de paralelismo, mas o paralelismo a nível de dado é o mais utilizado para aplicações científicas e de engenharia. Ele é inerente a laços de repetição, focando em distribuir o dado por diferentes nós computacionais para serem processados em paralelo [CULLER et al. 1999].

6.2 Problematização: Identificação de Números Primos

Para ilustrar o conceito de paralelismo, é proposto o seguinte problema: Encontrar todos os números primos entre 2 e um determinado número.

Os números primos, e as suas propriedades, foram pela primeira vez estudados extensivamente pelos antigos matemáticos Gregos. Entende-se por número primo, todo número em que somente o resto da divisão dele por 1(um) ou por ele mesmo é igual a 0(zero). Um algoritmo bem simples para saber se um número é primo é o seguinte:

```
Defina q=2
Loop
  Se q < valor, faça
    Se resto da divisão entre valor e q for igual a 0, faça
      O número não é primo, retorne falso
    Caso contrário, faça
      q=q+1
      Retorne para Loop
  Caso contrário, faça
    O número é primo, retorne verdadeiro
Fim Loop
```

O código 6.1 é um exemplo de função baseada no algoritmo acima, em Scheme Bigloo.

Código 6.1: Verificando se um número é primo

```
(define (primo? n)
   (let loop[(q 2)]
      (cond
         [(< q n)
            (cond
               [(equal? (remainder n q) 0)
                  #f]
               [else
                  (loop (+ q 1))]
            )]
         [else
            #t])))
```

```
1:=> (primo? 48)
#f
1:=> (primo? 79)
#t
```

Sabendo identificar quais números são primos, basta construir uma outra função que executa *primos?* para cada valor em um intervalo, colocando em uma lista os números identificados como primos. Um algoritmo para executar esta tarefa é o seguinte:

```
Dados: nini = limite inicial
       nfin = limite final
Defina i=nini
       res = lista de resultado
Loop
  Se i <= nfin, faça
    Se (primo? i)
      Coloque i dentro de res
    i=i+1
    Retorne para Loop
Fim Loop
Retorne a lista res
```

O código 6.2 é um exemplo de função baseada no algoritmo acima, em Scheme Bigloo.

Código 6.2: Identificando todos os números primos em um intervalo

```
(define (primos nini nfin)
   (let loop[(i nini) (res (list))]
      (cond
         [(<= i nfin)
            (cond
               [(primo? i)
                  (loop (+ i 1) (append res (list i)))]
               [else
                  (loop (+ i 1) res)]
            )]
         [else
            res])))
```

```
1:=> (primos 2 100)
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97)
```

Existe um outro algoritmo para encontrar números primos, chamado Crivo de Eratóstenes. Seu desempenho foi posto em prova, mas ele se mostrou mais lento do que o método sequencial simples. Isto se deve ao fato de ser necessário utilizar estruturas mais complexas, como varredura de listas, o que pode ter piorado sua performance. Mesmo assim ele será abordado no Anexo B. Os resultados dos testes desta comparação seguem a seguir:

```
Teste 1:
  - Otimização utilizada: Nenhuma

Valor: 10000
Crivo: 1.31 segundos.
1229 números primos identificados.
Sequencial: 0.64 segundos.
1229 números primos identificados.
Sequencial 2.04 vezes melhor.

Valor: 50000
Crivo: 51.87 segundos.
5133 números primos identificados.
Sequencial: 13.69 segundos.
5133 números primos identificados.
Sequencial 3.78 vezes melhor.
```

```
Teste 2:
  - Otimização utilizada: -Obench

Valor: 10000
Crivo: 1.06 segundos.
1229 números primos identificados.
Sequencial: 0.21 segundos.
1229 números primos identificados.
Sequencial 5.04 vezes melhor.

Valor: 50000
Crivo: 36.49 segundos.
5133 números primos identificados.
Sequencial: 4.15 segundos.
5133 números primos identificados.
Sequencial 8.79 vezes melhor.
```

O método Crivo de Eratóstenes teve um pior desempenho em todas as situações, pio-

rando ainda mais quando foi utilizada a otimização -Obench. Assim, o método sequencial simples será a referência para futuras comparações, além de ser o método estudado para uso da computação paralela. Isto se deve também ao fato de que o método sequencial é mais simples para se paralelizar.

O problema de encontrar todos os números primos em um intervalo, de maneira sequencial, já foi resolvido. Porém, é possível deixar a solução mais rápida, utilizando o recurso abordado anteriormente neste capítulo: a computação paralela.

6.3 Estratégia para Paralelização

Para o programa proposto na seção 6.2, é possível fazer uso da computação paralela, a nível de dado, de maneira bem simples. Basta dividir um intervalo grande em intervalos menores, já que um número não depende de outros do intervalo para ser identificado como primo. Dada uma quantidade de intervalos desejada e um intervalo, o seguinte algoritmo pode ser utilizado para fornecer intervalos menores.

```
Dados: n = limite final
       nthd = quantidade de intervalos
Defina int= (real->inteiro ((n - 2) / nthd) + 1)
       i = 2
       res = lista de resultado
Loop
  Se i <= (n - int), faça
    Coloque a lista (i (i + int - 1)) dentro da lista res
    i=i+int
    Retorne para Loop
  Caso contrário, faça
    Coloque a lista (i n) dentro da lista res
Fim Loop
Retorne a lista res
```

O código 6.3 é um exemplo de função baseada no algoritmo acima, em Scheme Bigloo.

Código 6.3: Dividindo um intervalo em intervalos menores

```
(define (limites n nthd)
   (define int (+ (/ (- n 2) nthd) 1))
   (cond
      [(flonum? int)
         (set! int (flonum->fixnum int))]
   )
   (let loop[(i 2) (res (list))]
      (cond
         [(<= i (- n int))
            (loop (+ i int) (append res (list (list i (- (+ i int) 1)))))]
         [else
            (append res (list (list i n)))])))
```

```
1:=> (limites 100 8)
((2 14) (15 27) (28 40) (41 53) (54 66) (67 79) (80 92) (93 100))
```

Outras estratégias mais complexas foram experimentadas, porém aumentavam a parcela sequencial do programa, não compensando na performance final.

No código final, vide anexo B, *nthd* é uma variável global, que representa o número de fragmentos em que será dividido o processamento. Com toda a lógica dominada, é hora de conhecer as ferramentas que permitirão a confecção de um programa que utilize computação paralela.

6.4 Modelos de Programação Paralela

Linguagens de programação, bibliotecas, API e modelos foram criados para programar computadores paralelos. Diversas abordagens foram estudadas, dentre elas: Multiplexação, multi-processos, e multi-threads [DAVIS et al. 2004]. Analisando as abordagens estudadas e a linguagem adotada, concluiu-se que a abordagem mais interessante e vantajosa é a multi-thread.

Na programação utilizando threads[3], é utilizada a concorrência com o uso de memória compartilhada. As threads se comportam de acordo com uma estratégia de programação,

[3]Processos "leves", caracterizados por compartilhar a memória do processo criador e de alternarem entre si na posse do recurso de maneira muito rápida.

que pode ser preemptiva ou cooperativa [SERRANO et al. 2004]. Ambas estratégias serão detalhadas a seguir:

Preemptive threads: A estratégia preemptiva permite que um segmento em execução seja retirado a qualquer momento para outra thread ser alocada ao processador. Em seguida, ele pode retomar a thread retirada a qualquer momento. Este tipo de abordagem possui vantagens, dentre as quais podemos ressaltar:

- Não é possível um processo ficar com o recurso para sempre;
- Programas usando esta estratégia podem ser usados em máquinas com múltiplos processadores sem nenhuma mudança.

Alguns problemas relacionados a este tipo de estratégia podem ser encontrados, dentre eles:

- Caso haja necessidade de comunicação ou sincronização, há necessidade de proteção dos dados envolvidos. Isto gera custo de processamento e possibilidade de erros, introduzindo o problema dos deadlocks;
- Altamente dependente da estratégia de preempção;
- Dificuldade muito grande na portabilidade, uma vez que é muito difícil sistemas se comportarem de maneira igual em diferentes plataformas.

Uma importante API deste modelo é a POSIX threads, também chamado de pthread, que define um padrão para criação e manipulação de threads. Dependendo da complexidade do programa, pode ser difícil implementar, já que todo o controle de condições de corrida, sincronização, entrada e saída, entre outros, é de total responsabilidade do programador.

Podem aparecer defeitos graves, como deadlocks e starvation. Mesmo disponível para uso no Bigloo, existe um modelo mais interessante do que as POSIX threads.

Cooperative threads: Uma estratégia cooperativa que não retira as threads de execução. Ela espera até que a thread coopere, abdicando do processador, para alocar

outra thread ao recurso. Muito viável para sistemas que necessitam de sincronização e comunicação, uma vez que os dados não necessitam de ser protegidos para tal.

Este tipo de estratégia facilita a programação, o controle e a portabilidade. Entretanto, possui algumas desvantagens:

- Impossibilidade de se beneficiar de uma arquitetura com múltiplos processadores;
- São necessários meios específicos de lidar com processos capazes de bloquear a execução;
- Possibilidade de deadlocks e de um processo ficar no processador para sempre, caso ocorra um loop infinito.

Além destas duas estratégias clássicas de programação com threads, existe uma terceira: A Fair threads [SERRANO et al. 2004]. Apresentada em 2004, esta estratégia está implementada e pronta para se utilizar no Bigloo[4]. Sua descrição é apresentada a seguir:

Fair threads: É um modelo de programação utilizando threads que diferencia das demais apresentadas pela sua estratégia simples e poderosa:

- Divide as threads em threads do usuário e de serviço. A primeira é abordada de acordo com a estratégia cooperativa, enquanto a outra usa estratégia preemptiva. A responsabilidade sobre as threads de serviço é do sistema operacional;
- Define um instante usado por todas as threads de usuário, independente de sua natureza. Ao final deste instante todas as threads são sincronizadas, fazendo com que sejam executadas no mesmo ritmo;
- Introduz os sinais de usuário, uma função que permite às threads uma poderosa forma de sincronização e comunicação, além de permitir a transformação de uma thread de usuário em uma de serviço para um determinado trecho de código;

[4]Testado em Linux Ubuntu 10.04

- A combinação de threads preemptivas e cooperativas confere a este modelo o aproveitamento de arquiteturas com múltiplos processadores.

Fair-threads é um método muito interessante, pois alia a simplicidade das threads cooperativas ao poder das threads preemptivas. Utilizando apenas uma função, é possível fazer uma thread cooperativa executar uma tarefa de maneira concorrente, e quando ela terminar, volta a ser cooperativa.

Através dos sinais é possível fazer uma thread esperar, dar lugar as outras ou executar quando for desejado, além de permitir a comunicação entre duas ou mais threads.

Pelo que foi estudado, o modelo fair-threads é o melhor para realizar a programação concorrente. Com a estratégia definida, serão detalhadas as ferramentas que permitirão a criação e manipulação das Fair Threads em Bigloo.

Antes de começar, é importante salientar que deve ser carregada a biblioteca *fthread*, para que seja possível a utilização desta estratégia de paralelismo. Abaixo segue a descrição das funções que serão utilizadas para implementar o paralelismo.

instantiate::fthread Cria uma fair thread, que executará uma função específica;

thread-start! Libera uma thread para ser escalonada pelo endereçador, isto é, que execute sua tarefa;

scheduler-start! Solicita ao endereçador que comece a escalonar as threads. A partir do momento que esta função é utilizada, somente threads serão executadas;

thread-await-values! Bloqueia a execução da thread até que o sinal seja recebido. Retorna, no instante seguinte do endereçador, uma lista com todos os valores que foram enviados com aquele sinal;

thread-await*! Bloqueia a execução da thread até que um dos sinais de uma lista seja recebido. Retorna, o conteúdo do sinal recebido;

broadcast! Difunde um sinal identificado para todas as threads iniciadas pelo endereçador;

make-asynchonous-signal Transforma uma thread de usuário em thread de serviço, permitindo a execução de um trecho de código de maneira concorrente, ou seja, em paralelo. Ao final da execução, um sinal é retornado e capturado com *thread-await-values!*, e a thread volta a ser uma thread de usuário.

Ao utilizar Fair Threads, somente threads serão executadas. Então é imprescindível que o programa seja composto exclusivamente por threads, inclusive a função principal.

6.5 Estratégia de Programação Multi-threading

Com as funções a serem utilizadas dominadas, foi desenvolvida uma estratégia para resolver diversos intervalos de números primos de maneira concorrente.

Serão criados três tipos de threads:

- **Thread Principal:** Responsável por ordenar o início do cálculo, emitindo um sinal com a lista de intervalos, de acordo com o número de threads que serão executadas. Após o envio, ela espera o resultado do cálculo;

Código 6.4: Função da Thread Principal

```
1 (define (principal n)
2   (lambda()
3     (broadcast! 'start (limites n))
4     (print "Resultado: " (car (thread-await-values! 'done)))
5     (exit 1)))
```

- **Thread Comandante:** Thread que espera o sinal para iniciar o cálculo da thread principal. Quando ordenada, envia sinais com os respectivos intervalos a cada uma das threads operárias. Em seguida, aguarda o sinal de todas as threads operárias, para montar o resultado e enviar a thread principal.

Código 6.5: Função da Thread Comandante

```
(define (comandante)
   (lambda()
      (let loop[(lista (car (thread-await-values! 'start)))]
         (let loopsend[(i 0)]
            (cond
               [(< i nthd)
                  (broadcast! (string-append "t" (number->string i)) (
                     list-ref lista i))
                  (loopsend (+ i 1))]
            )
         )
         (let looprec[(j 0) (sigs (list-tabulate nthd values)) (res (list))
            ]
            (cond
               [(< j nthd)
                  (multiple-value-bind(info sig)
                     (thread-await*! sigs)
                     (looprec (+ j 1) (delete sig sigs) (append res (caadr
                        info)))
                  )]
               [else
                  (broadcast! 'done res)
                  (loop (car (thread-await-values! 'start)))])))
   )
)
```

- **Thread Operária:** Aguarda o respectivo sinal com o intervalo de cálculo. Após o recebimento, realiza os cálculos de maneira assíncrona, emitindo um sinal para a thread comandante ao término de sua tarefa.

Código 6.6: Função da Thread Operária

```
(define (operaria n)
   (lambda()
      (let loop[(data (car (thread-await-values! (string-append "t" (
         number->string n)))))]
         (broadcast! n (list n (thread-await-values! (
            make-asynchronous-signal (lambda(x) (primos (car data) (cadr
            data))))))))
         (loop (car (thread-await-values! (string-append "t" (number->
            string n)))))
      )))
```

Na figura 6.1, o fluxograma de dados entre as threads é ilustrado.

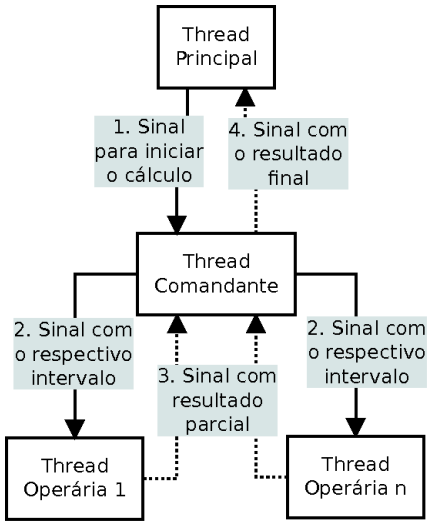


Figura 6.1: Fluxograma para identificação de números primos em paralelo

Analisando as condições de Bernstein para a estratégia proposta, temos:

- Independência de fluxo: Cada intervalo produz resultados independentes, ou seja, nenhum fragmento irá utilizar resultado de outro;
- Independência de variáveis: Cada fragmento utiliza suas próprias variáveis localmente. O resultado de cada intervalo será enviado através de sinais.
- Independência de saída: As saídas não escreverão no mesmo local. Uma thread irá recolher todos os resultados para depois enviar à thread principal.

Satisfeitas as condições de Bernstein, a seguir é descrito o algoritmo para identificar todos os números primos de um dado intervalo, em paralelo:

1. Quando o programa é executado, todas as threads abordadas (principal, comandante e operárias) são iniciadas. A priore, as threads comandante e operárias aguardam seus respectivos sinais. Posteriormente, o endereçador é iniciado, permitindo ao programa executar as threads;

Código 6.7: Funções para criar as threads e iniciar o endereçador

```
(define nthd 15)

(define (thdcriador threads)
   (let loop[(i 0)]
      (cond
         [(< i threads)
            (thread-start! (instantiate::fthread (body (operaria i))))
            (loop (+ i 1))]
      )
   )
)

(define (start args)
   (define n 100) ;Valor superior do intervalo
   (thdcriador nthd)
   (thread-start! (instantiate::fthread (body (comandante))))
   (thread-start! (instantiate::fthread (body (principal n))))
   (scheduler-start!)
)
```

2. A thread principal recebe todo o intervalo de números. Ele reparte em n partes, onde n é o número de threads, e envia um sinal para a thread comandante contendo a lista de intervalos. Após isto, ela aguarda o resultado final da thread comandante;

3. Recebidos todos os intervalos, a thread comandante repassa cada intervalo para a respectiva thread operária. Depois entra em modo de espera, aguardando a resposta de todas as threads operárias;

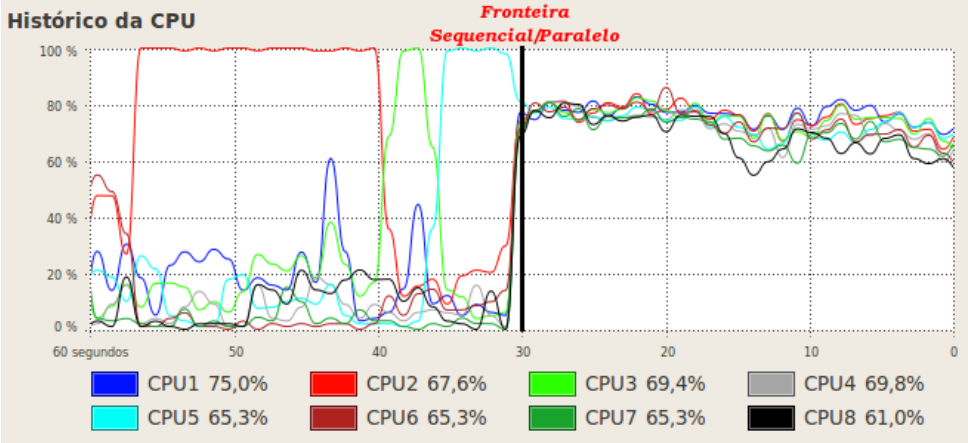
4. De posse do intervalo que cada uma deve trabalhar, as threads operárias executam, de maneira concorrente, sua tarefa. Após concluírem todos os cálculos, cada uma envia um sinal, contendo o resultado de seu trabalho, para a thread comandande;

5. A medida que os resultados parciais vão chegando, a thread comandante vai montando o resultado final. Quando a montagem acaba, ou seja, todas as threads operárias enviaram seu trabalho, um sinal é enviado para a thread principal, contendo o resultado final devidamente montado;

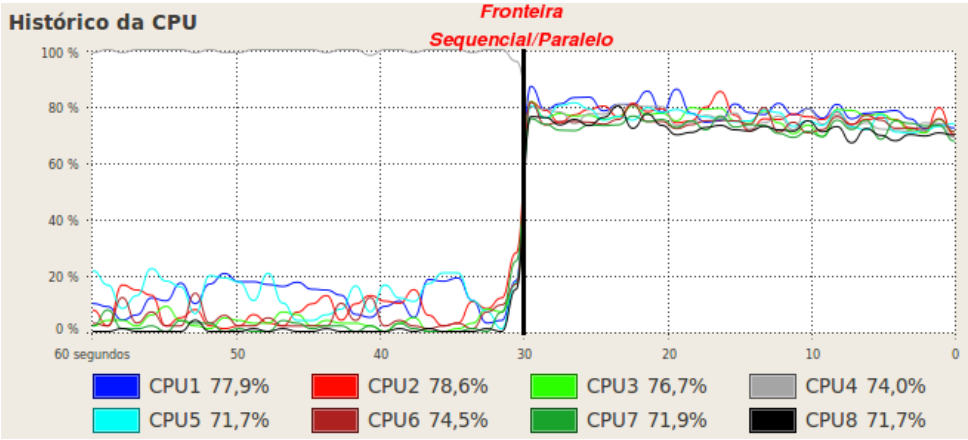
6. O resultado, uma lista com todos os números primos identificados, é mostrado para o usuário quando a thread principal recebe o resultado.

6.6 Resultados da Paralelização

O programa completo está no Anexo B. A respeito da quantidade de threads, foram feitos diversos experimentos. A figura 6.2 ilustra o comportamento sequencial/paralelo com 8 e 15 threads.



(a) 8 threads



(b) 15 threads

Figura 6.2: Comportamentos da CPU Sequencial/Paralelo

A figura 6.2 ilustra o comportamento dos núcleos de processamento na execução de dois programas: um sequencial, ao lado esquerdo da fronteira, e um paralelo, ao lado direito da fronteira. Note que enquanto no programa sequencial somente um processador é utilizado por vez pelo programa, todos os processadores são utilizados simultaneamente no programa paralelo.

Seguem abaixo algumas considerações:

- A quantidade de threads utilizada deve ser, no mínimo, a quantidade de processadores disponíveis. Uma thread operária roda paralelamente em relação às outras, porém sequencialmente dentro da sua tarefa;

- Não é contado no tempo o custo de criação das threads, pois elas são criadas antes da execução da tarefa. Esta estratégia visa melhorar o desempenho do programa em paralelo, evitando processamentos desnecessários no momento da execução;

- Existe um custo de comunicação entre as threads. Então não é recomendado criar threads demasiadamente, pois o custo de comunicação irá atrapalhar na performance da computação paralela;

- Se o tempo de execução das threads concorrentes for praticamente o mesmo, é interessante criar um número de threads igual ao número de processadores, de modo a minimizar o custo de comunicação e maximizar a performance;

- Caso o tempo de execução seja diferente, como é o caso da identificação dos números primos em um intervalo, considere colocar um pouco mais de threads. Algumas vão acabando o cálculo mais cedo, abrindo espaço, que será preenchido por outras que ainda não acabaram o cálculo[5]. É nítido o ganho de performance na figura 6.2 com 15 threads em relação a 8 threads utilizadas.

Os resultados dos testes feitos a seguir são uma média entre os testes realizados. Eles visam comparar a performance entre programa sequencial e o programa que utiliza computação paralela. É importante ressaltar que a melhora na performance varia conforme o programa e a otimização, pois depende da parcela do código que é paralelizável. Confira os resultados a seguir:

[5]Ex: É muito mais rápido verificar se 1000 é primo do que 10000.

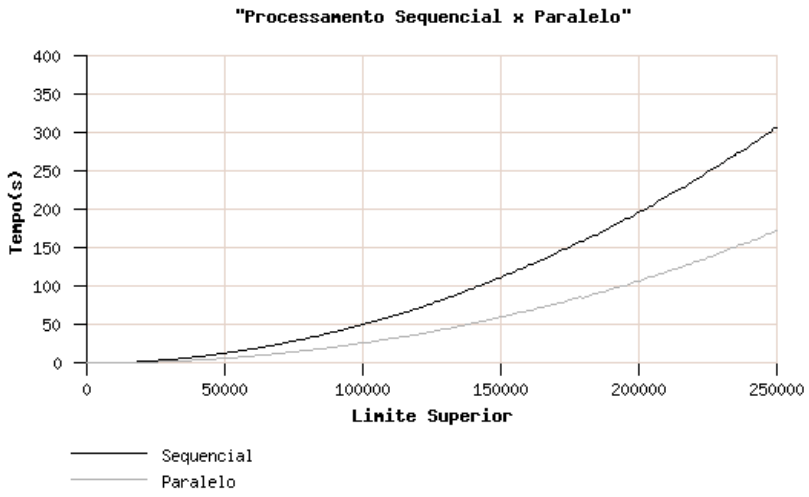
```
Teste 1:
  - Otimização utilizada: Nenhuma

Valor: 10000
Sequencial: 0.65 segundos.
1229 números primos identificados.
Paralelo: 0.32 segundos.
1229 números primos identificados.
Paralelo 2.03 vezes melhor.

Valor: 50000
Sequencial: 13.4 segundos.
5133 números primos identificados.
Paralelo: 6.2 segundos.
5133 números primos identificados.
Paralelo 2.16 vezes melhor.

Valor: 250000
Sequencial: 286.98 segundos.
22044 números primos identificados.
Paralelo: 137.97 segundos.
22044 números primos identificados.
Paralelo 2.08 vezes melhor.
```

```
Teste 2:
  - Otimização utilizada: -Obench

Valor: 10000
Sequencial: 0.21 segundos.
1229 números primos identificados.
Paralelo: 0.11 segundos.
1229 números primos identificados.
Paralelo 1.9 vezes melhor.

Valor: 50000
Sequencial: 3.27 segundos.
5133 números primos identificados.
Paralelo: 1.42 segundos.
5133 números primos identificados.
Paralelo 2.3 vezes melhor.

Valor: 250000
Sequencial: 72.94 segundos.
22044 números primos identificados.
Paralelo: 27.15 segundos.
22044 números primos identificados.
Paralelo 2.68 vezes melhor.
```

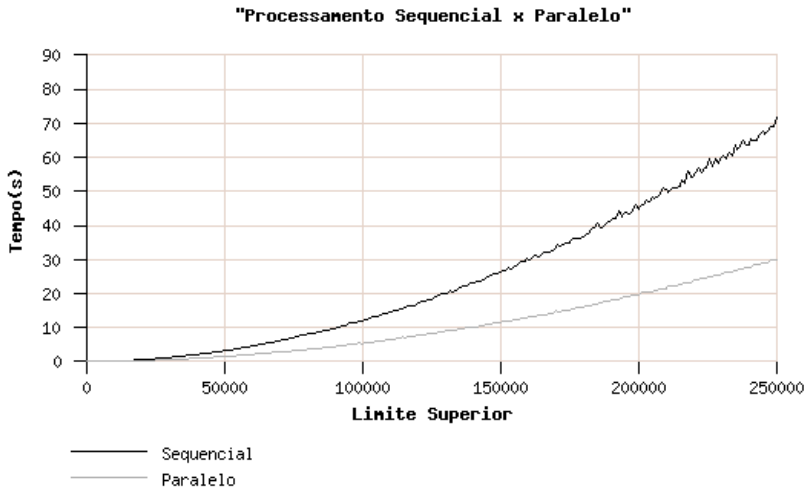
Apresentados os resultados, é visível a melhora da performance de um programa utilizando computação paralela, fazendo deste recurso uma ferramenta bastante poderosa para a diminuir o tempo de execução de programas paralelizáveis. Sem otimização, a performance do programa que utiliza paralelismo foi, em média, duas vezes melhor. Houve uma pequena variação na melhora quando o programa foi compilado utilizando -Obench. Isto deve ter sido causado por uma mudança na relação entre a parte que pode ser paralelizada e a que não pode, devido ao uso da otimização.

A figura 6.3 apresenta gráficos, feitos utilizando o software Ploticus [PLOTICUS 2009]. Neles é ilustrada a comparação entre processamento sequencial e paralelo para identificação de números primos, com limites superiores variando entre 1000 e 250000 números.

Como esta simulação demora muitas horas para ser feita, foi feito um programa para executar todos os intervalos, um por vez, e ao final da execução plotar um gráfico com os resultados. Podem haver algumas diferenças entre os testes feitos e os gráficos. Estes gráficos servem somente como referência para comparar o comportamento de um programa



(a) Sem otimização



(b) Otimização -Obench

Figura 6.3: Gráficos identificação de números primos

sequencial e um paralelo, utilizando diferentes intervalos de busca.

Com os resultados, foi possível observar que a estratégia de programação paralela cumpriu o que se propôs a fazer: Diminuir o tempo de execução de um programa que pode ser paralelizado. A estratégia utilizada foi bem simples e atendeu bem às necessidades. Com tudo esclarecido, no próximo capítulo serão aplicados os conhecimentos adquiridos sobre programação paralela no treinamento de uma RNA pelo método AIP.

Capítulo 7

Paralelização do Método AIP

Neste capítulo, será adotada uma abordagem semelhante a utilizada no capítulo 6 para treinar uma RNA pelo método AIP, vide capítulo 4. O principal objetivo é diminuir o tempo de execução do treinamento.

Antes de tudo é necessário verificar se o método AIP é passível de paralelização. Para isto serão utilizadas as condições de Bernstein. Veja a seguir:

- Independência de fluxo: Cada peso produz resultado independente, ou seja, nenhum fragmento irá utilizar resultado de outro;
- Independência de variáveis: Cada fragmento utilizará suas próprias variáveis localmente. O resultado de cada ajuste de peso será enviado através de sinais.
- Independência de saída: As saídas não escreverão no mesmo local. Uma thread irá recolher todos os resultados para depois enviar à thread principal.

7.1 Threads Utilizadas

A paralelização do treinamento utilizando Fair Threads, contará com três tipos de threads. As características destas novas threads são semelhantes as das threads abordadas anteriormente. Veja a seguir:

- A **thread principal**, responsável por monitorar se o treinamento continua e, se necessário, ordenar um novo ciclo de treinamento;

- Uma **thread comandante**, que espera pelo sinal de treinamento da thread principal. Quando o comando é recebido, ela manda sinais com dados para as threads que estão encarregadas de calcular os ajustes dos pesos. Após isto, a thread comandante espera os resultados, que, uma vez recebidos, são alocados em suas respectivas posições no vetor de ajustes. Em seguida ela envia o resultado para a thread principal;

- **Threads operárias**, que esperam pelos dados atuais da RNA e, paralelamente, realizam o ajuste dos pesos que são de sua responsabilidade.

7.2 Algoritmo de Treinamento

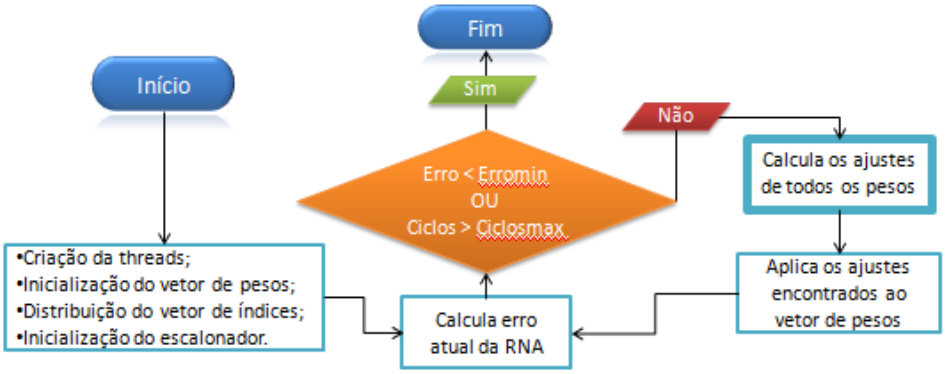
A figura 7.1a ilustra o comportamento do algoritmo de treinamento como um todo. Já a figura 7.1b ilustra como é realizado o intercâmbio de informações pelas threads, na tarefa de ajustar, paralelamente, os pesos sinápticos da RNA.

O algoritmo para treinar a RNA paralelamente ocorrerá como a seguir:

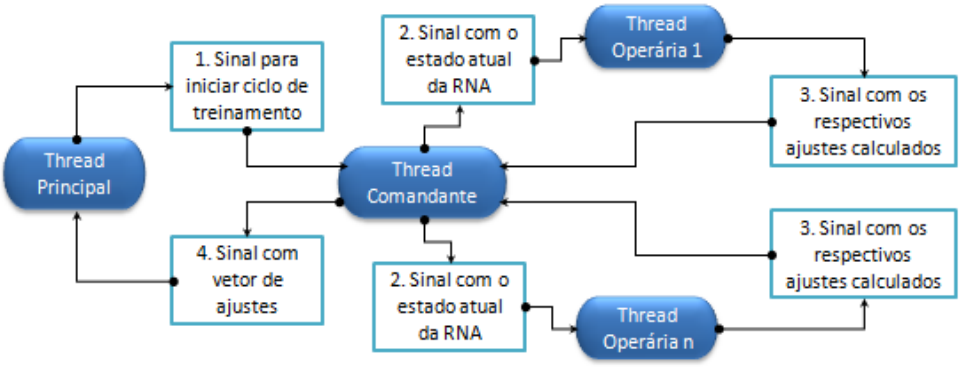
1. Assim que o programa começa sua execução, as threads operária e a thread comandante são criadas. O vetor de índices é distribuído igualmente entre todas as threads operárias. Se a divisão não for exata, a última thread recebe menos índices do que as demais;

2. Na thread principal, o erro quadrático total e os ciclos gastos são checados. Caso o primeiro parâmetro esteja abaixo de um valor estipulado ou o segundo acima, o treinamento acaba. Caso contrário o treinamento irá para o próximo passo;

3. Durante a rotina de treino, a thread principal pode mandar uma ordem de treinamento para a thread comandante. Após isto a thread que ordenou o treinamento espera a chegada de um sinal com o resultado final;



(a) Fluxograma geral do treinamento da RNA



(b) Fluxograma ajuste dos pesos sinápticos paralelamente

Figura 7.1: Fluxogramas do comportamento do treinamento da RNA

4. Após a thread comandante receber uma ordem de treinamento, ela envia sinais com dados sobre a RNA para todas as threads operárias, enquanto espera por todos os resultados parciais;

5. De posse dos dados atuais da RNA, as threads operárias calculam, assincronamente, todos os ajustes de pesos de sua responsabilidade. Após o término dos cálculos, cada uma envia um sinal com o resultado para a thread comandante;

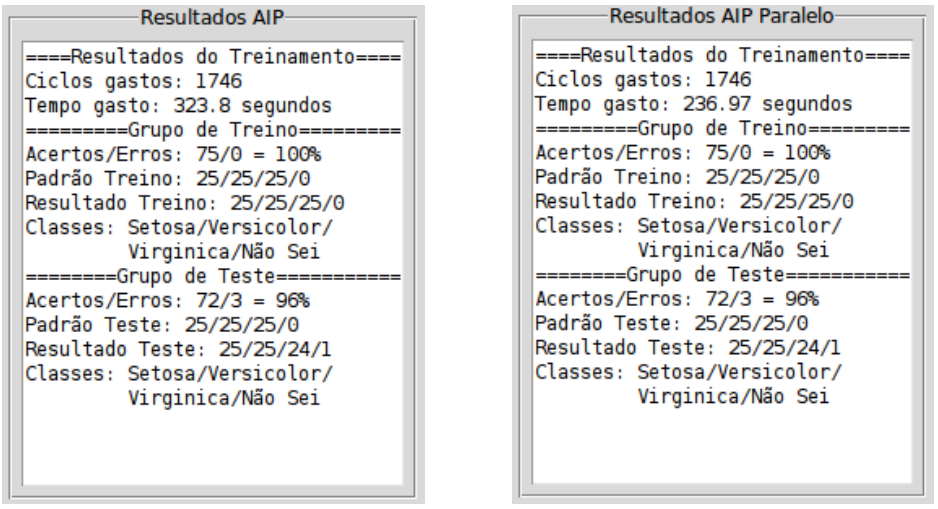
6. A medida que os resultados vão chegando, eles são alocados em suas respectivas posições no vetor de ajustes. Quando o vetor de ajustes estiver completo, isto é, com todos os resultados, um sinal é mandado com o resultado final para a thread principal;

7. Os ajustes são atualizados na thread principal e o algoritmo retorna para o passo 2.

Esta paralelização do método AIP, chamada AIP Paralelo, nada mais é do que uma junção do conhecimento abordado nos capítulos anteriores. O treinamento foi feito pelo método AIP e distribuído como no programa que identifica os números primos em um intervalo.

7.3 Resultados

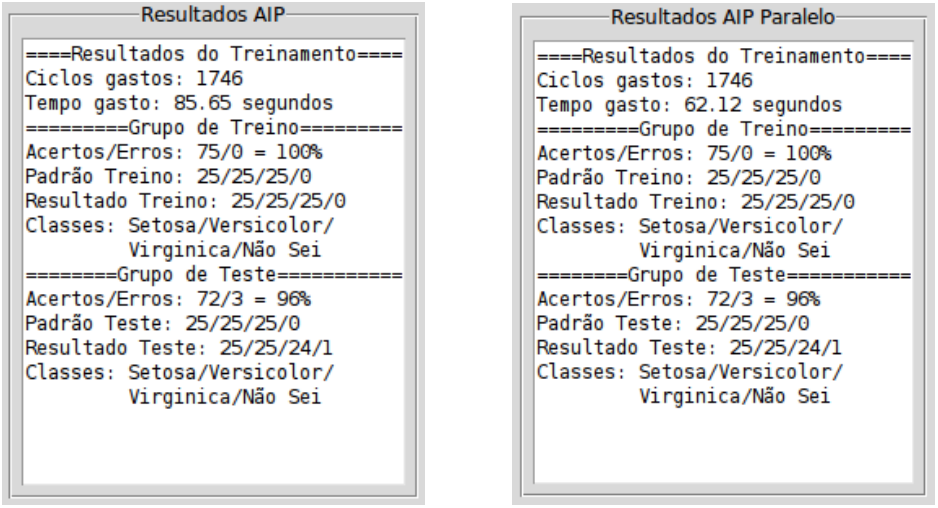
Utilizando o vetor apresentado no código 5.3, foram feitos diversos testes comparando os métodos AIP e AIP Paralelo. Como já era esperado, a única coisa que mudou de um método para o outro foi o tempo de treinamento. O número de ciclos, bem como os dados estatísticos foram iguais. Veja os resultados na figura 7.2.



(a) Método AIP (b) Método AIP Paralelo

Figura 7.2: Resultados da RNA treinada pelo método AIP sequencial/paralelo

Como é possível observar, a performance do treinamento da RNA pelo AIP Paralelo foi aproximadamente 1.36 vezes melhor. Na figura 7.3, estão os resultados do treinamento utilizando otimização *-Obench*.



(a) Método AIP (b) Método AIP Paralelo

Figura 7.3: Resultados da RNA treinada pelo método AIP sequencial/paralelo com otimização

A melhora da performance foi praticamente igual, independente da otimização, por volta de 1.37 vezes. Isto mostra que a parcela do programa que é paralelizável se manteve a mesma, e que a otimização afetou, de maneira proporcional, todo o programa para treinar uma RNA de 13 neurônios.

A tabela 7.1 detalha e compara o resultado de todos os treinamentos realizados, tanto utilizando o método BP como o AIP, além de segregar os resultados de acordo com a utilização da otimização.

Tabela 7.1: Comparação dos resultados de todos os treinamentos realizados

Método	Normal	Otimizado
Backpropagation	53.16s	14.14s
	1x	1x
AIP Sequencial	323.8s	85.05s
	0.164x	0.165x
AIP Paralelo	236.97s	62.12s
	0.224x	0.227x

Comparando os resultados das figuras 5.2a e 7.2b; e das figuras 5.4a e 7.3b, além dos resultados mostrados na tabela 7.1 é possível observar que o método BP ainda tem performance aproximadamente 4.4 vezes melhor do que o método AIP Paralelo. Porém, foi utilizada uma máquina de somente oito núcleos de processamento. Mesmo assim, já é um bom resultado para um método de treinamento mais simples e sem diferenciação.

É de conhecimento do autor que o método BP pode ser paralelizado. Porém o objeto de estudo deste trabalho é a paralelização de um método de treinamento sem topologia. O método error backpropagation é apenas uma referência para comparação da performance do método AIP.

Os resultados comparativos comprovam que o método BP é mais rápido. Isto acontece pelo fato de ele ser mais refinado. Porém, de acordo com a bibliografia, ele depende de uma topologia em camadas. Já o método AIP independe de topologia, além de poder aproveitar melhor arquiteturas com múltiplos núcleos de processamento.

Capítulo 8

Conclusão

Redes Neurais Artificiais são técnicas computacionais que representam o modelo matemático inspirado nas estruturas neurais de organismos inteligentes, e que adquirem conhecimento através de exemplos e experiência. Estas técnicas possuem diversas aplicações, como discutido no capítulo 1.

Este trabalho apresentou uma comparação entre o método mais popular de treinamento de RNAs, o BP, e o método abordado neste trabalho, o AIP. Com base nos resultados, conclui-se que ambos alcançaram o principal objetivo do trabalho: Classificar e reconhecer os dados da porta lógica XOR e da base de dados Iris com boa precisão.

Além disso, foi proposta uma estratégia para melhorar a performance do método AIP: a computação paralela. De maneira prática, eficiente e intuitiva foi abordada e implementada esta estratégia. Todos os requisitos e ferramentas para utilização de programação concorrente foram abordadas, bem como suas vantagens e desvantagens. Após isto, foi feita uma aplicação simples, de modo a ilustrar o comportamento da computação paralela, mostrando quando e como ela pode ser eficiente.

Depois foi a feita a incorporação da computação paralela ao treinamento de RNAs pelo método AIP. O resultado foi um sucesso, levando a uma melhora de aproximadamente 1.36 vezes em relação ao código sequencial.

Foram feitos testes com diferentes configurações de RNA, mudando o número de neurônios. De acordo com a lei de Amdahl, a eficiência da computação paralela depende

da parcela do programa que é paralelizável. Também foram discutidas as desvantagens da computação paralela, como o custo na criação, manutenção e comunicação das threads.

Uma conclusão lógica é a seguinte: No treinamento de uma RNA pelo método AIP, a parcela do programa paralelizável é diretamente proporcional a quantidade de neurônios da RNA. Isto se deve ao fato de que quanto mais pesos devem ser ajustados, maior é o custo computacional. O restante do código, que compreende a atualização dos pesos e a verificação da continuidade do treinamento, variam da mesma forma, porém com uma magnitude muito menor. Isto pode ser deduzido com bastante facilidade, visto que o custo computacional para preencher um vetor e para calcular o erro quadrático uma única vez varia com uma intensidade muito menor do que o restante do treinamento.

Testes comprovaram que quando maior a RNA, ou seja, quanto mais pesos sinápticos ela possuir, maior é o ganho de eficiência da computação paralela. Também foi testado a abordagem multithreading em RNAs muito pequenas, como a do capítulo 4. Neste caso, houve uma perda de performance interessante, onde ficaram evidentes os custos da comunicação e manutenção das threads[1].

Um fator interessante do método AIP é sua independência de topologia. Segundo a bibliografia, para ser mais preciso [JABRI e FLOWER 1992], este método de treinamento pode ser utilizado também para redes recorrentes. Um bom trabalho futuro nesta área seria um estudo comparando redes neurais com e sem topologia, visando estabelecer uma relação na quantidade de neurônio entre as duas. Talvez seja possível obter o mesmo efeito em duas redes neurais distintas, e se a RNA sem topologia possuir um número menor de pesos, seu treinamento pelo método AIP teria uma performance melhor.

Atividades futuras com intenção de dar continuidade a este trabalho seriam, além da comparação entre RNAs com e sem topologia, a aplicação do que foi produzido neste trabalho em computadores mais potentes, dotados de chips potentes, como o Knights Corner[2], da Intel®. Uma alternativa aos computadores mais potentes seria o uso de

[1]Não foram abordados os custos de criação pois as threads são criadas antes do treinamento ser iniciado.

[2]Possui mais de 50(cinquenta) núcleos de processamento e velocidade de 1(um) teraflop, equivalente a 1(um) trilhão de cálculos por segundo. Fonte: BBC: http://www.bbc.co.uk/news/technology-15758057.

clusteres[3].

Outras heurísticas com o objetivo de diminuir o treinamento da RNA, além do uso de programação genética para encontrar a melhor arquitetura de RNA também são apontados como possíveis trabalhos futuros. Como programação não é um trabalho absoluto, um estudo para melhorar o código do treinamento, de modo a aumentar a performance e a parcela paralelizável do treinamento também são propostas válidas.

Ao final deste trabalho conclui-se que, apesar de não ter sido alcançada uma performance superior pelo método AIP sobre o método BP, o resultado foi um sucesso. Um método para treinar RNAs sem topologia em camadas e menos matemático foi abordado.

[3]Um conjunto de computadores, que são ligados em rede e trabalham como se fossem uma única máquina de grande porte.

Referências Bibliográficas

[AMDAHL 1967] AMDAHL, G. (1967). Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. AFIPS Conference Proceedings (30): pp. 483-485.

[ALMASI e GOTTLIEB 1989] ALMASI, G. S. e GOTTLIEB A. (1989). Highly Parallel Computing. Benjamin-Cummings, Redwood City, CA.

[ADVE et al. 2008] ADVE S. V., ADVE V. S., AGHA G., FRANK, M. I., GARZARÁN, M. J., HART J. C., HWU W. W., JOHNSON R. E., KALE L. V., KUMAR R., MARINOV D., NAHRSTEDT. K., PADUA. D., PARTHASARATHY M., PATEL. S. J., ROSU G., ROTH D., SNIR M., TORRELLAS J. e ZILLES C. (2008). Parallel Computing Research at Illinois: The UPCRC Agenda. Parallel@Illinois, Universidade de Illinois, Urbana, IL.

[BERNSTEIN 1966] BERNSTEIN, A. J. (1966). Program Analysis for Parallel Processing, IEEE Trans. em Electronic Computers. EC-15, p. 757-762.

[BIGLOO 2011] SERRANO, M. (2011). Bigloo, a practical Scheme compiler, user manual for version 3.7a. Disponível em: http://www-sop.inria.fr/mimosa/fp/Bigloo/doc/bigloo.html. Inria Sophia-Antipolis. Acessado em: 12/11/2011.

[CAUSON e MINGHAM 2010] CAUSON D. M. e MINGHAM C. G. (2010). Introductory Finite DIfference Methods for PDEs. Ventus Publishings ApS. [ISBN: 978-87-7681-642-1]. pp. 17-33.

[CULLER et al. 1999] CULLER, D. E., SINGH J. P. e GUPTA A. (1999). Parallel Computer Architecture - A Hardware/Software Approach. Morgan Kaufmann Publishers. [ISBN: 1-55860-343-3]. pp. 124-125.

[DAVIS et al. 2004] DAVIS K., TURNER J. W. e YOCUM N. (2004). The Definitive Guide to Linux Network Programming (Expert's Voice). [ISBN-10: 1590593227], [ISBN-13: 978-1590593226] 1ª ed, Apress, 400p.

[FAUSSETT 1993] FAUSSETT L. (1993). Fundamentals of Neural Networks: Architectures, Algorithms And Applications. [ISBN-10: 0133341860], [ISBN-13: 978-0133341867] 1ª ed, Prentice Hall 461p.

[FRANK e ASUNCION 2010] FRANK, A. e ASUNCION, A. (2010). UCI Machine Learning Repository, Iris Data Set. Irvine, CA: University of California, School of Information and Computer Science. Disponível em: http://archive.ics.uci.edu/ml/datasets/Iris. Acessado em: 14/11/2011.

[GRAMA et al. 2003] GRAMA, A., GUPTA, A., KARYPIS, G. e KUMAR, V. (2003). Introduction to Parallel Computing. [ISBN: 0-201-64865-2] 2ª ed, Addison-Wesley, 856p.

[GUSTAFSON 1988] GUSTAFSON J. L. (1988). Reevaluating Amdahl's Law. Communications of the ACM 31(5), pp. 532-33.

[HAYKIN 1999] HAYKIN, S. (1999). Neural networks. A comprehensive foundation. 2 Ed. Prentice Hall. New Jersey. 823p.

[HEBB 1949] HEBB, D. O. (1949). The Organization of Behavior. New York : John-Wiley & Sons Inc, 1949.

[HENNESSY e PATTERSON 2002] HENNESY, J. L. e PATTERSON D. A. (2002). Computer Architecture: A Quantitative Approach. 3ª ed., Morgan Kaufmann, p. 43.

[HOYTE 2008] HOYTE, D. (2008). Let over lambda. 50 years of Lisp. HCSW and Hoytech. La Vergne. p. 17-37. [ISBN: 978-1-4357-1275-1].

[JABRI e FLOWER 1992] JABRI M. e FLOWER B. (1992). Weight perturbation: an optimal architecture and learning technique for analog VLSI feedforward and recurrent multi-layer networks. IEEE Transactions on Neural Networks, Vol. 3, No. 1, pp. 154-157, 1992.

[PATTERSON e HENNESSY 1998] PATTERSON, D. A. e HENNESSY J. L. (1998). Computer Organization and Design, Morgan Kaufmann Publishers, p. 715. [ISBN: 1558604286].

[PLOTICUS 2009] GRUBB S. (2009). Ploticus, a free, GPL, non-interactive software package for producing plots, charts, and graphics from data. Versão 2.41. Disponível em: http://www.ploticus.sourceforge.net. Acessado em: 12/11/2011.

[RAMON y CAJAL 1906] RAMON y CAJAL S. (1906). The structure and connexions of neurons. From Nobel Lectures, Physiology or Medicine 1901-1921, Elsevier Publishing Company, Amsterdam, 1967.

[ROSENBLATT 1958] ROSENBLATT F. (1958). The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain, Cornell Aeronautical Laboratory, Psychological Review, v65, No. 6, pp. 386-408. [doi:10.1037/h0042519].

[RUMELHART et al. 1986] RUMELHART D. E., HINTONT G. E. e WILLIANS R. J. (1986). Learning representations by back-propagating errors. Nature 323, 533 - 536 (09 de Outubro de 1986); [doi:10.1038/323533a0].

[SERRANO et al. 2004] SERRANO, M., BOUSSINOT, F. e SERPETTE B. (2004). Scheme fair threads. Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming, p.203-214, 24-26 de Agosto, 2004, Verona, Itália [doi:10.1145/1013963.1013986].

[WIDROW e LEHR 1990] WIDROW B. e LEHR, M. A. (1990). 30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation. Proceedings of the IEEE, vol.78, no.9, pp.1415-1442, Set 1990. [doi: 10.1109/5.58323].

[WIDROW e HOFF 1960] WIDROW B. e HOFF Jr. M. E. (1960). Adaptive Switching Circuits. IRE WESCON Convention Record, 4:96-104, Agosto de 1960.

Anexo A

Treinamento de uma RNA para Representar a Porta Lógica XOR

No capítulo 4, foi apresentado o método AIP. Este método treina, utilizando

Abaixo segue o código integral do capítulo em questão.

Código A.1: Programa completo para treinar uma RNA para representar a porta lógica XOR utilizando o método AIP.

```
;Programa para treinar uma RNA que aprenda a representar a porta lógica XOR
;Autor: Will Roger Pereira
;Programado em Bigloo 3.6a

(module xor
   (main start)
)

(define (start args)
   (define inputs (list (list 0 0) (list 0 1) (list 1 0) (list 1 1)))
   (define targets (list (list 0) (list 1) (list 1) (list 0)))
   (define weights (make-vector 7 0))
   (define xor (gate weights))
   (setweights (xor '()))
   (multiple-value-bind (res rtime)
      (time (lambda() (train xor inputs targets 5000 0.01 0.9 0.1)))
      (print "Tempo gasto: " rtime " milisegundos.")
   )
   (print "0 0 = " (xor (list-ref inputs 0)))
   (print "0 1 = " (xor (list-ref inputs 1)))
   (print "1 0 = " (xor (list-ref inputs 2)))
   (print "1 1 = " (xor (list-ref inputs 3)))
)

(define (sigma x)
```

```
26      (/ 1.0 (+ 1.0 (exp (- x))))
27  )
28
29  (define (setweights v)
30      (define n (vector-length v))
31      (let loop[(i 0)]
32          (cond
33              [(< i n)
34                  (vector-set! v i (- 0.5 (/ (random 1001) 1000)))
35                  (loop (+ i 1))])))
36
37  (define (newneuron weights indexes)
38      (lambda(inputs)
39          (let loop[(i indexes) (x (cons 1.0 inputs)) (acc 0.0)]
40              (cond
41                  [(or (null? i) (null? x))
42                    (sigma acc)]
43                  [else
44                      (loop (cdr i) (cdr x) (+ acc (* (vector-ref weights (car i))
                                (car x))))])))
45
46  (define (gate weights)
47      (let [(ne (newneuron weights '(0 1 2)))
48            (ns (newneuron weights '(3 4 5 6)))]
49          (lambda(in)
50              (cond
51                  [(null? in)
52                      weights]
53                  [else
54                      (let [(outc1 (list (ne in)))]
55                          (list
56                              (ns (append in outc1))
57                          )
58                      )])))
59
60  (define (errsum targets reals)
61      (define err 0.0)
62      (map (lambda(x1 x2) (map (lambda(y1 y2) (set! err (+ err (* (- y1 y2) (-
            y1 y2))))) x1 x2)) targets reals)
63      err)
64
65  (define (updateweights weights adjusts)
66      (define tam (vector-length weights))
67      (let loop[(i 0)]
68          (cond
69              [(< i tam)
70                  (vector-set! weights i (+ (vector-ref weights i) (vector-ref
                        adjusts i)))
71                  (loop (+ i 1))])))
72
73  (define (AIPadj i weights erro inputs targets alfa dx)
74      (define tam (vector-length weights))
75      (define newrna (gate (copy-vector weights tam)))
76      (vector-set! (newrna '()) i (+ (vector-ref (newrna '()) i) dx))
77      (let [(newerro (errsum targets (map (lambda(x) (newrna x)) inputs)))]
78          (- (* alfa (/ (- newerro erro) dx)))))
```

```
79
80 (define (AIP adj weights erro inputs targets alfa dx)
81    (define tam (vector-length weights))
82    (let loop[(i 0)]
83       (cond
84          [(< i tam)
85             (vector-set! adj i (AIPadj i weights erro inputs targets alfa
                  dx))
86             (loop (+ i 1))])))
87
88 (define (train rna inputs targets cicmax errmax alfa dx)
89    (define adj (make-vector (vector-length (rna '())) 0))
90    (let loop[(ciclos 0) (erro (errsum targets (map (lambda(x) (rna x))
          inputs)))]
91       (cond
92          [(or (>= ciclos cicmax) (< erro errmax))
93             (print "Treinamento concluído:")
94             (print "Ciclos gastos: " ciclos)
95             (print "Erro quadrático total: " erro)]
96          [else
97             (AIP adj (rna '()) erro inputs targets alfa dx)
98             (updateweights (rna '()) adj)
99             (loop (+ ciclos 1) (errsum targets (map (lambda(x) (rna x))
                  inputs)))])))
```

Anexo B

Identificação de Números Primos

No capítulo 6, foi apresentado como identificar, em um dado intervalo, todos os números primos. Para tanto, foram utilizadas duas abordagens: Uma sequencial e uma paralela.

Abaixo segue o código integral do capítulo em questão.

Código B.1: Programa completo que identifica todos os números primos em um intervalo.

```
1  ;Programa para identificar números primos em um intervalo, utilizando
       abordagem sequencial e paralela.
2  ;Autor: Will Roger Pereira
3  ;Programado em Bigloo 3.6a
4
5  (module primos
6     (library fthread)
7     (main start))
8
9  (define nthd 15)
10
11 (define (start args)
12    (define n 100)
13    (thdcriador nthd)
14    (thread-start! (instantiate::fthread (body (comandante))))
15    (thread-start! (instantiate::fthread (body (principal n))))
16    (scheduler-start!))
17
18 (define (principal n)
19    (lambda()
20       (print "Valor: " n)
21       (multiple-value-bind (res rtime)
22          (time
23             (lambda()
24                (crivo n)
25             )
26          )
27          (print res)
28          (print "Crivo: " (/ rtime 1000) " segundos.")
```

```
29          )
30          (multiple-value-bind (res rtime)
31             (time
32                (lambda()
33                   (primos 2 n)
34                )
35             )
36             (print res)
37             (print "Sequencial: " (/ rtime 1000) " segundos.")
38          )
39          (multiple-value-bind (res rtime)
40             (time
41                (lambda()
42                   (broadcast! 'start (limites n))
43                   (car (thread-await-values! 'done))
44                )
45             )
46             (print res)
47             (print "Paralelo: " (/ rtime 1000) " segundos.")
48          )
49          (exit 1)))
50
51    (define (primo? n)
52       (let loop[(q 2)]
53          (cond
54             [(< q n)
55                (cond
56                   [(equal? (remainder n q) 0)
57                      #f]
58                   [else
59                      (loop (+ q 1))]
60                )]
61             [else
62                #t])))
63
64    (define (primos nini nfin)
65       (let loop[(i nini) (res (list))]
66          (cond
67             [(<= i nfin)
68                (cond
69                   [(primo? i)
70                      (loop (+ i 1) (append res (list i)))]
71                   [else
72                      (loop (+ i 1) res)]
73                )]
74             [else
75                res])))
76
77    (define (limites n)
78       (define int (+ (/ (- n 2) nthd) 1))
79       (cond
80          [(flonum? int)
81             (set! int (flonum->fixnum int))]
82       )
83       (let loop[(i 2) (res (list))]
84          (cond
```

```
85          [(<= i (- n int))
86             (loop (+ i int) (append res (list (list i (- (+ i int) 1)))))]
87          [else
88             (append res (list (list i n)))])))
89
90 (define (thdcriador threads)
91    (let loop[(i 0)]
92       (cond
93          [(< i threads)
94             (thread-start! (instantiate::fthread (body (operaria i))))
95             (loop (+ i 1))])))
96
97 (define (operaria n)
98    (lambda()
99       (let loop[(data (car (thread-await-values! (string-append "t" (
             number->string n)))))]
100         (broadcast! n (list n (thread-await-values! (
             make-asynchronous-signal (lambda(x) (primos (car data) (cadr
             data)))))))
101         (loop (car (thread-await-values! (string-append "t" (number->
             string n)))))
102      )))
103
104 (define (comandante)
105    (lambda()
106       (let loop[(lista (car (thread-await-values! 'start)))]
107          (let loopsend[(i 0)]
108             (cond
109                [(< i nthd)
110                   (broadcast! (string-append "t" (number->string i)) (
                         list-ref lista i))
111                   (loopsend (+ i 1))]
112             )
113          )
114          (let looprec[(j 0) (sigs (list-tabulate nthd values)) (res (list))
                ]
115             (cond
116                [(< j nthd)
117                   (multiple-value-bind(info sig)
118                      (thread-await*! sigs)
119                      (looprec (+ j 1) (delete sig sigs) (append res (caadr
                            info)))
120                   )]
121                [else
122                   (broadcast! 'done res)
123                   (loop (car (thread-await-values! 'start)))]))
124       )))
125
126 (define (maior n)
127    (define res (sqrt n))
128    (cond
129       [(flonum? res)
130          (flonum->fixnum res)]
131       [else
132          res]))
133
```

```
134 (define (remmult p lst)
135    (define i (list-ref lst p))
136    (let loop[(l (drop lst (+ p 1))) (res (take lst (+ p 1)))]
137       (cond
138          [(> (length l) 0)
139             (cond
140                [(= (remainder (car l) i) 0)
141                   (loop (cdr l) res)]
142                [else
143                   (loop (cdr l) (append res (list (car l))))]
144             )]
145          [else
146             res])))
147
148 (define (crivo n)
149    (define limite (maior n))
150    (let loop [(i 0) (l (cddr (list-tabulate (+ n 1) values)))]
151       (cond
152          [(<= (list-ref l i) limite)
153             (loop (+ i 1) (remmult i l))]
154          [else
155             l])))
```

Anexo C

Base de Dados

Nos capítulos 5 e 7, foi apresentada uma aplicação para demonstrar o funcionamento do método AIP. Nesta aplicação, foi utilizada uma base de dados [FRANK e ASUNCION 2010]. Para executar o programa do Anexo D, é necessário colocar esta base de dados dentro de um arquivo de nome "iris.data", na mesma pasta do programa.

Abaixo seguem os dados utilizados no treinamento da RNA.

Código C.1: Base de dados utilizada no treinamento das RNAs.

```
1  4.9 3.0 1.4 0.2 Iris-setosa
2  5.1 3.5 1.4 0.2 Iris-setosa
3  4.7 3.2 1.3 0.2 Iris-setosa
4  4.6 3.1 1.5 0.2 Iris-setosa
5  5.0 3.6 1.4 0.2 Iris-setosa
6  5.4 3.9 1.7 0.4 Iris-setosa
7  4.6 3.4 1.4 0.3 Iris-setosa
8  5.0 3.4 1.5 0.2 Iris-setosa
9  4.4 2.9 1.4 0.2 Iris-setosa
10 4.9 3.1 1.5 0.1 Iris-setosa
11 5.4 3.7 1.5 0.2 Iris-setosa
12 4.8 3.4 1.6 0.2 Iris-setosa
13 4.8 3.0 1.4 0.1 Iris-setosa
14 4.3 3.0 1.1 0.1 Iris-setosa
15 5.8 4.0 1.2 0.2 Iris-setosa
16 5.7 4.4 1.5 0.4 Iris-setosa
17 5.4 3.9 1.3 0.4 Iris-setosa
18 5.1 3.5 1.4 0.3 Iris-setosa
19 5.7 3.8 1.7 0.3 Iris-setosa
20 5.1 3.8 1.5 0.3 Iris-setosa
21 5.4 3.4 1.7 0.2 Iris-setosa
22 5.1 3.7 1.5 0.4 Iris-setosa
23 4.6 3.6 1.0 0.2 Iris-setosa
24 5.1 3.3 1.7 0.5 Iris-setosa
25 4.8 3.4 1.9 0.2 Iris-setosa
```

```
5.0 3.0 1.6 0.2 Iris-setosa
5.0 3.4 1.6 0.4 Iris-setosa
5.2 3.5 1.5 0.2 Iris-setosa
5.2 3.4 1.4 0.2 Iris-setosa
4.7 3.2 1.6 0.2 Iris-setosa
4.8 3.1 1.6 0.2 Iris-setosa
5.4 3.4 1.5 0.4 Iris-setosa
5.2 4.1 1.5 0.1 Iris-setosa
5.5 4.2 1.4 0.2 Iris-setosa
4.9 3.1 1.5 0.2 Iris-setosa
5.0 3.2 1.2 0.2 Iris-setosa
5.5 3.5 1.3 0.2 Iris-setosa
4.9 3.6 1.4 0.1 Iris-setosa
4.4 3.0 1.3 0.2 Iris-setosa
5.1 3.4 1.5 0.2 Iris-setosa
5.0 3.5 1.3 0.3 Iris-setosa
4.5 2.3 1.3 0.3 Iris-setosa
4.4 3.2 1.3 0.2 Iris-setosa
5.0 3.5 1.6 0.6 Iris-setosa
5.1 3.8 1.9 0.4 Iris-setosa
4.8 3.0 1.4 0.3 Iris-setosa
5.1 3.8 1.6 0.2 Iris-setosa
4.6 3.2 1.4 0.2 Iris-setosa
5.3 3.7 1.5 0.2 Iris-setosa
5.0 3.3 1.4 0.2 Iris-setosa
7.0 3.2 4.7 1.4 Iris-versicolor
6.4 3.2 4.5 1.5 Iris-versicolor
6.9 3.1 4.9 1.5 Iris-versicolor
5.5 2.3 4.0 1.3 Iris-versicolor
6.5 2.8 4.6 1.5 Iris-versicolor
5.7 2.8 4.5 1.3 Iris-versicolor
6.3 3.3 4.7 1.6 Iris-versicolor
4.9 2.4 3.3 1.0 Iris-versicolor
6.6 2.9 4.6 1.3 Iris-versicolor
5.2 2.7 3.9 1.4 Iris-versicolor
5.0 2.0 3.5 1.0 Iris-versicolor
5.9 3.0 4.2 1.5 Iris-versicolor
6.0 2.2 4.0 1.0 Iris-versicolor
6.1 2.9 4.7 1.4 Iris-versicolor
5.6 2.9 3.6 1.3 Iris-versicolor
6.7 3.1 4.4 1.4 Iris-versicolor
5.6 3.0 4.5 1.5 Iris-versicolor
5.8 2.7 4.1 1.0 Iris-versicolor
6.2 2.2 4.5 1.5 Iris-versicolor
5.6 2.5 3.9 1.1 Iris-versicolor
5.9 3.2 4.8 1.8 Iris-versicolor
6.1 2.8 4.0 1.3 Iris-versicolor
6.3 2.5 4.9 1.5 Iris-versicolor
6.1 2.8 4.7 1.2 Iris-versicolor
6.4 2.9 4.3 1.3 Iris-versicolor
6.6 3.0 4.4 1.4 Iris-versicolor
6.8 2.8 4.8 1.4 Iris-versicolor
6.7 3.0 5.0 1.7 Iris-versicolor
6.0 2.9 4.5 1.5 Iris-versicolor
5.7 2.6 3.5 1.0 Iris-versicolor
5.5 2.4 3.8 1.1 Iris-versicolor
```

```
5.5 2.4 3.7 1.0 Iris-versicolor
5.8 2.7 3.9 1.2 Iris-versicolor
6.0 2.7 5.1 1.6 Iris-versicolor
5.4 3.0 4.5 1.5 Iris-versicolor
6.0 3.4 4.5 1.6 Iris-versicolor
6.7 3.1 4.7 1.5 Iris-versicolor
6.3 2.3 4.4 1.3 Iris-versicolor
5.6 3.0 4.1 1.3 Iris-versicolor
5.5 2.5 4.0 1.3 Iris-versicolor
5.5 2.6 4.4 1.2 Iris-versicolor
6.1 3.0 4.6 1.4 Iris-versicolor
5.8 2.6 4.0 1.2 Iris-versicolor
5.0 2.3 3.3 1.0 Iris-versicolor
5.6 2.7 4.2 1.3 Iris-versicolor
5.7 3.0 4.2 1.2 Iris-versicolor
5.7 2.9 4.2 1.3 Iris-versicolor
6.2 2.9 4.3 1.3 Iris-versicolor
5.1 2.5 3.0 1.1 Iris-versicolor
5.7 2.8 4.1 1.3 Iris-versicolor
6.3 3.3 6.0 2.5 Iris-virginica
5.8 2.7 5.1 1.9 Iris-virginica
7.1 3.0 5.9 2.1 Iris-virginica
6.3 2.9 5.6 1.8 Iris-virginica
6.5 3.0 5.8 2.2 Iris-virginica
7.6 3.0 6.6 2.1 Iris-virginica
4.9 2.5 4.5 1.7 Iris-virginica
7.3 2.9 6.3 1.8 Iris-virginica
6.7 2.5 5.8 1.8 Iris-virginica
7.2 3.6 6.1 2.5 Iris-virginica
6.5 3.2 5.1 2.0 Iris-virginica
6.4 2.7 5.3 1.9 Iris-virginica
6.8 3.0 5.5 2.1 Iris-virginica
5.7 2.5 5.0 2.0 Iris-virginica
5.8 2.8 5.1 2.4 Iris-virginica
6.4 3.2 5.3 2.3 Iris-virginica
6.5 3.0 5.5 1.8 Iris-virginica
7.7 3.8 6.7 2.2 Iris-virginica
7.7 2.6 6.9 2.3 Iris-virginica
6.0 2.2 5.0 1.5 Iris-virginica
6.9 3.2 5.7 2.3 Iris-virginica
5.6 2.8 4.9 2.0 Iris-virginica
7.7 2.8 6.7 2.0 Iris-virginica
6.3 2.7 4.9 1.8 Iris-virginica
6.7 3.3 5.7 2.1 Iris-virginica
7.2 3.2 6.0 1.8 Iris-virginica
6.2 2.8 4.8 1.8 Iris-virginica
6.1 3.0 4.9 1.8 Iris-virginica
6.4 2.8 5.6 2.1 Iris-virginica
7.2 3.0 5.8 1.6 Iris-virginica
7.4 2.8 6.1 1.9 Iris-virginica
7.9 3.8 6.4 2.0 Iris-virginica
6.4 2.8 5.6 2.2 Iris-virginica
6.3 2.8 5.1 1.5 Iris-virginica
6.1 2.6 5.6 1.4 Iris-virginica
7.7 3.0 6.1 2.3 Iris-virginica
6.3 3.4 5.6 2.4 Iris-virginica
```

```
138 6.4 3.1 5.5 1.8 Iris-virginica
139 6.0 3.0 4.8 1.8 Iris-virginica
140 6.9 3.1 5.4 2.1 Iris-virginica
141 6.7 3.1 5.6 2.4 Iris-virginica
142 6.9 3.1 5.1 2.3 Iris-virginica
143 5.8 2.7 5.1 1.9 Iris-virginica
144 6.8 3.2 5.9 2.3 Iris-virginica
145 6.7 3.3 5.7 2.5 Iris-virginica
146 6.7 3.0 5.2 2.3 Iris-virginica
147 6.3 2.5 5.0 1.9 Iris-virginica
148 6.5 3.0 5.2 2.0 Iris-virginica
149 6.2 3.4 5.4 2.3 Iris-virginica
150 5.9 3.0 5.1 1.8 Iris-virginica
```

Anexo D

Aplicação Completa

Código D.1: Função *setweights* para o vetor de pesos inicial determinado.

```
(define (setweights v1 v2 v3)
   (define v '#(0.499 0.262 0.181 -0.079 -0.361 -0.162 -0.153 0.392 0.435
      0.113 0.212 -0.174 0.493 0.194 -0.169 0.198 -0.095 0.278 -0.463 0.01
      -0.454 0.24 -0.339 -0.121 0.131 -0.206 -0.404 0.358 -0.434 0.189
      -0.295 -0.125 -0.049 -0.305 -0.394 0.4 0.344 -0.047 -0.399 0.279
      -0.124 -0.378 -0.086 -0.131 0.317 -0.445 -0.434 -0.278 0.334 -0.086
      0.232 -0.31 -0.347 -0.297 0.38 0.285 0.308 0.476 0.143 -0.316 0.475
      -0.342 0.06 -0.075 -0.146 0.166 0.135 -0.303 -0.071 0.237 -0.214
      0.306 0.36 0.201 -0.325 0.487 0.256 0.052 0.018 0.4 0.466 0.06
      -0.411))
   (define n (vector-length v1))
   (let loop[(i 0)]
      (cond
         [(< i n)
            (vector-set! v1 i (vector-ref v i))
            (vector-set! v2 i (vector-ref v i))
            (vector-set! v3 i (vector-ref v i))
            (loop (+ i 1))])))
```

Código D.2: Aplicação completa para treinamento das RNAs.

```
;Programa para treinar uma RNA para aprender a reconhecer classes de flor
    Iris, baseando-se na largura e no comprimento de pétalas e sépalas.
    Utiliza os métodos BP, AIP e AIP paralelo
;Autor: Will Roger Pereira
;Programado em Bigloo 3.6a

(module iris
   (library fthread)
   (main start)
)

(define nthd 8) ;Número de threads utilizadas
(define nweights 83) ;Número de pesos das RNAs
```

```
12 (define ciclosmax 10000) ;Número máximo de ciclos
13 (define erromax 0.25) ;Erro máximo permitido
14 (define alfa 0.025) ;Taxa de aprendizagem
15 (define dx 0.1) ;Delta método AIP
16
17 (define (start args)
18    (thdcriador nthd nweights)
19    (thread-start! (instantiate::fthread (body (comandante))))
20    (thread-start! (instantiate::fthread (body (principal))))
21    (scheduler-start!))
22
23 (define (principal)
24    (lambda()
25       (define data (getinfo "iris.data"))
26       (define traininputs '())
27       (define testinputs '())
28       (define traintargets '())
29       (define testtargets '())
30       (define gw (make-vector nweights 0))
31       (define bw (make-vector nweights 0))
32       (define tw (make-vector nweights 0))
33       (define rnaGD (gate gw))
34       (define rnaBP (gatebp bw))
35       (define rnaTH (gate tw))
36       (define dx 0.01)
37       (define saida '())
38       (multiple-value-bind (xi xt yi yt)
39          (splitter data)
40          (set! traininputs xi)
41          (set! traintargets xt)
42          (set! testinputs yi)
43          (set! testtargets yt)
44       )
45       (setweights bw gw tw)
46       (print "Treinando RNA pelo metodo BP. Aguarde...")
47       (multiple-value-bind (res rtime)
48          (time (lambda() (train rnaBP traininputs traintargets ciclosmax
                erromax alfa dx "BP")))
49          (print "============RESULTADOS MÉTODO BP===============")
50          (print "Ciclos Gastos: " (car res))
51          (print "Tempo Gasto: " (/ rtime 1000) " segundos.")
52          (print "Erro Quadrático Total: " (cadr res))
53       )
54       (set! saida (setallsaidas (map (lambda(x) (cadr (rnaBP x)))
             traininputs)))
55       (print "\n=====Grupo de Treino=====")
56       (print "Acertos/Erros: " (corretos traintargets saida))
57       (set! saida (calcallsaidas (classallsaidas saida)))
58       (print "Padrão Treino: " (calcallsaidas (classallsaidas traintargets)
             ))
59       (print "Resultado Treino: " saida)
60       (print "Classes: Setosa/Versicolor/Virginica/Não Sei")
61       (set! saida (setallsaidas (map (lambda(x) (cadr (rnaBP x)))
             testinputs)))
62       (print "\n=====Grupo de Teste=====")
63       (print "Acertos/Erros: " (corretos testtargets saida))
```

```
64      (set! saida (calcallsaidas (classallsaidas saida)))
65      (print "Padrão Teste: " (calcallsaidas (classallsaidas testtargets)))
66      (print "Resultado Teste: " saida)
67      (print "Classes: Setosa/Versicolor/Virginica/Não Sei")
68
69      (print "\nTreinando RNA pelo método AIP. Aguarde...")
70      (multiple-value-bind (res rtime)
71         (time (lambda() (train rnaGD traininputs traintargets ciclosmax
               erromax alfa dx "GD")))
72         (print "============RESULTADOS MÉTODO AIP===============")
73         (print "Ciclos Gastos: " (car res))
74         (print "Tempo Gasto: " (/ rtime 1000) " segundos.")
75         (print "Erro Quadrático Total: " (cadr res))
76      )
77      (set! saida (setallsaidas (map (lambda(x) (rnaGD x)) traininputs)))
78      (print "\n=====Grupo de Treino=====")
79      (print "Acertos/Erros: " (corretos traintargets saida))
80      (set! saida (calcallsaidas (classallsaidas saida)))
81      (print "Padrão Treino: " (calcallsaidas (classallsaidas traintargets)
           ))
82      (print "Resultado Treino: " saida)
83      (print "Classes: Setosa/Versicolor/Virginica/Não Sei")
84      (set! saida (setallsaidas (map (lambda(x) (rnaGD x)) testinputs)))
85      (print "\n=====Grupo de Teste=====")
86      (print "Acertos/Erros: " (corretos testtargets saida))
87      (set! saida (calcallsaidas (classallsaidas saida)))
88      (print "Padrão Teste: " (calcallsaidas (classallsaidas testtargets)))
89      (print "Resultado Teste: " saida)
90      (print "Classes: Setosa/Versicolor/Virginica/Não Sei")
91
92      (print "\nTreinando RNA pelo método AIP paralelo. Aguarde...")
93      (multiple-value-bind (res rtime)
94         (time (lambda() (train rnaTH traininputs traintargets ciclosmax
               erromax alfa dx "TH")))
95         (print "========RESULTADOS MÉTODO AIP PARALELO============")
96         (print "Ciclos Gastos: " (car res))
97         (print "Tempo Gasto: " (/ rtime 1000) " segundos.")
98         (print "Erro Quadrático Total: " (cadr res))
99      )
100     (set! saida (setallsaidas (map (lambda(x) (rnaTH x)) traininputs)))
101     (print "\n=====Grupo de Treino=====")
102     (print "Acertos/Erros: " (corretos traintargets saida))
103     (set! saida (calcallsaidas (classallsaidas saida)))
104     (print "Padrão Treino: " (calcallsaidas (classallsaidas traintargets)
           ))
105     (print "Resultado Treino: " saida)
106     (print "Classes: Setosa/Versicolor/Virginica/Não Sei")
107     (set! saida (setallsaidas (map (lambda(x) (rnaTH x)) testinputs)))
108     (print "\n=====Grupo de Teste=====")
109     (print "Acertos/Erros: " (corretos testtargets saida))
110     (set! saida (calcallsaidas (classallsaidas saida)))
111     (print "Padrão Teste: " (calcallsaidas (classallsaidas testtargets)))
112     (print "Resultado Teste: " saida)
113     (print "Classes: Setosa/Versicolor/Virginica/Não Sei")
114
115     (print "\n===========PROGRAMA FINALIZADO============")
```

```
116         (exit 1)))
117
118 (define (bipsig x) ;Função de ativação sigmoide bipolar
119     (- (/ 2 (+ 1 (exp (- x)))) 1))
120
121 (define (dbipsig x) ;Derivada da função de ativação(BP)
122     (* 0.5 (+ 1 x) (- 1 x)))
123
124 (define (newneuron weights indexes)
125     (lambda(inputs)
126         (let loop[(i indexes) (x (cons 1.0 inputs)) (acc 0.0)]
127             (cond
128                 [(or (null? i) (null? x))
129                     (bipsig acc)]
130                 [else
131                     (loop (cdr i) (cdr x) (+ acc (* (vector-ref weights (car i))
                                (car x))))]))))
132
133 (define (gate weights)
134     (let [(n10 (newneuron weights (map (lambda(x) (+ x (* 5 0))) (
            list-tabulate 5 values))))
135             (n11 (newneuron weights (map (lambda(x) (+ x (* 5 1))) (
                list-tabulate 5 values))))
136             (n12 (newneuron weights (map (lambda(x) (+ x (* 5 2))) (
                list-tabulate 5 values))))
137             (n13 (newneuron weights (map (lambda(x) (+ x (* 5 3))) (
                list-tabulate 5 values))))
138             (n14 (newneuron weights (map (lambda(x) (+ x (* 5 4))) (
                list-tabulate 5 values))))
139             (n15 (newneuron weights (map (lambda(x) (+ x (* 5 5))) (
                list-tabulate 5 values))))
140             (n16 (newneuron weights (map (lambda(x) (+ x (* 5 6))) (
                list-tabulate 5 values))))
141             (n17 (newneuron weights (map (lambda(x) (+ x (* 5 7))) (
                list-tabulate 5 values))))
142             (n18 (newneuron weights (map (lambda(x) (+ x (* 5 8))) (
                list-tabulate 5 values))))
143             (n19 (newneuron weights (map (lambda(x) (+ x (* 5 9))) (
                list-tabulate 5 values))))
144             (n20 (newneuron weights (map (lambda(x) (+ x (* 11 0) (* 5 10))) (
                list-tabulate 11 values))))
145             (n21 (newneuron weights (map (lambda(x) (+ x (* 11 1) (* 5 10))) (
                list-tabulate 11 values))))
146             (n22 (newneuron weights (map (lambda(x) (+ x (* 11 2) (* 5 10))) (
                list-tabulate 11 values))))]
147         (lambda(in)
148             (cond
149                 [(null? in)
150                     weights]
151                 [else
152                     (let [(outc1 (list (n10 in) (n11 in) (n12 in) (n13 in) (n14
                            in) (n15 in) (n16 in) (n17 in) (n18 in) (n19 in)))]
153                         (list
154                             (n20 outc1)
155                             (n21 outc1)
156                             (n22 outc1)
```

```
157                  )
158                )])))))
159
160 (define (gatebp weights)
161    (let [(n10 (newneuron weights (map (lambda(x) (+ x (* 5 0))) (
          list-tabulate 5 values))))
162          (n11 (newneuron weights (map (lambda(x) (+ x (* 5 1))) (
             list-tabulate 5 values))))
163          (n12 (newneuron weights (map (lambda(x) (+ x (* 5 2))) (
             list-tabulate 5 values))))
164          (n13 (newneuron weights (map (lambda(x) (+ x (* 5 3))) (
             list-tabulate 5 values))))
165          (n14 (newneuron weights (map (lambda(x) (+ x (* 5 4))) (
             list-tabulate 5 values))))
166          (n15 (newneuron weights (map (lambda(x) (+ x (* 5 5))) (
             list-tabulate 5 values))))
167          (n16 (newneuron weights (map (lambda(x) (+ x (* 5 6))) (
             list-tabulate 5 values))))
168          (n17 (newneuron weights (map (lambda(x) (+ x (* 5 7))) (
             list-tabulate 5 values))))
169          (n18 (newneuron weights (map (lambda(x) (+ x (* 5 8))) (
             list-tabulate 5 values))))
170          (n19 (newneuron weights (map (lambda(x) (+ x (* 5 9))) (
             list-tabulate 5 values))))
171          (n20 (newneuron weights (map (lambda(x) (+ x (* 11 0) (* 5 10))) (
             list-tabulate 11 values))))
172          (n21 (newneuron weights (map (lambda(x) (+ x (* 11 1) (* 5 10))) (
             list-tabulate 11 values))))
173          (n22 (newneuron weights (map (lambda(x) (+ x (* 11 2) (* 5 10))) (
             list-tabulate 11 values))))]
174       (lambda(in)
175          (cond
176             [(null? in)
177                weights]
178             [else
179                (let [(outc1 (list (n10 in) (n11 in) (n12 in) (n13 in) (n14
                      in) (n15 in) (n16 in) (n17 in) (n18 in) (n19 in)))]
180                   (list
181                      outc1
182                      (list
183                         (n20 outc1)
184                         (n21 outc1)
185                         (n22 outc1)
186                      )
187                   )
188                )])))))
189
190 (define (errsum targets reals mode)
191    (define err 0.0)
192    (cond
193       [(or (equal? mode "GD") (equal? mode "TH"))
194          (map (lambda(x1 x2) (map (lambda(y1 y2) (set! err (+ err (* (- y1
                y2) (- y1 y2))))) x1 x2)) targets reals)]
195       [(equal? mode "BP")
196          (map (lambda(x1 x2) (map (lambda(y1 y2) (set! err (+ err (* (- y1
                y2) (- y1 y2))))) x1 x2)) targets (map (lambda(x) (cadr x))
```

```
            reals))]
197    )
198    err)
199
200 (define (updateweights weights adjusts)
201    (define tam (vector-length weights))
202    (let loop[(i 0)]
203       (cond
204          [(< i tam)
205             (vector-set! weights i (+ (vector-ref weights i) (vector-ref
                  adjusts i)))
206             (loop (+ i 1))])))
207
208 (define (train rna inputs targets cicmax errmax alfa dx mode)
209    (define adj (make-vector (vector-length (rna '())) 0))
210    (define adjback (make-vector (vector-length (rna '())) 0))
211    (let loop[(ciclos 0) (erro (errsum targets (map (lambda(x) (rna x))
       inputs) mode))]
212       (cond
213          [(or (>= ciclos cicmax) (< erro errmax))
214             (list ciclos erro)]
215          [else
216             (cond
217                [(equal? mode "GD")
218                   (AIP adj (rna '()) erro inputs targets alfa dx)
219                   (updateweights (rna '()) adj)]
220                [(equal? mode "TH")
221                   (broadcast! 'start (list (rna '()) erro inputs targets
                        alfa dx))
222                   (updateweights (rna '()) (car (thread-await-values! 'done
                        )))]
223                [(equal? mode "BP")
224                   (map (lambda(x y) (BP adj (rna '()) x y (rna x) alfa) (
                        updateweights adjback adj)) inputs targets)
225                   (updateweights (rna '()) adjback)
226                   (set! adjback (make-vector (vector-length (rna '())) 0))]
227             )
228             (loop (+ ciclos 1) (errsum targets (map (lambda(x) (rna x))
                inputs) mode))])))
229
230 (define (BP adj weights input target output alfa) ;Backpropagation
231    (define nmax 13)
232    (define nescondida 10)
233    (define nsaida (- nmax nescondida))
234    (define imax (length input))
235    (define neu (make-vector nmax 0))
236    (define wmax (- (vector-length weights) 1))
237    ;Valores dos neurônios da camada de saida
238    (let loop[(n (- nmax 1))]
239       (cond
240          [(> n (- nescondida 1))
241             (vector-set! neu n (* (dbipsig (list-ref (list-ref output 1) (-
                  n nescondida))) (- (list-ref target (- n nescondida)) (
                  list-ref (list-ref output 1) (- n nescondida)))))
242             (loop (- n 1))]))
243    ;Valores dos neurônios da camada escondida
```

```
244     (let loop[(n (- nescondida 1)) (w wmax) (s (- nmax 1)) (sum 0)]
245        (cond
246           [(>= n 0)
247              (cond
248                 [(> s (- nescondida 1))
249                    (loop n (- w (+ nescondida 1)) (- s 1) (+ sum (* (
                          vector-ref neu s) (vector-ref weights w))))]
250                 [else
251                    (vector-set! neu n (* (dbipsig (list-ref (list-ref output
                          0) n)) sum))
252                    (loop (- n 1) (- wmax (- nescondida 1 n)) (- nmax 1) 0)]
253              )]))
254     ;Ajuste da camada escondida
255     (let loop[(a 0) (n 0) (in 0)]
256        (cond
257           [(< n nescondida)
258              (cond
259                 [(and (= (remainder a (+ imax 1)) 0) (< in imax))
260                    (vector-set! adj a (* alfa (vector-ref neu n)))
261                    (loop (+ a 1) n in)]
262                 [(< in imax)
263                    (vector-set! adj a (* alfa (vector-ref neu n) (list-ref
                          input in)))
264                    (loop (+ a 1) n (+ in 1))]
265                 [else
266                    (loop a (+ n 1) 0)]
267              )]))
268     ;Ajuste da camada de saída
269     (let loop[(a 0) (n nescondida) (out 0)]
270        (cond
271           [(< n nmax)
272              (cond
273                 [(and (= (remainder a (+ nescondida 1)) 0) (< out nescondida
                       ))
274                    (vector-set! adj (+ a (* (+ imax 1) nescondida)) (* alfa
                          (vector-ref neu n)))
275                    (loop (+ a 1) n out)]
276                 [(< out nescondida)
277                    (vector-set! adj (+ a (* (+ imax 1) nescondida)) (* alfa
                          (vector-ref neu n) (list-ref (list-ref output 0) out))
                          )
278                    (loop (+ a 1) n (+ out 1))]
279                 [else
280                    (loop a (+ n 1) 0)]
281              )])))
282
283  (define (AIPadj i weights erro inputs targets alfa dx)
284     (define tam (vector-length weights))
285     (define newrna (gate (copy-vector weights tam)))
286     (vector-set! (newrna '()) i (+ (vector-ref (newrna '()) i) dx))
287     (let [(newerro (errsum targets (map (lambda(x) (newrna x)) inputs) "GD")
           )]
288        (- (* alfa (/ (- newerro erro) dx)))))
289
290  (define (AIP adj weights erro inputs targets alfa dx)
291     (define tam (vector-length weights))
```

```
292     (let loop[(i 0)]
293        (cond
294           [(< i tam)
295              (vector-set! adj i (AIPadj i weights erro inputs targets alfa
                    dx))
296              (loop (+ i 1))])))
297
298  (define (comandante)
299     (lambda()
300        (define adjs (make-vector nweights 0))
301        (let loop[(lista (car (thread-await-values! 'start)))]
302           (let loopsend[(i 0)]
303              (cond
304                 [(< i nthd)
305                    (broadcast! (string-append "t" (number->string i)) lista)
306                    (loopsend(+ i 1))]
307              )
308           )
309           (let looprec[(j 0) (sigs (list-tabulate nthd values))]
310              (cond
311                 [(< j nthd)
312                    (multiple-value-bind(info sig)
313                       (thread-await*! sigs)
314                       (map
315                          (lambda(x y)
316                             (vector-set! adjs x y)
317                          )
318                          (car info)
319                          (cadr info)
320                       )
321                       (looprec (+ j 1) (delete sig sigs))
322                    )]
323              )
324           )
325           (broadcast! 'done adjs)
326           (loop (car (thread-await-values! 'start)))
327        )))
328
329  (define (operaria n index)
330     (lambda()
331        (let loop[(data (car (thread-await-values! (string-append "t" (
              number->string n)))))]
332           (let* [(weights (list-ref data 0)) (erro (list-ref data 1)) (
              inputs (list-ref data 2)) (targets (list-ref data 3)) (alfa (
              list-ref data 4)) (dx (list-ref data 5))]
333              (let loop1[(l index) (result (list))]
334                 (cond
335                    [(> (length l) 0)
336                       (loop1 (cdr l) (append result (list (car (
                             thread-await-values! (make-asynchronous-signal (
                             lambda(x) (AIPadj (car l) weights erro inputs
                             targets alfa dx))))))))]
337                    [else
338                       (broadcast! n (list index result))
339                       (loop (car (thread-await-values! (string-append "t" (
                             number->string n)))))])))
```

```
340         )))
341
342 (define (thdcriador nthreads npesos)
343     (define lista (list-split (list-tabulate npesos values) (flonum->fixnum
            (+ (/ npesos nthreads) 1))))
344     (let loop[(i 0) (l lista)]
345         (cond
346             [(< i nthreads)
347                 (thread-start! (instantiate::fthread (body (operaria i (car l))
                        )))
348                 (loop (+ i 1) (cdr l))])))
349
350 ;Funções para extrair e tratar a informação da base de dados
351 (define (getinfo file)
352     (define res (list))
353     (define maximos (list))
354     (define minimos (list))
355     (set! res
356         (with-input-from-file file
357             (lambda()
358                 (map (lambda(x) (string-split x " ")) (read-lines)))))
359     (set! res
360         (map
361             (lambda(x)
362                 (map
363                     (lambda(y)
364                         (cond
365                             [(string->number y)
366                                 (string->number y)]
367                             [else
368                                 y]
369                         )
370                     )
371                     x
372                 )
373             )
374             res))
375     (set! maximos (getlstfunc res >))
376     (set! minimos (getlstfunc res <))
377     (map (lambda(x) (append (normaliza (take x (- (length x) 1)) maximos
            minimos) (drop x (- (length x) 1)))) res))
378
379 (define (getindex lst i)
380     (define res '())
381     (map (lambda(x) (set! res (append res (list (list-ref x i))))) lst)
382     res)
383
384 (define (getlstfunc lst func)
385     (define tam (- (length (car lst)) 2))
386     (define res '())
387     (define i 0)
388     (let loop[(valor (car (getindex lst i))) (l (cdr (getindex lst i)))]
389         (cond
390             [(null? l)
391                 (set! res (append res (list valor)))
392                 (cond
```

```
393                 [(< i tam)
394                     (set! i (+ i 1))
395                     (loop (car (getindex lst i)) (cdr (getindex lst i)))]
396             )]
397         [else
398             (cond
399                 [(func (car l) valor)
400                     (loop (car l) (cdr l))]
401                 [else
402                     (loop valor (cdr l))]
403             )]
404         )
405     )
406     res)
407 
408 (define (normaliza lst maxi mini)
409     (define tam (length lst))
410     (define res '())
411     (let loop[(i 0)]
412         (cond
413             [(< i tam)
414                 (set! res (append res (list (- (* 2 (/ (- (list-ref lst i) (
                        list-ref mini i)) (- (list-ref maxi i) (list-ref mini i))))
                        1))))
415                 (loop (+ i 1))]
416         )
417     )
418     res)
419 
420 (define (settargets t)
421     (cond
422         [(equal? t "Iris-setosa")
423             (list 1 -1 -1)]
424         [(equal? t "Iris-versicolor")
425             (list -1 1 -1)]
426         [(equal? t "Iris-virginica")
427             (list -1 -1 1)]))
428 
429 (define (splitter lista)
430     (define treino '())
431     (define teste '())
432     (define treinoin '())
433     (define treinotarget '())
434     (define testein '())
435     (define testetarget '())
436     (let loop[(i 0) (l lista)]
437         (cond
438             [(even? i)
439                 (set! treino (append treino (list (car l))))]
440             [else
441                 (set! teste (append teste (list (car l))))]
442         )
443         (cond
444             [(> (length l) 1)
445                 (loop (+ i 1) (cdr l))]
446         )
```

```
447     )
448     (set! treinoin (map (lambda(x) (take x (- (length x) 1))) treino))
449     (set! treinotarget (map (lambda(x) (settargets (car x))) (map (lambda(y)
            (drop y (- (length y) 1))) treino)))
450     (set! testein (map (lambda(x) (take x (- (length x) 1))) teste))
451     (set! testetarget (map (lambda(x) (settargets (car x))) (map (lambda(y)
            (drop y (- (length y) 1))) teste)))
452     (values treinoin treinotarget testein testetarget))
453
454 ;Função para fornecer pesos aleatórios
455 (define (setweights v1 v2 v3)
456     (define n (vector-length v1))
457     (let loop[(i 0) (num (- 0.5 (/ (random 1001) 1000)))]
458         (cond
459             [(< i n)
460                 (vector-set! v1 i num)
461                 (vector-set! v2 i num)
462                 (vector-set! v3 i num)
463                 (loop (+ i 1) (- 0.5 (/ (random 1001) 1000)))])))
464
465 ;Funções para tratar as saídas RNA
466 (define (setallsaidas saidas::pair)
467     (map (lambda(x) (map (lambda(y) (setsaida y)) x)) saidas))
468
469 (define (setsaida s::real)
470     (cond
471         [(>= s 0.7)
472             1]
473         [(<= s -0.7)
474             -1]
475         [else
476             0]))
477
478 (define (classallsaidas saidas::pair)
479     (map (lambda(x) (classsaida x)) saidas))
480
481 (define (classsaida s::pair)
482     (cond
483         [(equal? s '(1 -1 -1))
484             1]
485         [(equal? s '(-1 1 -1))
486             2]
487         [(equal? s '(-1 -1 1))
488             3]
489         [else
490             0]))
491
492 (define (calcallsaidas saidas::pair)
493     (define um 0)
494     (define dois 0)
495     (define tres 0)
496     (define idk 0)
497     (let loop[(l saidas)]
498         (cond
499             [(= (car l) 1)
500                 (set! um (+ um 1))]
```

```
501             [(= (car l) 2)
502                (set! dois (+ dois 1))]
503             [(= (car l) 3)
504                (set! tres (+ tres 1))]
505             [(= (car l) 0)
506                (set! idk (+ idk 1))]
507          )
508          (cond
509             [(> (length l) 1)
510                (loop (cdr l))]
511          )
512       )
513       (string-append (number->string um) "/" (number->string dois) "/" (
             number->string tres) "/" (number->string idk)))
514
515    (define (corretos target real)
516       (define certo 0)
517       (define errado 0)
518       (map (lambda(x y)
519          (cond
520             [(equal? x y)
521                (set! certo (+ certo 1))]
522             [else
523                (set! errado (+ errado 1))]
524          ))
525          target real
526       )
527       (string-append (number->string certo) "/" (number->string errado) " = "
             (number->string (flonum->fixnum (round (* (/ certo (+ certo errado))
             100.0)))) "%"))
```