TRABALHO DE GRADUAÇÃO

IMPLEMENTAÇÃO DE REDE NEURAL EM HARDWARE DE PONTO FIXO

Carlos Felipe Ávila Klein
Joel Fernando Jardim Martins

Brasília, dezembro de 2006.

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia

TRABALHO DE GRADUAÇÃO

IMPLEMENTAÇÃO DE REDE NEURAL EM HARDWARE DE PONTO FIXO

Carlos Felipe Ávila Klein
Joel Fernando Jardim Martins

Relatório submetido como requisito parcial para obtenção
do grau de Engenheiro Eletricista

Banca Examinadora

Prof. Alexandre Ricardo Soares Romariz, Ph.D, UnB/ ENE (Orientador) ____________________

Prof. Francisco Assis de Oliveira Nascimento, Doutor, UnB/ ENE ____________________

Prof.ª Janaína Gonçalves Guimarães, Doutora, UnB/ ENE ____________________

FICHA CATALOGRÁFICA

KLEIN, CARLOS FELIPE ÁVILA & MARTINS, JOEL FERNANDO JARDIM
Implementação de Rede Neural em Hardware de Ponto Fixo [Distrito Federal] 2006.
87p., (ENE/FT/UnB, Engenheiro Eletricista, 2006). Monografia de Graduação –
Universidade de Brasília. Faculdade de Tecnologia.
Departamento de Engenharia Elétrica.

1. Redes Neurais 2. Algoritmo "backpropagation"
3. Hardware de ponto fixo TMS320VC5510 4. Code Composer Studio
I. ENE/FT/UnB II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

KLEIN, C. F. A. & MARTINS, J. F. J. (2006). Implementação de Rede Neural em Hardware de Ponto Fixo. Monografia de Graduação, Publicação ENE 02/2006, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 87p.

CESSÃO DE DIREITOS

AUTORES: Carlos Felipe Ávila Klein e Joel Fernando Jardim Martins.

TÍTULO: Implementação de Rede Neural em Hardware de Ponto Fixo.

GRAU: Engenheiro Eletricista ANO: 2006

Carlos Felipe Ávila Klein
CLN 408, bloco E, sala 110.
70.856-550 Brasília – DF – Brasil.

Joel Fernando Jardim Martins
CEU/UnB, bloco B, apt. 201.
70.910-900 Brasília – DF – Brasil.

Dedicatórias

Aos meus pais.
A Michella.

Joel Fernando Jardim Martins

Aos meus pais.

Carlos Felipe Ávila Klein

Agradecimentos

Ao fim dessa jornada, agradecemos a todos que conosco partilharam o caminho. Com sonhos e medos, lágrimas e sorrisos, por ele passamos e chegamos ao fim.

Agradecemos primeiramente a Deus que sempre esteve presente em nossas madrugadas insones e nos deu forças em nossos dias de dúvidas, silêncio, desânimo ou solidão.

Com gratidão lembramos de nossos mestres, cujas tarefas não se resumiram somente a transmitir conhecimentos, sendo antes de tudo, exemplos de dedicação, doação, dignidade pessoal e, principalmente, amor. A vocês, professores, que souberam ser nossos amigos, alegrando-se com nossas vitórias e apoiando-nos em nossas lutas mais difíceis, o nosso muito obrigado.

Aos nossos queridos colegas, temos certeza de que não cabe aqui um adeus, mas um até breve. Com vocês, amigos, vivemos tantas experiências que, com nosso convívio, nos transformamos em companheiros leais de lutas e vitórias.

Aos nossos pais, que muitas vezes sacrificaram seus sonhos para que os nossos fossem realizados e que sempre nos incentivaram para que não desistíssemos de nossos ideais, ajudando-nos a transpor os obstáculos por meio da conversa sincera, amiga e segura, as palavras são insuficientes para externar nossos sentimentos de apreço e amor. A vocês, conselheiros por excelência, dedicamos toda a nossa conquista, porque ela também lhes pertence.

Ao Professor Doutor Francisco Assis de Oliveira Nascimento por ter possibilitado os meios materiais necessários à nossa pesquisa.

Aos professores Alexandre Romariz, Alexandre Zaghetto e Janaína Guimarães, que em todo tempo estiveram dispostos a nos ajudar e resolver quaisquer dúvidas que surgiram durante a execução do projeto, sempre apresentando muita paciência, cordialidade e respeito para conosco.

Por fim, a todos aqueles a quem amamos, o nosso abraço de carinho e nossa gratidão. Sentiremos saudades...

Carlos Felipe Ávila Klein
Joel Fernando Jardim Martins

RESUMO

IMPLEMENTAÇÃO DE REDE NEURAL EM HARDWARE DE PONTO FIXO.
Autores: Carlos Felipe Ávila Klein e Joel Fernando Jardim Martins
Orientador: Alexandre Ricardo Soares Romariz
Palavras-chave: Redes Neurais MLP, algoritmo “backpropagation”, hardware de ponto fixo TMS320VC5510, Code Composer Studio.
Brasília, 12 de dezembro de 2006.

O presente trabalho apresenta um estudo sobre a utilização de Redes Neurais Multi-Layer Perceptron (MLP) com o seu treinamento efetuado baseado no algoritmo “Backpropagation”, também conhecido como regra delta generalizada. Demonstra-se o funcionamento dessas Redes em computadores domésticos e em hardware de ponto fixo TMS320VC5510® da Texas Instruments, com intuito de fazer uma comparação entre elas e checar a viabilidade de uma rede em hardware de precisão limitada.

Tendo em vista o tipo de rede e o algoritmo de aprendizagem, foram desenvolvidos algoritmos em linguagem de programação C portáveis para as plataformas PC e TMS320VC5510.

Foram feitas análises relativas à quantidade de iterações necessárias para que as redes fornecessem saídas com uma precisão pré-determinada para alguns problemas bastante simples. Analisou-se também a porcentagem de acerto para o problema de classificação de espécies de Íris, clássico na literatura de redes neurais. Além disso, foram feitas análises de convergência de erro para todos os programas desenvolvidos.

ABSTRACT

NEURAL NETWORK IMPLEMENTATION IN FIXED-POINT HARDWARE.

Authors: Carlos Felipe Ávila Klein e Joel Fernando Jardim Martins
Advisor: Alexandre Ricardo Soares Romariz
Keywords: MLP Neural Networks, Backpropagation Algorithm, TMS320VC5510 Fixed-Point Hardware, Code Composer Studio.
Brasília, december/2006.

This work presents a study of the use of Multi-Layer Perceptron (MLP) Neural Networks, its training based on the Backpropagation algorithm, also known as generalized delta rule. The implementation of these Networks in home computers and in the TMS320VC5510 fixed-point processor, from Texas Instruments, is demonstrated with the intention of making a comparison between them and to check the viability of a network in a hardware of limited precision.

Knowing the type of network and the learning algorithm, some programs were developed using the C programming language. The algorithms implemented kept portability for both IBM and TMS320VC5510 platforms.

Simulations were made to determine the amount of iterations necessary for the networks to provide outputs with a predetermined precision in simple problems. The ratio of success for the classical Iris problem was also investigated. Also, some examples of the convergence of error for all developed programs are shown.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1 INTRODUÇÃO

1.1 CONSIDERAÇÕES INICIAIS

O estudo das redes neurais artificiais é multidisciplinar. É uma área do conhecimento de interesse de neurocientistas, matemáticos, físicos, estatísticos, cientistas da computação e engenheiros. Suas aplicações se estendem por vários campos distintos, como aproximações de funções, modelagem, previsão de séries temporais e classificação de padrões [1].

O avanço da microeletrônica tem propiciado desde os anos 80 um renovado interesse pelo assunto, já que era difícil realizar a simulação de redes neurais nos computadores das décadas anteriores.

Nos dias de hoje há uma profusão de equipamentos que fazem uso de microcontroladores. Tais equipamentos vão de brinquedos a robôs autônomos, passando por telefones públicos, celulares e eletrodomésticos. Vários destes aparelhos contam com unidades de processamento muito simples que possuem limitações de precisão, por motivos como a necessidade de baixo custo de produção e baixo consumo de energia. Poucos microprocessadores suportam tipos de dados em ponto-flutuante e operações em nível de instrução.

Devido à capacidade finita de representação de números em qualquer tipo de hardware digital, em qualquer tipo de processamento sempre há alguma perda de precisão. Essa perda é um fator importante que deve ser levada em consideração ao se trabalhar com um hardware de ponto fixo.

1.2 OBJETIVOS DO PRESENTE TRABALHO

O presente trabalho tem como objetivo o estudo do funcionamento de redes neurais com aprendizado assistido aplicadas à resolução de determinados problemas. Possui também como meta estabelecer uma comparação entre a precisão obtida nos treinamentos e testes de algoritmos simuladores de redes neurais utilizando cálculos em ponto flutuante em computadores pessoais ou em ponto fixo em hardware autônomo.

O projeto faz uma abordagem discursiva sobre o modelo clássico Multi-Layer Perceptron de redes neurais artificiais e apresenta os dois paradigmas principais de aprendizado, detendo-se na descrição detalhada da regra de aprendizado assistido “backpropagation”.

As primeiras redes neurais desenvolvidas simulam problemas bastante simples, intencionalmente voltados à análise do funcionamento dos algoritmos desenvolvidos em linguagem C. O último caso desenvolvido possuiu um nível de complexidade maior, exigindo a estruturação de um algoritmo responsável por uma organização matricial de uma rede neural.

O projeto conta com o auxílio do compilador DEV C/C++, do software de interfaceamento entre o computador e o hardware TMS320VC5510® denominado Code Composer Studio® versão 3.1 e de manipulações vetoriais e elaboração de gráficos de convergência de erro efetuados na ferramenta Matlab® 7.

O trabalho está organizado da seguinte maneira:

- O capítulo 2 apresenta uma revisão bibliográfica sobre os conceitos de redes neurais artificiais, algoritmo “backpropagation”, descrição da arquitetura e funcionamento do hardware de ponto fixo TMS320VC5510® da Texas Instruments e utilização do Code Composer Studio®;
- O capítulo 3 descreve a metodologia utilizada neste trabalho;
- No capítulo 4 há uma exposição dos problemas que foram resolvidos com o uso de redes neurais neste trabalho, seus resultados e análises;
- O capítulo 5 traz as considerações finais do presente trabalho.

2 REVISÃO BIBLIOGRÁFICA

2.1 REDES NEURAIS ARTIFICIAIS

Uma rede neural artificial é um sistema de processamento adaptativo e muitas vezes não-linear. É um sistema paralelo e distribuído caracterizado pela presença de unidades básicas de processamento, as quais calculam algum tipo de função matemática. Sua característica intrínseca é a possibilidade da alteração de parâmetros internos de maneira que a rede aprenda a desenvolver respostas otimizadas a problemas apresentados a ela. Essa área do conhecimento, também chamada de conexionismo, se iniciou com as pesquisas de McCulloch e Pitts em 1943, ressurgindo no campo das pesquisas na década de 80. A origem desse ramo de pesquisa se deu através da busca incessante em entender e conseguir reproduzir o funcionamento do cérebro humano, isto é, sua lógica de processamento e estrutura interna [2].

2.1.1 O NEURÔNIO BIOLÓGICO.

Rede Neural Artificial (RNA) é uma lógica de processamento de dados que se baseia intrinsecamente no funcionamento fisiológico do cérebro humano. Considerado o mais interessante processador do universo, o cérebro é uma máquina composta por aproximadamente dez bilhões de neurônios, sendo o responsável por controlar todas as funções e todos os movimentos do organismo.

Os neurônios são interconectados através das sinapses, as quais permitem a intercomunicação entre eles através de um processo de liberação e reconhecimento de substâncias químicas em diferentes concentrações, tais como íons de Sódio (Na^+) e Potássio (K^+). Esta comunicação é feita através de impulsos. Ao receber um impulso, o neurônio o processa e, dependendo do tipo e da quantidade da substância contida no impulso recebido, dispara outro impulso com um neurotransmissor que flui do corpo celular para o axônio. A freqüência destes pulsos é controlada aumentando ou diminuindo a polaridade na membrana pós sináptica.

Os neurônios podem ser subdivididos entre os seguintes componentes:

- Dendritos – recebem os estímulos transmitidos por outros neurônios;

- Corpo ou Somma – coleta e combina informações vindas de outros neurônios que passam pelos dendritos. Pode ser considerado a unidade de processamento de dados do neurônio.
- Axônio – é constituído por uma fibra tubular e possui a função de transmitir impulsos a outros neurônios.

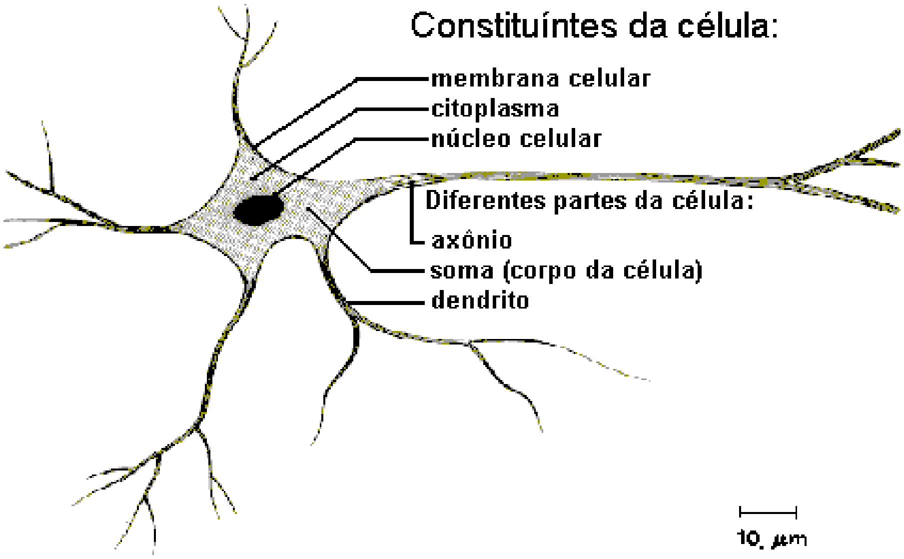

FIGURA 2.1 – ESQUEMÁTICO DE UM NEURÔNIO BIOLÓGICO [12].

2.1.2 HISTÓRICO DAS REDES NEURAIS

Em 1943, Warren McCulloch e Walter Pitts publicaram um trabalho apresentando o primeiro modelo artificial de um neurônio biológico. Este artigo intitulado “A Logical Calculus of the Ideas Immanent in Nervous Activity” (Um Cálculo Lógico das Idéias Inerentes à Atividade Nervosa) apresenta novas idéias sobre máquinas de estados finitos, elementos de decisão de limiar lineares e representações lógicas de várias formas de comportamento e memória, além de apresentar uma discussão de redes lógicas de nodos, ou seja, unidades básicas de processamento de sinais distribuídos em uma rede paralelizada (modelo MCP) [1].

O trabalho de McCulloch e Pitts não focava técnicas de aprendizado para o modelo. Este aprendizado veio a ser objeto de estudo somente em 1949 por Donald Hebb em sua obra chamada “The Organization of Behaviour” (A Organização do Comportamento). Sua teoria propunha que a explicação do aprendizado em neurônios biológicos é baseada no reforço das ligações sinápticas entre nodos excitados. Posteriormente Widrow e Hoff sugeriram uma regra de aprendizado baseada no método do gradiente para minimizar o erro na saída de um neurônio, chamada de regra delta [1].

Marvin Minsky em 1951 construiu o primeiro neurocomputador, denominado Snark. Este operava a partir de um ponto de partida técnico, fazendo o ajuste automático de seus pesos.

Em uma conferência de verão realizada em 1956 no Dartmouth College, localizado em Hanover, no estado de New Hampshire nos Estados Unidos foram estabelecidos os dois paradigmas da inteligência artificial: a abordagem simbólica e a conexionista. A abordagem simbólica envolve métodos agora classificados como aprendizado de máquina, caracterizados por formalismo e análise estatística. A abordagem conexionista envolve desenvolvimento iterativo ou aprendizado baseado em dados empíricos. É nessa abordagem que se encaixa o conhecimento sobre Redes Neurais.

Rosenblatt foi o criador do modelo perceptron. Em 1958 demonstrou com sua pesquisa que seus nodos MCP, ao serem acrescidos de sinapses ajustáveis, poderiam ser treinados para classificar certos tipos de padrões. Ele descreveu uma topologia de Rede Neural Artificial, estruturas de ligação entre os nodos e um algoritmo de treinamento para essa rede. Sua rede se comportava basicamente como um classificador de padrões, mas possuía a limitação de funcionar apenas para padrões linearmente separáveis. Sua motivação era conseguir projetar RNAs capazes de resolver problemas complexos sem a necessidade de regras para descrevê-los [1].

Minsky e Papert publicaram um artigo em 1969 chamando a atenção para o fato de que o perceptron não era capaz de resolver problemas não-linearmente separáveis. Somente conseguia resolver problemas cuja solução pode ser obtida dividindo-se o espaço de entrada em duas regiões através de uma reta [1]. Eles afirmaram que o problema do crescimento explosivo, tanto de espaço ocupado quando do tempo necessário para a resolução de problemas complexos, afetaria de forma significativa as RNAs. Argumentaram também que seria impossível criar um algoritmo de aprendizado para uma rede com mais de uma camada de neurônios.

Este artigo teve uma repercussão extremamente negativa para o campo de estudo conexionista, causando um alto declínio no número de pesquisadores atuantes neste ramo.

Em 1982, John Hopfield, renomado físico de reputação mundial, publicou um artigo mostrando a relação entre as redes recorrentes auto-associativas e sistemas físicos, o que abriu espaço para a utilização de teorias correntes da Física para estudar tais modelos. Outra descoberta importante desta década ocorreu em 1986, com a publicação de um artigo intitulado “Learning representations by back-propagating errors” (Representações de aprendizado através de retropropagação de erros) por Rumelhart, Hinton e Williams. Neste artigo foi afirmado que era possível estabelecer um algoritmo de aprendizado para redes com mais de uma camada de neurônios. Assim foi estabelecida a origem do algoritmo “backpropagation”.

Atualmente, a comunidade internacional apresenta grande interesse nos campos de pesquisa utilizando redes neurais artificiais. Esse interesse é possível graças ao grande avanço na tecnologia, principalmente da microeletrônica, que permite a construção de modelos físicos de redes neurais bastante complexos, e possui uma motivação grande: o fato de a escola simbolista ainda não ter conseguido resolver problemas considerados simples para o ser humano [1].

2.1.3 FUNCIONAMENTO DE REDES NEURAIS ARTIFICIAIS

O funcionamento de uma rede neural geralmente envolve três etapas: treinamento, validação e utilização. Seus parâmetros variam durante sua operação na fase de treinamento. Passada essa fase, seus parâmetros são fixados e o sistema é posto a prova para verificar se ele é capaz de resolver determinado problema, esta é a chamada fase de validação. Atingido um percentual aceitável de acerto a rede pode ser empregada na resolução do problema para o qual foi treinada. O banco de dados de entradas e saídas da rede tem uma importância fundamental, pois possui a informação necessária para atingir o ponto ótimo de funcionamento da rede. A característica da não-linearidade do sistema se apresenta como uma flexibilidade para executar qualquer mapeamento de relações de entradas/saídas, fazendo com que essas redes sejam também consideradas como mapeadores universais [4].

Um dos estilos mais usuais de computação neural pode ser descrito da seguinte maneira: é apresentada uma entrada para a rede, a saída obtida da rede é então comparada com um valor de saída desejado. A diferença entre a saída desejada e a saída obtida do sistema é considerada como um erro. A informação deste erro é retropropagada no sistema, ajustando seus parâmetros seguindo uma regra pré-determinada (regra de aprendizado). Este processo se repete até que a rede obtenha uma performance aceitável. Isso nos mostra o principal benefício de uma rede neural: se existe a disponibilidade de um banco de dados rico em quantidade e o problema a se resolver é de difícil modelagem matemática, então o uso de redes neurais é uma boa solução. Em contrapartida, é necessário o uso de heurística na definição das características da rede, como número de neurônios em cada camada da rede, número de interconexões e número de camadas, o que traz dificuldade no refinamento da solução.

Atualmente, redes neurais estão cada vez mais se tornando a tecnologia empregada na resolução de problemas em diversas aplicações, como reconhecimento de padrões, identificação e controle de sistemas e predição de séries temporais.

2.1.4 O MODELO MCCULLOCH E PITTS

O modelo desenvolvido por McCulloch e Pitts funciona baseado em uma simplificação do funcionamento do neurônio biológico. Esse modelo possui n terminais de entrada ($x_1, x_2, ..., x_n$), pesos acoplados a cada terminal ($w_1, w_2, ..., w_n$), uma unidade de processamento central e um terminal de saída. Funciona da seguinte maneira: cada sinal apresentado em cada terminal é multiplicado pelo valor de seu peso correspondente. Em seguida, é feito um somatório de todos os sinais multiplicados por seus respectivos pesos. O valor dessa soma é injetado em uma função de ativação, sendo essa uma função de limiar, isto é, se o valor da soma atinge um valor limite θ (threshold), a unidade de processamento dispara um sinal de pulso no terminal de saída de acordo com a equação abaixo:

$$\sum_{i=1}^{n} x_i \cdot w_i \geq \theta \qquad (2.1)$$

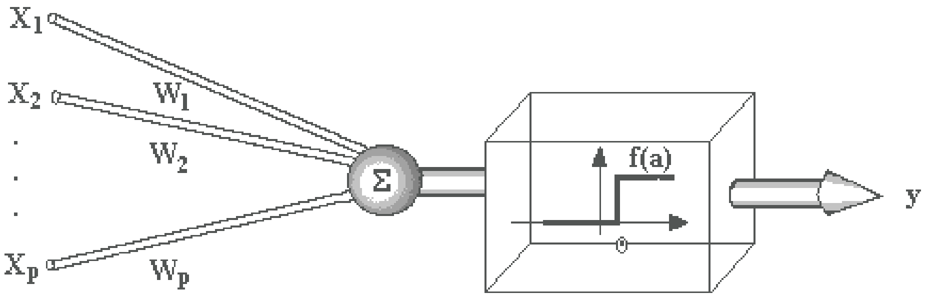

FIGURA 2.2 – MODELO DE NEURÔNIO DESENVOLVIDO POR MCCULLOCH E PITTS [12].

Cabe lembrar que de acordo com o modelo descrito por McCulloch e Pitts, os neurônios de cada camada da rede disparam seus sinais sincronamente.

Utilizando este modelo, vários cientistas derivaram outras funções de ativação para as unidades de processamento da rede, entre elas podem-se destacar:

- Função linear – definida pela equação $y = \alpha \cdot x$, onde alpha é um número real que define a saída linear para os valores de entrada, x é a entrada e y é a saída.

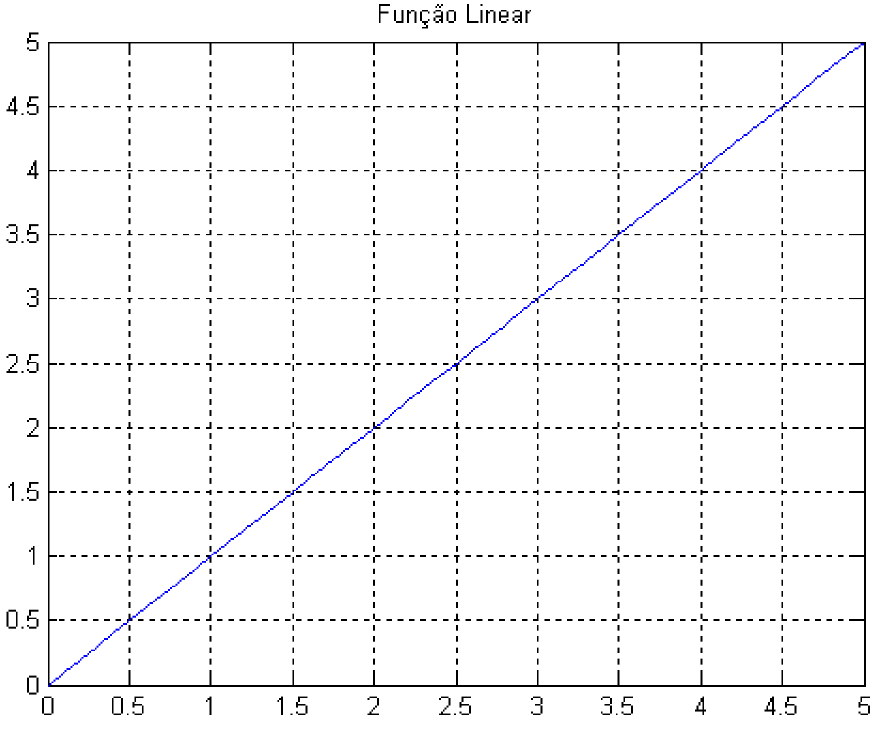

FIGURA 2.3 – FUNÇÃO LINEAR.

- Função "Step" (Degrau) – produz uma saída *y* para valores de entrada maiores do que 0 e uma saída –*y* para valores de entrada menores ou iguais a 0.

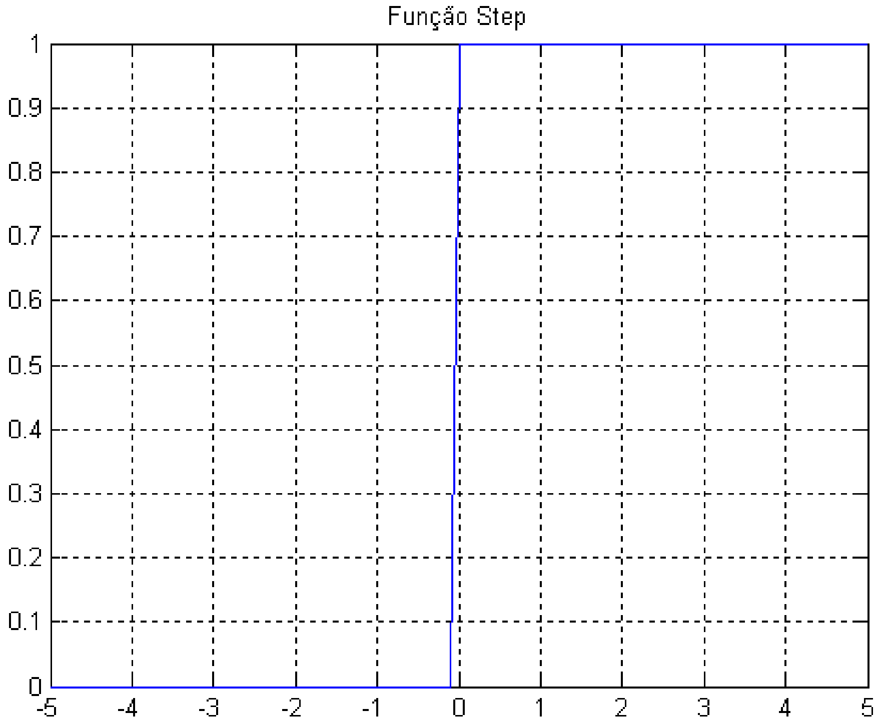

FIGURA 2.4 – FUNÇÃO "STEP" OU DEGRAU.

- Função Sigmóide – também chamada de função logística, é uma função monotônica, real, limitada e diferenciável. Possui apenas um máximo e um mínimo. Seu valor tende para zero quando a entrada tende para $-\infty$ e tende para um quando sua entrada tende para $+\infty$. Pode ser descrita pela equação abaixo:

$$Y(x) = \frac{1}{1 - e^{-\alpha \cdot x}} \tag{2.2}$$

onde,

x = entrada da função;

alpha = parâmetro que determina a suavidade da curva.

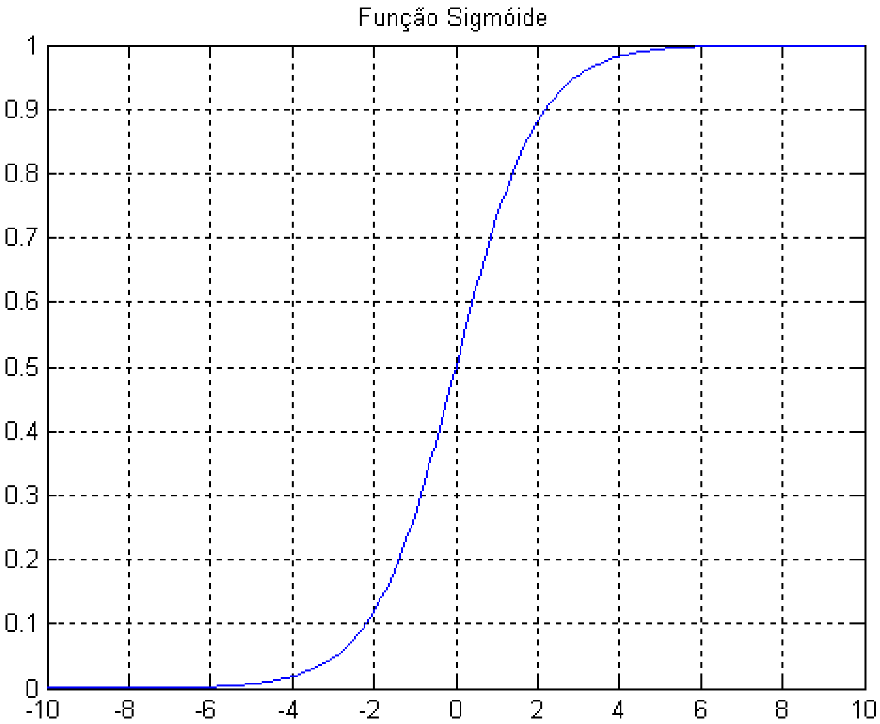

FIGURA 2.5 – GRÁFICO BIDIMENSIONAL DE UMA FUNÇÃO SIGMÓIDE.

- Função tangente hiperbólica – é uma função com o mesmo comportamento da sigmóide. Seu valor tende para -1 quando a entrada tende para $-\infty$ e tende para 1 quando sua entrada tende para $+\infty$. Pode ser descrita pela equação abaixo:

$$\tanh x = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{2.3}$$

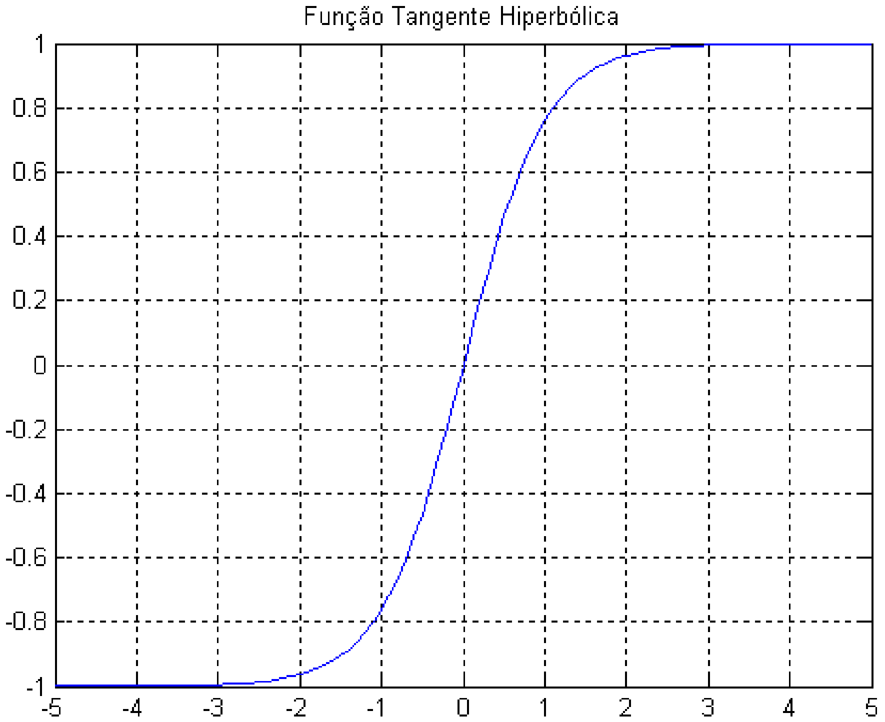

FIGURA 2.6 – GRÁFICO BIDIMENSIONAL DE UMA FUNÇÃO TANGENTE HIPERBÓLICA.

2.1.5 REDES MULTI-LAYER PERCEPTRON

Este tipo de rede consiste num arranjo de neurônios artificiais segundo o modelo de McCulloch e Pitts. É constituída por uma camada de entrada, onde são recebidos os sinais de entrada, por uma camada de saída, que tem por função retornar a saída da rede e por camadas que não possuem acesso direto externo, as quais são denominadas camadas escondidas (não necessariamente presentes, como é o caso do perceptron, que é uma rede constituída apenas pela camada de entrada e pela camada de saída). Cada conexão entre neurônios possui um peso, o qual é adaptado durante o processo de aprendizado.

Em cada unidade ocorre o processamento do somatório, o qual é utilizado como entrada para uma função. Esta geralmente é a função sigmóide ou a função tangente hiperbólica.

O importante a ressaltar sobre redes MLP é que, quando possuem duas camadas escondidas e quantidade de neurônios suficiente, podem mapear praticamente qualquer vetor de entradas/saídas. Até mesmo redes com apenas uma camada escondida conseguem aproximar com bastante precisão o mapeamento de vetores de entrada/saída.

Cabe ao projetista somente escolher quantas camadas e quantos neurônios em cada camada utilizar [4].

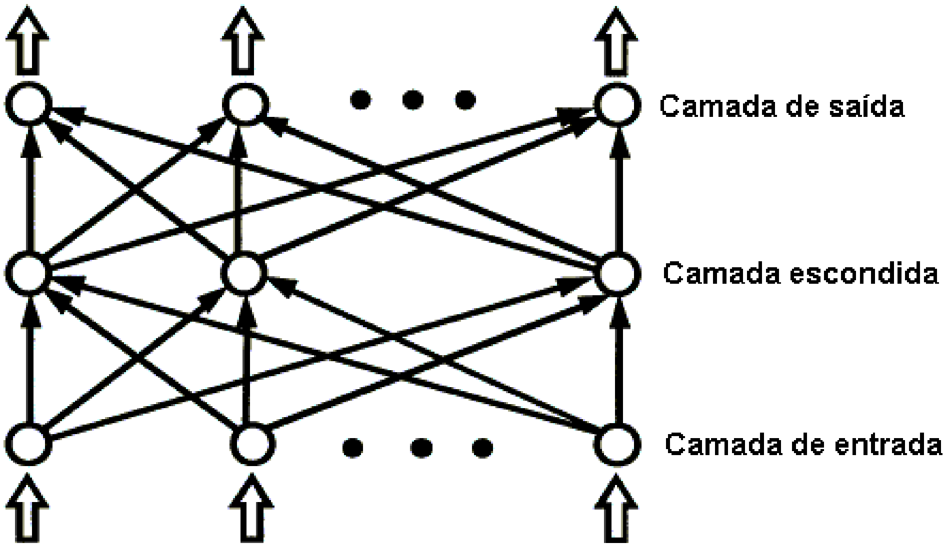

FIGURA 2.7 – ESQUEMA REPRESENTANDO UMA REDE MULTI-LAYER PERCEPTRON [12].

2.1.6 TREINAMENTO EM REDES MULTI-LAYER PERCEPTRON

Treinamento de uma rede neural é o conjunto de procedimentos para adaptar seus parâmetros, de forma que a mesma possa aprender a solucionar um determinado problema. Existem dois tipos de treinamento: o supervisionado e o não-supervisionado.

O treinamento supervisionado consiste na utilização de pares de entrada/saída desejada. A cada padrão de entrada apresentado à rede faz-se uma comparação entre a saída desejada e a saída calculada, comparação que define o erro da resposta atual. Os pesos e “bias” são então recalculados de forma a minimizar esse fator de erro. O termo “bias” é um termo presente no somatório de entrada de cada neurônio. Este serve para aumentar os graus de liberdade da rede, permitindo uma melhor adaptação ao conhecimento a ela fornecido. Esta minimização é feita incrementalmente. A cada etapa de treinamento faz-se pequenos ajustes nos pesos para que estes caminhem para a solução desejada. Como exemplos podem ser citados os treinamentos com a regra delta e sua generalização para redes com mais de uma camada, o algoritmo “backpropagation”.

No aprendizado não-supervisionado, apenas os padrões de entrada são disponibilizados para a rede. Assim que a rede estabelece uma relação de harmonia com as regularidades estatísticas dos dados de entrada, desenvolve-se uma habilidade de formar representações internas para codificar características da entrada e criar novas classes ou grupos automaticamente [1]. Este processo é possível somente com a presença de redundância nos dados de entrada. Como exemplos deste tipo de treinamento podem ser citados o aprendizado hebbiano e o aprendizado por competição.

Atualmente, vários tipos de algoritmos são empregados no treinamento de redes MLP, a maioria do tipo supervisionado. Estes podem ser estáticos, isto é, só alteram os valores dos pesos ou podem ser dinâmicos, podendo além de alterar os pesos reduzir ou aumentar a rede no que diz respeito a número de camadas, neurônios e conexões.

O algoritmo “backpropagation” é um processo de aprendizado supervisionado, que pode ser subdividido em duas etapas. A primeira denominada “forward” consiste no estabelecimento do valor da saída da rede ao se apresentar uma entrada. A segunda denominada “backward” utiliza a comparação entre a saída desejada e a saída obtida para atualizar os pesos das conexões da rede. Esta fase possui sentido inverso à fase “forward”, pois atualiza primeiramente os pesos das conexões dos neurônios da camada de saída, em seguida os pesos da última camada escondida e assim sucessivamente até chegar à primeira camada escondida [4].

Para possibilitar a atualização dos pesos, é necessário estabelecer uma função de erro a ser minimizada. Neste caso, a função utilizada é a soma dos erros quadráticos representada pela equação abaixo:

$$J = \sum_{p} \left(d_p - y_p \right)^2 \tag{2.4}$$

onde,

J = função de erro;

d_p = saída desejada do padrão p;

y_p = saída calculada do padrão p.

E cada peso w_i na iteração k é ajustado de acordo com a equação a seguir:

$$w_i(k+1) = w_i(k) - \eta \cdot \frac{\partial}{\partial w_i} J(k) \tag{2.5}$$

onde,

$\frac{\partial}{\partial w_i}J(k)$ = gradiente da superfície de erro;

η = taxa de aprendizado, chamada também de “step size”.

Para uma função de ativação não-linear e diferenciável, uma estimativa razoável para computar o gradiente do erro em cada iteração para cada padrão p pode ser descrita na equação a seguir:

$$\frac{\partial J}{\partial w_i} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial net}\frac{\partial}{\partial w_i}net = -(d-y)f'(net)x_i = -\varepsilon f'(net)x_i \tag{2.6}$$

onde,

$\frac{\partial}{\partial w_i}J(k)$ = gradiente da superfície de erro;

$\frac{\partial J}{\partial y}$ = derivada parcial da função erro com relação ao estado do neurônio (sinal de saída);

ε = diferença entre a saída desejada e a saída calculada;

f’(net) = derivada da função de ativação do neurônio w_i.

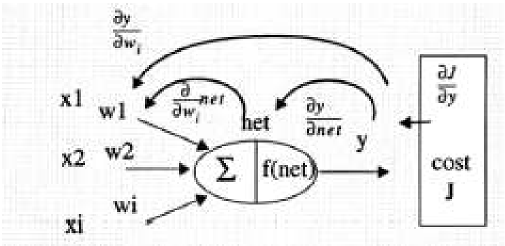

FIGURA 2.8 – REPRESENTAÇÃO DA ESTIMATIVA UTILIZADA PARA COMPUTAR O GRADIENTE DE ERRO [4].

A estimativa acima funciona perfeitamente para a camada de saída da rede neural. Para utiliza-la nas camadas escondidas, é necessário fazer uma adaptação. O único problema é conseguir fazer a computação da derivada parcial da função erro com relação ao estado do neurônio. Para um neurônio da camada de saída, essa expressão corresponde ao valor do erro ε. Para um neurônio na camada escondida, é necessário somar todas as componentes de erro dos neurônios da camada posterior que são conectados a ele. Assim, para um neurônio na i-ésima camada escondida, tem-se:

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial y_i}\frac{\partial y_i}{\partial net_i}\frac{\partial}{\partial w_{ij}}net_i \tag{2.7}$$

e

$$\frac{\partial J}{\partial w_{ij}} = -x_j f'(net_i)\left[\sum_k \varepsilon_k f'(net_k) w_{ki}\right] \tag{2.8}$$

onde,

$-x_j f'(net_i)$ = estimativa da derivada parcial com relação à ativação local dos pesos;

$\left[\sum_k \varepsilon_k f'(net_k) w_{ki}\right]$ = estimativa da derivada parcial da função erro com relação ao estado do neurônio.

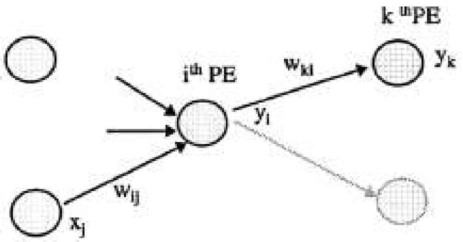

FIGURA 2.9 – ESQUEMÁTICO REPRESENTANDO A ATUALIZAÇÃO DOS PESOS DA I-ÉSIMA CAMADA ESCONDIDA [4].

A atualização do termo “bias” de cada neurônio é um pouco mais simples que a atualização dos pesos. A regra é a mesma, somente desconsidera-se a participação dos sinais de entrada do neurônio na fórmula do algoritmo “backpropagation”.

2.2 HARDWARE TMS320VC5510®

O hardware utilizado neste projeto é a placa de desenvolvimento DSP Starter Kit que possui como processador o modelo TMS320VC5510® da Texas Instruments. Este hardware é acompanhado pelos documentos Quick Starter Guide, Technical Reference, Customer Support Guide, softwares C5510 DSK Code Composer Studio™ v3.1 IDE e Matlab® 7 Trial, fonte e cabo de alimentação +5V e cabo USB.

2.2.1 CARACTERÍSTICAS DO PROCESSADOR

O processador da série C5510 é projetado para consumo baixo de potência, performance otimizada e alta densidade de código. Sua arquitetura multiplicação-acumulação dual (MAC) possibilita duas vezes a eficiência de ciclo para computar produtos de vetores, operação fundamental em processamento digital de sinais [9].

Algumas de suas características essenciais estão listadas abaixo:

- Fila de armazenamento de dados temporária (“buffer queue”) de 64 bytes que funciona como uma memória cache de programa;
- Duas unidades MAC de 17 por 17 bits executando operações multiplicação-acumulação duplas em um único ciclo;
- Uma unidade lógica aritmética (ALU) de 40 bits que efetua aritmética de alta precisão e operações lógicas com uma unidade lógica aritmética adicional de 16 bits efetuando operações paralelas à unidade central;
- Quatro acumuladores de 40 bits para armazenamento de resultados computacionais com o objetivo de diminuir o acesso à memória;
- Oito registradores auxiliares estendidos para endereçamento de dados mais quatro registradores temporários de dados para diminuir as exigências de processamento de dados;
- Modo de endereçamento circular que suporta até cinco buffers circulares;
- Operações de repetição de instrução única e de repetição de bloco para dar suporte a laços sem “overhead”.

2.2.2 ARQUITETURA TMS320VC5510

A unidade central de processamento C5510 consiste de quatro unidades de processamento: uma unidade de instrução de buffer (IU), uma unidade de fluxo de programa (PU), uma unidade de fluxo de endereçamento de dados (AU) e uma unidade de computação de dados (DU). Essas unidades estão conectadas a 12 barramentos de endereçamento e dados de acordo com a figura abaixo:

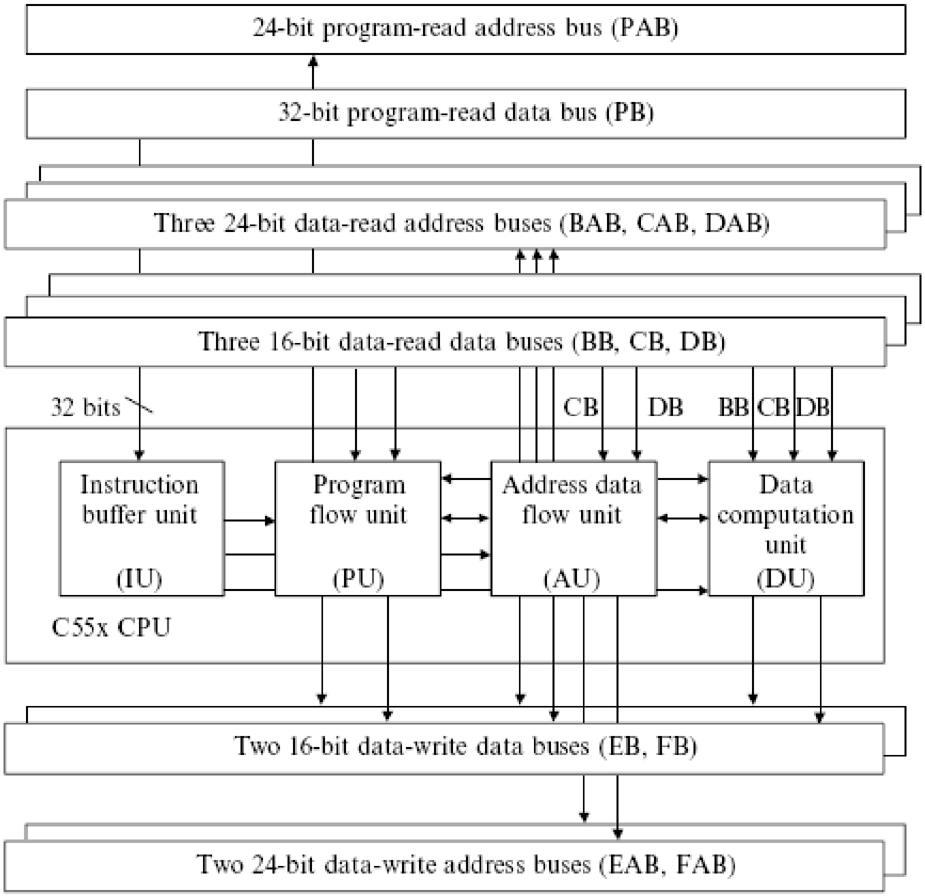

FIGURA 2.10 – DIAGRAMA DE BLOCO DA UNIDADE CENTRAL DE PROCESSAMENTO C5510 [9].

A unidade de instrução de buffer (IU) recebe instruções da memória e envia para a CPU. O conjunto de instruções do C5510 pode variar em tamanho. Instruções simples utilizam apenas oito bits, enquanto instruções mais complicadas podem conter até 48 bits. Para cada ciclo de relógio, a IU pode buscar até quatro bytes de código de programa através de seu barramento de leitura de programa de 32 bits. Ao mesmo tempo a IU pode decodificar até seis bytes de programa. Após buscar quatro bytes de programas, os coloca em um buffer de instrução de 64 bytes. Ao mesmo tempo a lógica de decodificação traduz uma instrução de um até seis bytes que foi armazenada anteriormente no decodificador de instrução. A instrução decodificada é em seguida mandada para a PU, AU ou DU [9].

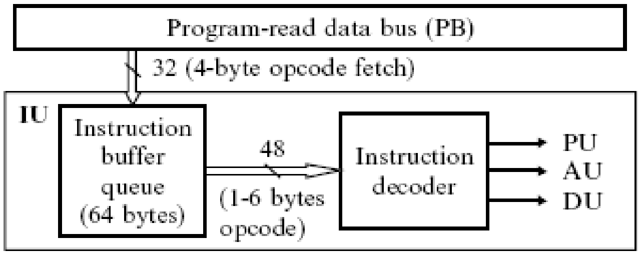

FIGURA 2.11 – DIAGRAMA DE BLOCO DA UNIDADE DE BUFFER DE INSTRUÇÃO DO C5510 [9].

Esta unidade aumenta a eficiência de execução do programa, pois mantém o fluxo de instruções entre as quatro unidades internas da CPU. Uma opção interessante é utilizar essa unidade para manter um segmento do código dentro de um laço, para realizar a execução do programa várias vezes. Isso diminui o acesso de memória em busca de código adicional, o que reduz o consumo de energia. Outra vantagem é que a IU pode armazenar instruções múltiplas que podem ser utilizadas em conjunto com controle de fluxo de programa condicional. Isso minimiza o "overhead" causado por descontinuidades no fluxo do programa como chamadas condicionais.

A unidade de fluxo de programa (PU) controla o fluxo de execução de programa. A unidade PU consiste em um contador de programa, quatro registradores de status, um gerador de endereçamento de programa e uma unidade de proteção de "pipeline". O contador de programa rastreia a execução de programa C5510 a cada ciclo de execução. O gerador de endereçamento do programa produz um endereço de 24 bits que cobre 16 Mbytes de espaço de programa. Como a maioria das instruções é executada em seqüência, o C5510 utiliza estrutura "pipeline" para melhorar sua eficiência. Porém, instruções como saltos, chamadas, retornos, execução condicional e interrupção causam uma troca de endereçamento de programa não-sequencial. A PU usa uma unidade de proteção de “pipeline” para prevenir o fluxo de programa de qualquer vulnerabilidade causada por uma execução não-sequencial.

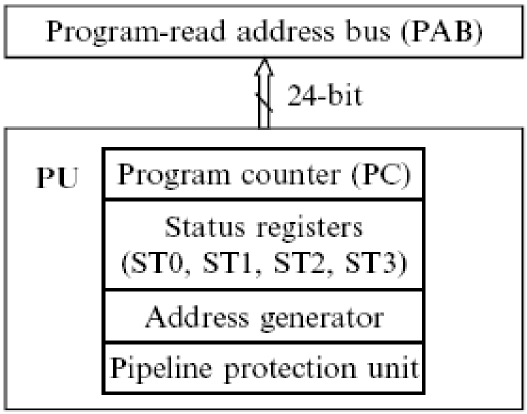

Figura 2.12 – Diagrama de Bloco Simplificado da unidade de fluxo de programa do C5510 [9].

A unidade de fluxo de endereçamento de dados (AU) gera os espaços de endereçamento para leitura e escrita de dados. Consiste em oito registradores auxiliares de 23 bits (XAR0 a XAR7), quatro registradores temporários de 16 bits (T0 a T3), um ponteiro de coeficientes de dados estendidos de 23 bits (XCDP) e um ponteiro de pilha estendido de 23 bits (XSP). Possui também uma unidade lógica aritmética adicional de 16 bits para operações aritméticas simples. Os registradores temporários podem ser utilizados para minimizar a necessidade de acessos à memória [9].

E, finalmente, a unidade de computação de dados (DU). Essa unidade faz o processamento de dados para a maioria das aplicações em hardware C5510. É constituída por um par de unidades MAC, uma unidade lógica aritmética de 40 bits, quatro acumuladores de 40 bits (AC0, AC1, AC2 e AC3), um “barrel shifter”, isto é, um deslocador bit-a-bit e por uma lógica de controle de arredondamento e saturação. Em cada ciclo, cada unidade MAC consegue executar uma multiplicação de 17 bits e uma operação de adição ou subtração com uma opção de saturação em 40 bits. A unidade lógica aritmética consegue executar operações de aritmética, lógica, arredondamento e saturação em 40 bits usando os quatro acumuladores. Pode ser utilizada para executar duas operações de aritmética em 16 bits em ambas as partes mais alta e mais baixa do acumulador ao mesmo tempo. A unidade de instrução de buffer (IU) pode mandar valores imediatos para a ALU como dados. Existe também uma comunicação entre a ALU e os registradores das unidade AU e PU. O “barrel shifter” consegue fazer um deslocamento de quantos bits forem necessários em um único ciclo, no alcance de 2^{-32} a 2^{31}.

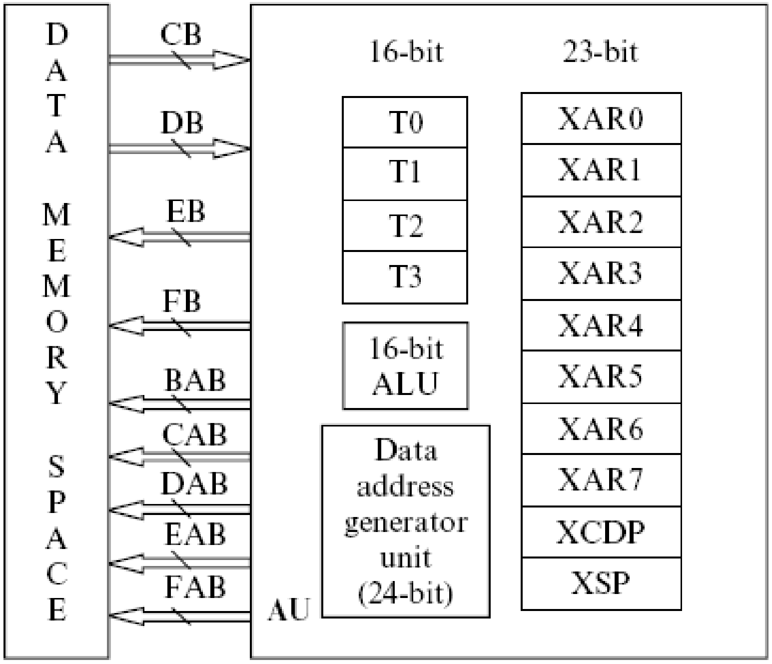

FIGURA 2.13 – DIAGRAMA DE BLOCO SIMPLIFICADO DA UNIDADE DE FLUXO ENDEREÇAMENTO DE DADOS DO C5510 [9].

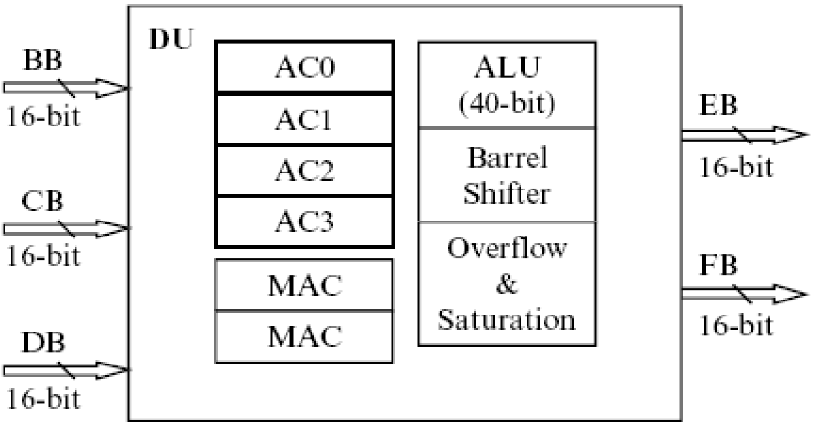

FIGURA 2.14 – DIAGRAMA DE BLOCO SIMPLIFICADO DA UNIDADE DE COMPUTAÇÃO DE DADOS DO C5510 [9].

O hardware TMS320VC5510 possui um barramento de dados de programa de 32 bits, cinco barramentos de dados de 16 bits e seis de endereçamento de 24 bits. Os barramentos de programa incluem um barramento de 32 bits de dados de leitura do programa (PB) e um barramento de endereço de leitura do programa de 24 bits (PAB). O PAB carrega o endereço de memória do programa para ler o código do espaço do programa. O espaço de programa endereçável está contido no intervalo entre 0x000000 até 0xFFFFFF (o formato de endereçamento é em base hexadecimal). O PB transfere quatro bytes de código de programa para a IU em cada ciclo do relógio.

Os barramentos de dados consistem em três barramentos de leitura de dados de 16 bits (BB, CB e DB) e três barramentos de endereçamento de leitura de dados de 24 bits (BAB, CAB e DAB). Essa arquitetura suporta três leituras simultâneas de dados da memória ou do espaço entrada/saída. Os barramentos CB e DB podem mandar dados para as unidades PU, AU e DU, enquanto o barramento BB somente acessa a unidade DU. As operações de escrita de dados são efetuadas utilizando dois barramentos de escrita de dados (EB e FB) e dois barramentos de endereçamento de escrita de dados (EAB e FAB). Para uma escrita de dados de 16 bits, somente o EB é utilizado. Uma escrita de dados de 32 bits usa ambos os EB e FB em um ciclo. Os barramentos de endereçamento de escrita de dados possuem o mesmo alcance de endereçamento de 24 bits. Como o acesso de dados utiliza uma unidade de uma palavra (2 bytes), o espaço para a memória de dados se torna 23 bits, endereçável do endereço 0x000000 até 0x7FFFFF [9].

A arquitetura do hardware é construída em torno destes 12 barramentos. Os barramentos de programa carregam os códigos de instrução e os operandos imediatos a partir da memória do programa, enquanto os barramentos de dados conectam várias unidades. Esta arquitetura maximiza o poder de processamento, pois mantém as estruturas de barramento de memória separadas, possibilitando uma execução à máxima velocidade possível.

2.2.3 UTILIZAÇÃO DO KIT DSK

Para efetuar a transição dos programas escritos em linguagem C para a memória do hardware, foi utilizado o programa chamado Code Composer Studio® v.3.1 (CCS). Este programa possui uma interface simples, intuitiva e de fácil utilização. Ele possibilita a criação, edição, depuração e análise de programas de processamento digital de sinais. Seu ambiente de trabalho suporta a utilização de vários processadores da linha TMS320 da Texas Instruments. Para projetar aplicações, o CCS disponibiliza um gerenciador que manuseia as tarefas de programação. Para fins de depuração, ele disponibiliza “breakpoints”, acompanhamento de variáveis escolhidas pelo usuário, visualização da memória, registradores e pilha, análise gráfica e a capacidade de mostrar instruções em código em C misturadas com instruções em “disassembled”.

Uma característica importante do CCS é sua habilidade em criar e manusear projetos extensos a partir de um ambiente gráfico de interface com o usuário.

Um projeto do CCS serve para armazenar toda a informação necessária para, através de um processo chamado “build”, escrever um arquivo .out que é transferido para a memória do hardware. Informação esta constituída por arquivos de documentação do projeto, outros projetos dependentes deste, configurações DSP/BIOS, arquivos gerados, bibliotecas de funções inclusas e códigos-fonte em linguagem C ou Assembly.

3 METODOLOGIA E DESENVOLVIMENTO DO PROJETO

Primeiramente, este capítulo traz uma apresentação da metodologia utilizada durante a execução do projeto. Em seguida, é mostrado o desenvolvimento deste de acordo com as seguintes etapas: descrição da implementação e testes, resultados obtidos e suas respectivas análises.

3.1 METODOLOGIA DO PROJETO

De acordo com os princípios teóricos estabelecidos anteriormente sobre os conceitos de redes neurais artificiais, buscou-se implementar o modelo clássico do algoritmo de aprendizado “backpropagation” baseado na regra delta de Widrow-Hoff em programas simulando redes neurais MLP em linguagem C. Em seguida, testou-se a viabilidade de algumas implementações com a utilização do software Code Composer Studio® versão 3.1 em conjunto com o hardware TMS320VC5510® disponibilizado no DSP Starter Kit.

As redes escolhidas para realizar tal teste são redes simples, cujos resultados esperados são conhecidos, de modo a facilitar a comparação entre a implementação em hardware de ponto fixo com a implementação em um computador doméstico (PC).

Primeiramente foi implementado um único neurônio artificial para realizar a representação de uma porta lógica “AND” (E). Esse problema é linearmente separável e passível de ser resolvido por apenas um neurônio. Logo essa implementação, apesar de ser a mais simples possível, é capaz de realizar a tarefa.

Com os objetivos de testar o algoritmo “backpropagation” para camadas escondidas e resolver problemas não-linearmente separáveis, foram implementadas redes um pouco mais complexas. Os problemas escolhidos foram o da porta lógica “XOR” (OU-EXCLUSIVO) e do codificador binário. Em seguida implementou-se estes exemplos no hardware de ponto fixo e observou-se o seu comportamento.

As etapas anteriores mostraram que o algoritmo funciona e que a implementação no hardware é viável. Mostraram também que os programas escritos até então não eram reutilizáveis, pelo fato de que eram escritos para uma implementação específica (um programa para cada problema a ser resolvido). Assim percebeu-se a necessidade de se escrever um algoritmo generalizado que pudesse ser utilizado para a resolução de diversos problemas.

Surgiu então a idéia de utilizar este novo programa para implementar uma rede capaz de solucionar um problema em que nem todas as entradas possíveis são conhecidas. Para tal escreveu-se uma rede matricial para realizar a função de classificação de padrões.

Para testar este algoritmo, utilizou-se o banco de dados de entradas e saídas referente à classificação das três espécies de plantas do gênero *Íris*. Este banco de dados foi criado por Fisher em 1936 em seu trabalho intitulado "The use of multiple measurements in taxonomic problems". Este conjunto contém três classes, com 50 amostras de cada classe. Cada classe se refere a um tipo de Íris (setosa, versicolor e virginica). A classe setosa é linearmente separável das outras duas; as demais não são linearmente separáveis uma da outra.

Como etapa final na fase de implementação, transferiu-se esse programa para o hardware, observando seu comportamento. Primeiramente testou-se o funcionamento da rede com seu treinamento previamente efetuado no compilador C. Em seguida testou-se o funcionamento da rede com seu treinamento efetuado no próprio hardware.

Por último, utilizando os resultados obtidos na fase de implementação, foram feitas análises de desempenho de cada uma das redes, comparando seu funcionamento no compilador C e no kit DSK.

3.2 DESCRIÇÃO DA IMPLEMENTAÇÃO

3.2.1 PROGRAMAS EM LINGUAGEM C

Primeiramente foram elaborados códigos simulando redes neurais simples, de fácil entendimento, de forma a testar a validade do código ao observar seu comportamento.

Nesta etapa, escreveu-se o código simulando uma rede neural com apenas um neurônio perceptron. Este foi treinado para aprender o comportamento de uma porta lógica "AND" (E) de duas entradas.

Seu código era composto pelas seguintes funções:

- função inicializa() – inicializa os pesos das conexões entre as entradas e a unidade de processamento e o “bias” do neurônio;
- função Recebe_Entradas() – faz a leitura das duas entradas da rede, neste caso podem ser 0 ou 1;
- função forward() – responsável por fazer a soma ponderada das entradas e retornar o valor da função sigmóide desta soma. É utilizada dentro da função treinamento();
- função backprop() – calcula a alteração dos valores dos pesos e “bias” da rede;
- função treinamento() – apresenta os dados de treinamento da rede. Conforme o número de iterações determinado pelo usuário do programa, faz o treinamento da rede utilizando as entradas de teste. Neste caso, em que o mais importante era a visualização do comportamento do algoritmo de treinamento, foram utilizadas todas as possibilidades de entradas com as respectivas saídas, ou seja, a rede aprendeu todas as possíveis combinações do comportamento desejado. Essa função registra também o valor de cada um dos erros estabelecidos como o valor de cada saída desejada subtraída da saída obtida, soma estes erros dentro da função e imprime seu valor em um arquivo .dat, posteriormente utilizado nas análises de desempenho da rede. Este procedimento tem a função de acompanhar o aprendizado da rede.
- função main() – estabelecia um laço possibilitando ao usuário fazer o treinamento da rede com o número de iterações à sua escolha ou fazer um teste de funcionamento da rede.

Em seguida, foram escritos códigos simulando redes um pouco mais complexas, mas ainda com o fato indesejado de serem treinadas com todas as possibilidades. O objetivo destas redes foi validar o funcionamento da variação do algoritmo “backpropagation” escrito para camadas escondidas da rede.

Foi escrito um código para simular o funcionamento de uma porta lógica “OU-EXCLUSIVO” (XOR) de duas entradas. Esta rede era constituída por um neurônio MLP na camada de saída e dois neurônios MLP na camada escondida. Sua topologia está descrita na figura abaixo:

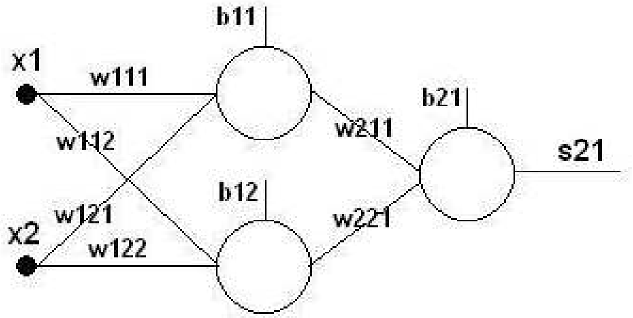

FIGURA 3.1 – DESCRIÇÃO DA TOPOLOGIA DA REDE NEURAL PARA REPRESENTAR A XOR COM DOIS NEURÔNIOS NA CAMADA INTERMEDIÁRIA.

Esse programa possuía uma estrutura bastante parecida com a estrutura do programa anterior. Mas nesta etapa o erro impresso na tela não convergia. Primeiramente como solução foi testada a utilização de um neurônio a mais na camada intermediária, o que não surtiu efeito em uma melhor convergência do erro. Após consideráveis esforços, detectou-se a causa do erro: como os pesos e os bias dos neurônios eram inicializados com valor igual a 0, provavelmente a convergência da rede caía em um mínimo local da função estabelecida pelo gradiente do erro. Utilizou-se então a função geradora de números aleatórios presente na biblioteca math.h do compilador C: rand(). Esta função gera um número natural aleatório no intervalo entre 0 e 32767. Em seguida dividiu-se este número gerado por 32767, pois o objetivo era gerar números aleatórios entre zero e um.

Esta alteração funcionou bem no algoritmo da porta “xor” de três neurônios na camada intermediária, mas não surtiu efeito no algoritmo com dois neurônios na camada intermediária. O algoritmo “backpropagation” ainda não conseguia convergir para o mínimo global da função gradiente do erro. Após um momento de reflexão, foi percebida a necessidade de inicializar os pesos e “bias” entre os valores –1 e 1. O que foi feito utilizando uma técnica simples. Primeiramente fez-se a inicialização randômica anteriormente descrita. Em seguida subtraiu-se deste número 0.5 e multiplicou-se o total por 0.5. Assim, a convergência do treinamento se estabeleceu em aproximadamente três mil iterações com uma taxa de aprendizado de 0.3.

Com um nível maior de complexidade, foi criado um código para uma rede neural com quatro entradas, dois neurônios na camada intermediária e quatro neurônios na camada de saída, cujo treinamento tinha o seguinte objetivo: fazer com que os neurônios da camada intermediária aprendessem a representação binária de números. Sua topologia pode ser mostrada na figura abaixo:

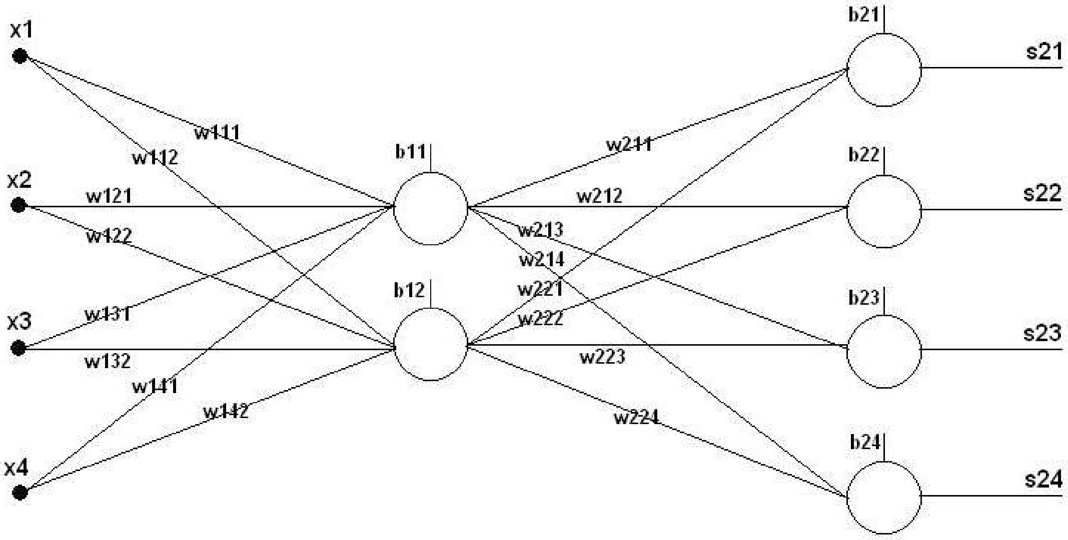

FIGURA 3.2 – DESCRIÇÃO DA TOPOLOGIA DA REDE NEURAL CODIFICADORA.

A estrutura deste código seguia a linha dos códigos criados anteriormente, diferenciando-se apenas pelo número de entradas, equações e saídas presentes. O funcionamento desejado da rede pode ser descrito pela tabela abaixo:

Tabela 3.1 – Possível comportamento desejado da rede codificadora.

Rede Codificadora									
Entradas				Saídas				Comportamento Desejado	
x1	x2	x3	x4	y1	y2	y3	y4	Saída Neurônio 1	Saída Neurônio 2
1	0	0	0	1	0	0	0	0	0
0	1	0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0	1	0
0	0	0	1	0	0	0	1	1	1

Lembrando que a ordem das representações binárias pode variar, pois esta é determinada exclusivamente pela rede.

O treinamento da rede era feito apresentando as quatro possibilidades de entradas com suas respectivas saídas desejadas, verificando a diferença entre cada saída obtida e sua respectiva saída desejada. Cada uma dessas quatro diferenças era armazenada em uma variável de erro, utilizada no algoritmo “backpropagation” para atualizar os pesos da rede.

Este algoritmo funcionou da maneira esperada, mostrando que as funções “forward”, “backpropagation” e treinamento no algoritmo em linguagem C estavam apresentando a funcionalidade prevista pela teoria.

3.2.2 IMPLEMENTAÇÃO NO HARDWARE TMS320VC5510®

O objetivo desta etapa é implementar os programas desenvolvidos anteriormente no hardware TMS320VC5510 da Texas Instruments. Para conseguir transferir os códigos desenvolvidos para a memória do hardware, foi utilizado o programa “Code Composer Studio®” versão 3.1.

Este programa funciona de maneira bastante amigável, possibilitando a escrita do código do programa em linguagem C, que, ao ser compilado, é transformado na linguagem Assembly utilizada pelo hardware.

Um ponto importante a ser considerado neste momento é o fato da unidade lógica aritmética presente no hardware trabalhar somente com aritmética de ponto fixo, isto é, só consegue realizar cálculos com números inteiros. Em todos os programas escritos foi utilizada a aritmética de ponto flutuante. Isto era necessário, pois, por definição, a função sigmóide retorna números no intervalo entre zero e um. Tornou-se necessário desenvolver uma metodologia para executar uma simulação da aritmética de ponto flutuante utilizando hardware de ponto fixo.

Pensou-se em dois procedimentos diferentes, o primeiro seria utilizar os códigos da mesma maneira que foram escritos e observar se o “Code Composer” conseguiria compilá-los para um arquivo .out e em seguida verificar se este arquivo era carregado e executado com sucesso pelo hardware.

O segundo procedimento, um pouco mais trabalhoso é bastante utilizado pela indústria de programação para hardwares microcontroladores e processadores digitais de sinais, pois, várias aplicações que necessitam de operações em ponto flutuante exigem hardware com baixo consumo de potência e menor custo, o que impossibilita a utilização de hardwares com unidade lógica aritmética de ponto flutuante.

Hardwares de ponto fixo lidam com os tipos de dados numéricos de forma diferente dos hardwares de ponto flutuante. A notação de ponto flutuante segue o padrão IEEE 754.

FIGURA 3.3 – REPRESENTAÇÃO EM PONTO FLUTUANTE.

No caso dos DSP sem suporte a ponto flutuante ele simplesmente separa parte dos bits de uma palavra como parte inteira e o resto para parte fracionária.

FIGURA 3.4 – REPRESENTAÇÃO EM PONTO FIXO.

O que se faz é na verdade utilizar um número inteiro como se fosse um número fracionário, por exemplo, representa-se o número 0,1234 por um número inteiro: 1234. No exemplo acima a manipulação é feita em base decimal. No caso dos computadores, essa manipulação é feita em base binária, já que a operação de divisão por estes valores é só uma questão de deslocamento de bits. Pensando em um hardware que armazena suas variáveis utilizando um espaço de 16 bits, poder-se-ia utilizar um intervalo de números entre –32768 e 32767. Assim, ao se pensar que o valor dos pesos, “bias” e entradas da rede não ultrapassam um intervalo determinado, por exemplo entre –32,768 e 32,767, pode-se representá-los com uma precisão de três casas decimais. Ao fazer esta conversão, é necessário arredondar a primeira casa decimal a ser desconsiderada, utilizar truncamento seria extremamente prejudicial à precisão dos cálculos. É necessário também utilizar um procedimento específico para evitar o fenômeno chamado “overflow”, isto é, quando o “carry” binário ultrapassa o número de casas utilizadas, sendo então jogado fora.

Ao verificar que o primeiro procedimento funcionou corretamente e por ser de uma maior facilidade de implementação, este foi utilizado. Alguns erros menores tiveram de ser corrigidos nos códigos utilizados, como o fato de o compilador do Code

Composer não conseguir inserir no código a função getch() da biblioteca conio.h. Para resolver essa situação, primeiramente foi tentado passar as bibliotecas conio.h e _mingw.h presentes no diretório include do compilador DEV para a pasta CCStudio_v3.1\C5500\cgtools\include. Sem obter resultados satisfatórios, simplesmente substituiu-se a função getch() pela função scanf() para receber os caracteres de comando do laço da função main() dos códigos.

Em seguida, fez-se uma análise de como o programa Code Composer fazia para que o código enviado para a memória do hardware simulasse a aritmética em ponto flutuante. Analisando sua lógica de funcionamento, foi percebido que a biblioteca padrão do programa própria para ser utilizada com o hardware C5510 possuía funções necessárias para efetuar a simulação de aritmética de ponto flutuante no hardware ponto fixo, possibilitando os cálculos na unidade lógica aritmética e o retorno dos valores calculados em ponto flutuante.

Assim, todos os códigos escritos até este momento funcionaram perfeitamente na memória do hardware, inclusive o treinamento da rede que, possivelmente, seria mais problemático em termos de necessidade de maior tempo de processamento e perda de precisão.

3.2.3 CLASSIFICAÇÃO DOS PADRÕES DA ÍRIS

Como já foi dito, uma das muitas funções das redes neurais artificiais é a de classificar padrões. Para testar a viabilidade de se implementar uma rede neural capaz de realizar uma tarefa que seria impraticável utilizando outro método escolheu-se o clássico problema de classificação das íris. A base de dados utilizada neste caso foi introduzida por Ronald A. Fisher em 1936 no livro “The use of multiple measurements in taxonomic problems. Annals of Eugenics 7”, e é uma das bases de dados mais conhecida na literatura de reconhecimento de padrões. Trata-se de 150 amostras contendo medidas do comprimento e largura da sépala e comprimento e largura da pétala de três espécies diferentes de flores, sendo 50 amostras para cada espécie. Uma das espécies tem amostras que são linearmente separáveis das outras duas.

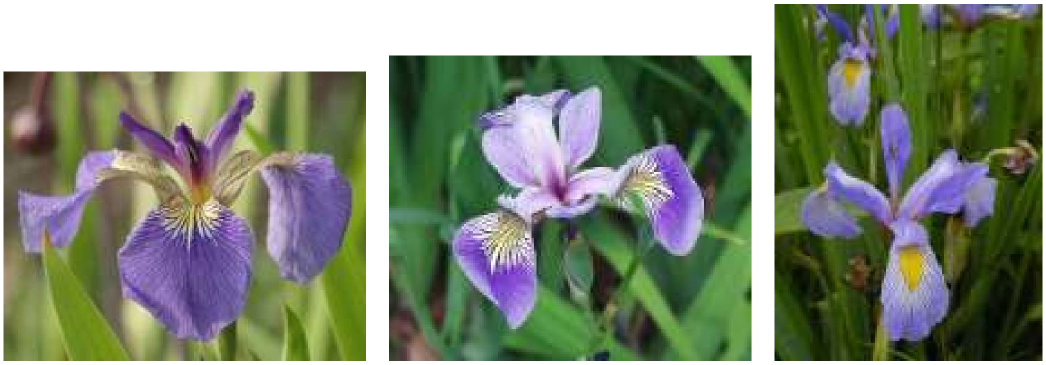

Figura 3.5 – Imagens das três espécies de planta do Gênero Íris: setosa, versicolor e virginica.

No projeto da rede foi levado em consideração que a saída esperada seria correspondente a uma e somente uma espécie. Usamos, portanto, três neurônios na camada de saída, todos com a sigmóide como função de ativação, de acordo com a figura a seguir:

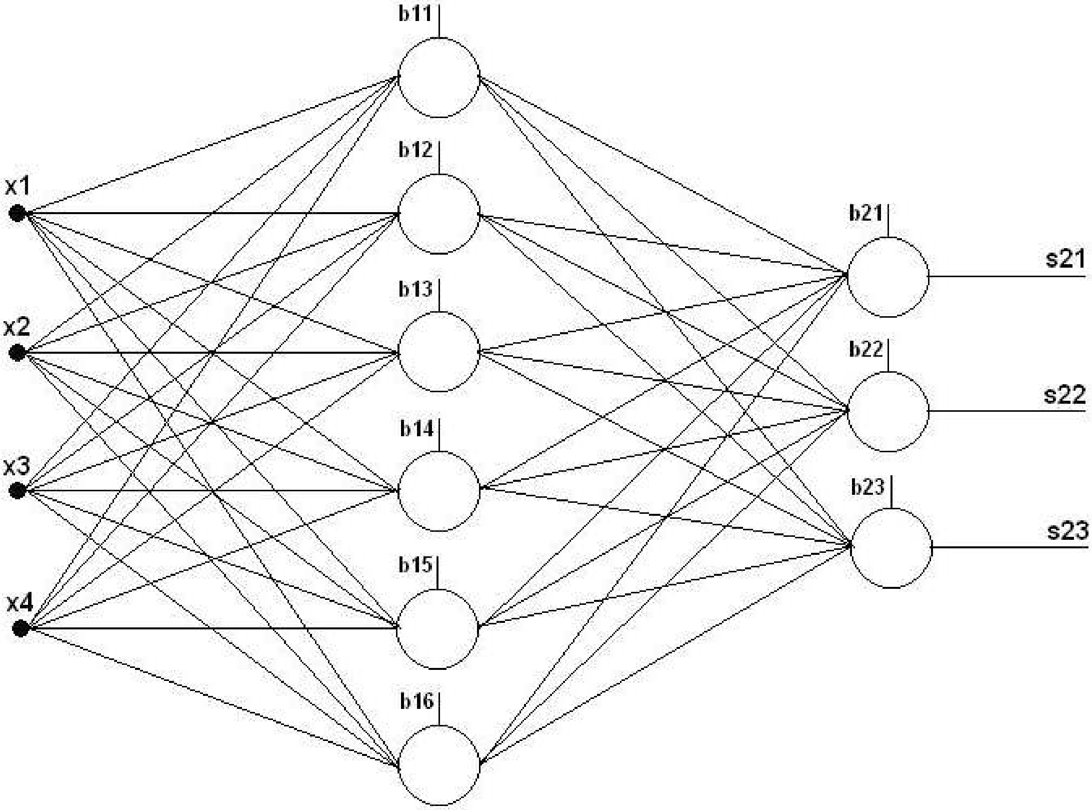

Figura 3.6 – Descrição da Topologia de rede neural representando a Classificadora de Íris.

A saída esperada em cada caso está indicada na tabela abaixo:

Tabela 3.2 – Parâmetros de saída da rede.

Espécie	Saída do neurônio 1	Saída do neurônio 2	Saída do neurônio 3
Íris setosa	1	0	0
Íris versicolor	0	1	0
Íris virginica	0	0	1

Separamos as 150 amostras em dois conjuntos. Um foi utilizado para o treinamento da rede e o outro para o teste de validação. O conjunto de treinamento foi composto de 30 amostras de cada espécie, totalizando 90 amostras. As 60 amostras restantes compuseram o conjunto de teste, sendo 20 de cada espécie.

3.2.4 TREINAMENTO (TREINAMENTO5510.C)

O primeiro passo foi a definição de algumas constantes:

#define NUMEROPADROES 90 */*define o número de amostras que serão utilizadas*/*

#define NUMEROENTRADAS 4 */*define o número de valores por amostra*/*

#define NUMEROESCONDIDA 6 */*define o número de neurônios na camada escondida*/*

#define NUMEROSAIDAS 3 */*define o número de neurônios na camada de saída*/*

#define NUMEROITERACOES 600 */*define o número de iterações para o treinamento*/*

Tal solução foi adotada por dois motivos: facilitar a leitura do programa e torná-lo mais generalista, no sentido de que se pode adaptar o código para redes neurais de tamanhos diferentes. No caso estudado, já era conhecido de antemão que 6 neurônios na camada escondida seriam suficientes para resolver o problema.

É possível alterar o número de neurônios da camada escondida facilmente, apenas alterando o valor da constante NUMEROESCONDIDA. Ao longo do experimento, utilizamos sempre 6 neurônios para resolver o problema da íris.

Os valores de entrada da rede foram carregados em um vetor entrada, cuja dimensão era de 90 linhas e 4 colunas. Cada linha era para um padrão diferente, enquanto cada coluna era para um dos 4 diferentes valores de uma certa amostra. A função do programa que realiza esta tarefa é a recebe_entradas(void).

Os valores de saída esperados foram passados ao vetor target, de dimensão 90 linhas por 3 colunas. Cada linha é referente a um padrão de entrada que será testado e cada coluna era para a saída esperada em cada neurônio. A função que realiza esta tarefa é a recebe_target(void). Para efeitos de cálculo do erro, há um trecho do programa que mapeia todas as entradas de treinamento para a saída. Por exemplo, a tabela abaixo mostra a relação entre entrada e saída esperada (target) para três exemplos:

Tabela 3.3 – Exemplos de padrões de entradas e saídas esperadas

CS	LS	CP	LP	Vetor-alvo		
5.1	3.5	1.4	0.2	1	0	0
7.0	3.2	4.7	1.4	0	1	0
6.3	3.3	6.0	2.5	0	0	1

CS ➔ Comprimento da sépala
LS ➔ Largura da sépala
CP ➔ Comprimento da pétala
LP ➔ Largura da pétala

É necessária a normalização dos valores de entrada da rede para que a média de cada amostra seja próxima de zero para garantir a convergência do erro com um número baixo de iterações.

$$p_i' = \frac{p_i}{(p_1^2 + p_2^2 + p_3^2 + p_4^2)^{1/2}} \tag{3.1}$$

onde,

p_i = valor de entrada a ser normalizado.

A função *main(void)* acumulou o que seriam inicialmente várias pequenas funções, por que assim seria mais fácil escrever o programa. Logo depois de chamar as funções *recebe_entradas(void)* e *recebe_target(void)* ela realiza a inicialização dos pesos e dos “bias” de forma aleatória, com valores entre -1 a 1. Em seguida se inicia o laço principal do programa, onde é realizada a propagação para frente para se obter uma saída de acordo com as entradas disponíveis e a propagação para trás (“backpropagation”) para que se realizem as correções nos valores dos pesos necessárias.

Antes disso é feito um pequeno trecho de código para que se altere a ordem em que se apresentam as entradas para a rede ao longo das iterações.

Em seguida é feita a soma ponderada de cada neurônio da camada intermediária e depois utiliza-se o resultado desta operação como argumento da função de ativação de cada neurônio (neste caso, a função sigmóide). A partir dos valores de saída de cada neurônio desta camada se repete o processo para os neurônios da camada de saída.

Dentro ainda o mesmo laço, realiza-se o cálculo do erro que será usado no algoritmo de "backpropagation", que corrigirá os pesos de modo com que o erro diminua ao longo das iterações.

No fim do laço principal do programa há um teste para saber se o número de iterações até o momento é um múltiplo inteiro de 100. Se for o caso o programa imprime o valor do erro. Há também o teste do critério de parada, caso o erro seja muito pequeno (menor que 0,0004) cessa-se as iterações, desde que não se atinja o valor definido na constante NUMEROITERACOES.

O restante do programa somente imprime em um arquivo os dados de saída relevantes para análise.

3.3 RESULTADOS E ANÁLISE

Os testes feitos com os algoritmos de simulação de redes sem a utilização do algoritmo "backpropagation" matricial obedeceram a critérios simples. Não se estabeleceu um número de iterações fixo no treinamento. Considerou-se como regra o número de iterações que fosse suficiente para que se pudesse obter os valores de resposta adequados de acordo com o seguinte critério de aproximação: valores de saídas menores ou iguais a 0,1 quando as saídas esperadas fossem 0 e valores de saídas maiores ou iguais a 0.9 para saídas esperadas iguais a 1. Isto se deve ao fato de a função sigmóide só atingir esses valores limites quando tende a mais ou menos infinito. Em seguida foi feita uma análise da convergência da soma dos erros de cada padrão apresentado ao longo das iterações do treinamento para o programa no compilador DEV C/C++ em um computador doméstico e no kit DSK. Cabe lembrar aqui que a análise da convergência da soma dos erros não tem significado sem a utilização de um critério de porcentagem de acerto que, nestes casos, foi estabelecido em 100%.

Nos testes feitos com a rede matricial classificadora de Íris, estabeleceu-se o número de iterações no treinamento igual a 600 épocas e comparou-se os resultados entre as três análises propostas, descrevendo a porcentagem de acerto e a convergência do erro de cada uma delas.

3.3.1 REDES NEURAIS SIMPLES

Porta “E”

Os testes do algoritmo responsável pela simulação de uma rede neural com aprendizado orientado ao comportamento de uma porta “E” apresentaram resultados satisfatórios. Em ambas as simulações (computador doméstico e hardware C5510) foi utilizada uma taxa de aprendizado de 0,5.

No compilador DEV C/C++ os resultados podem ser demonstrados pela tabela-verdade e pelo gráfico de convergência de erro descritos a seguir:

Tabela 3.4 – Padrões de entradas e saídas obtidas da rede “AND” no computador.

Rede "E" em computador (500 iterações)	
Pares de Entradas (x1 e x2)	Saída
0 e 0	0,000915
0 e 1	0,083842
1 e 0	0,084463
1 e 1	0,902163

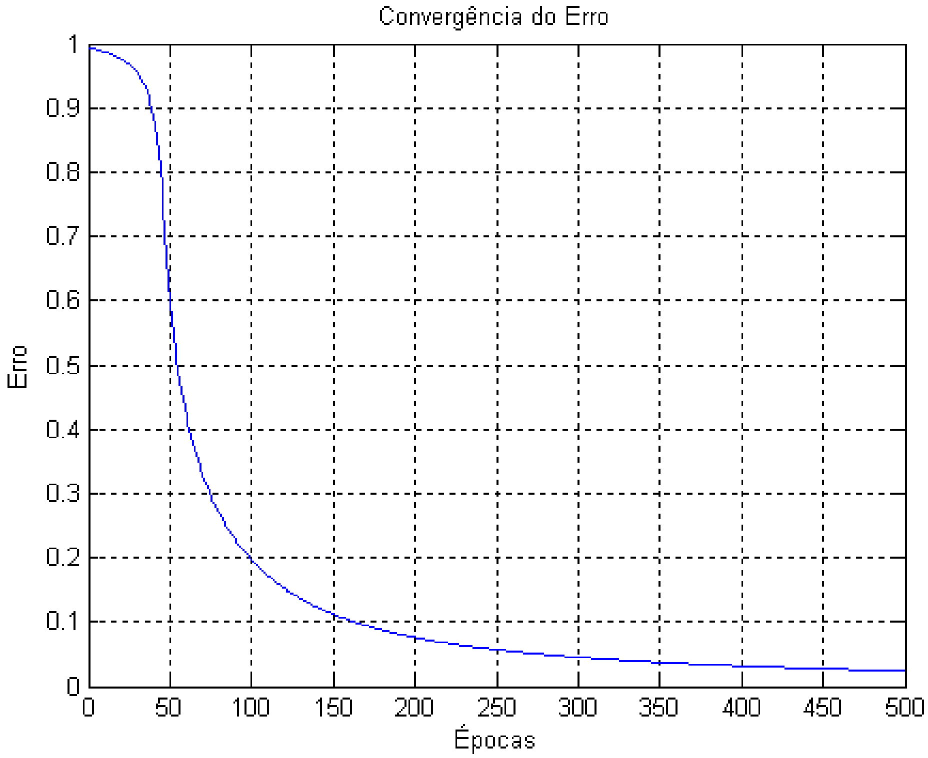

FIGURA 3.7 – GRÁFICO DA CONVERGÊNCIA DO ERRO PARA A REDE “AND” SIMULADA NO COMPUTADOR.

Ao analisar a performance do treinamento e teste da rede no C5510, foram obtidos resultados também bastante satisfatórios, apesar de ter-se verificado que o número de iterações necessárias para que a rede apresentasse a mesma precisão obtida na simulação desta mesma rede no computador foi maior.

Tabela 3.5 – Padrões de entradas e saídas obtidas da rede “AND” no hardware C5510.

Rede "And" em C5510 (850 iterações)	
Pares de Entradas (x1 e x2)	Saída
0 e 0	0,000927
0 e 1	0,084195
1 e 0	0,084823
1 e 1	0,901764

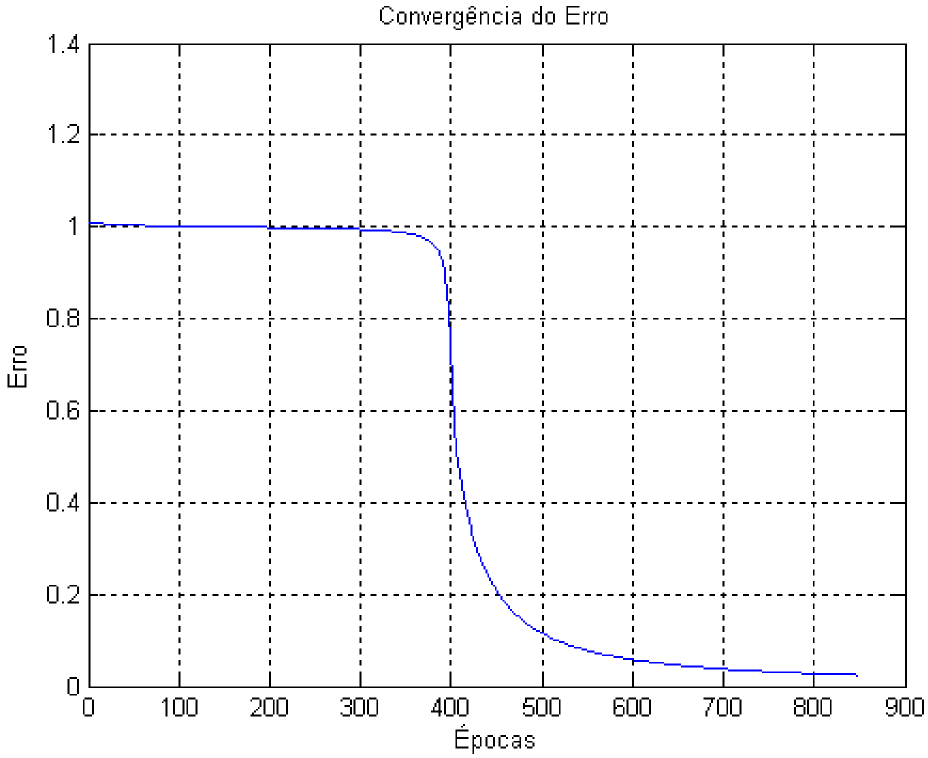

FIGURA 3.8 – GRÁFICO DA CONVERGÊNCIA DO ERRO PARA A REDE "AND" SIMULADA NO HARDWARE C5510.

Porta "OU-EXCLUSIVO" com dois neurônios na camada intermediária

Os mesmos procedimentos utilizados na etapa anterior foram utilizados para analisar o comportamento de uma rede neural com seu treinamento voltado para simular o comportamento de uma porta "OU-EXCLUSIVO" de duas entradas. Esta rede possui dois neurônios na camada intermediária, com taxa de aprendizado igual a 0,5.

De acordo com as tabelas e gráficos a seguir, percebe-se que, de acordo com o critério de aproximação e a convergência do erro, a rede obteve o desempenho desejado utilizando um menor número de iterações em sua implementação no hardware do que em sua implementação no computador.

Tabela 3.6 – Padrões de entradas e saídas obtidas da rede "OU-EXCLUSIVO" no computador.

Rede "OU-EXCLUSIVO" em computador (1250 iterações)	
Pares de Entradas (x1 e x2)	Saída
0 e 0	0,05845
0 e 1	0,917214
1 e 0	0,914552
1 e 1	0,098667

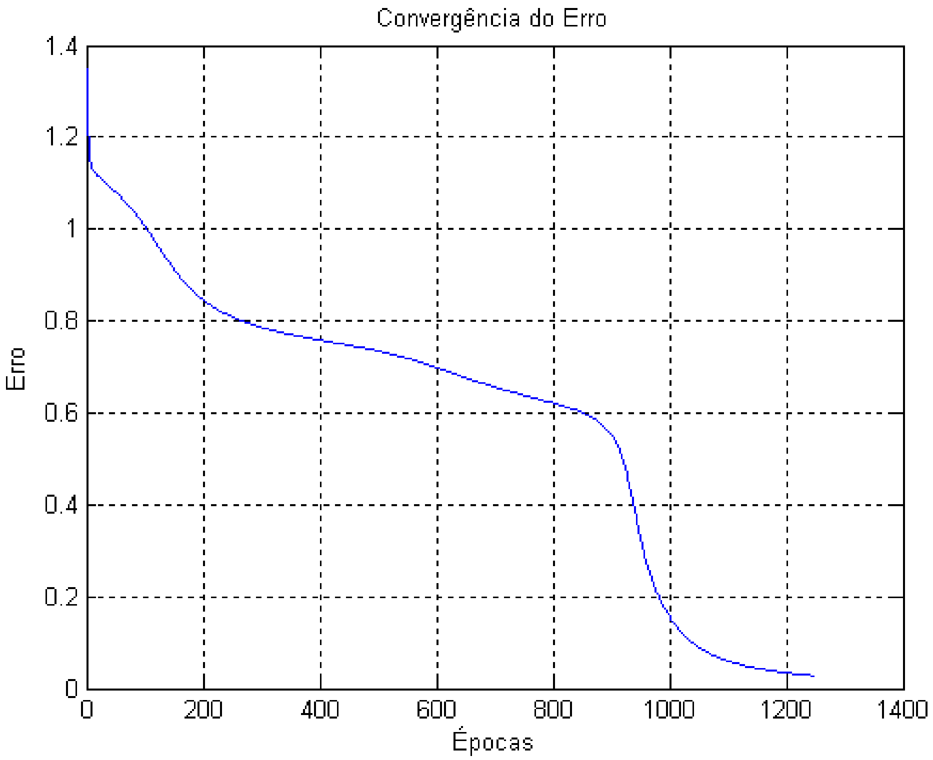

FIGURA 3.9 – GRÁFICO DA CONVERGÊNCIA DO ERRO PARA A REDE "XOR" COM DOIS NEURÔNIOS NA CAMADA INTERMEDIÁRIA SIMULADA NO COMPUTADOR.

Tabela 3.7 – Padrões de entradas e saídas obtidas da rede "XOR" no hardware C5510.

Rede "XOR" em C5510 (750 iterações)	
Pares de Entradas (x1 e x2)	Saída
0 e 0	0,064696
0 e 1	0,926187
1 e 0	0,925056
1 e 1	0,098077

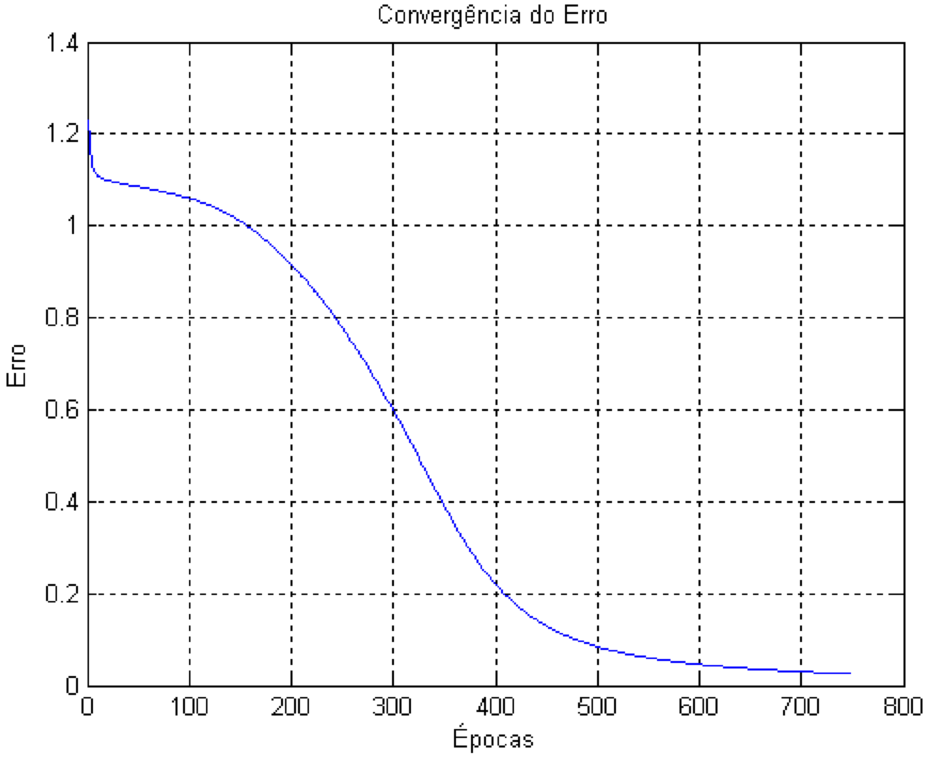

FIGURA 3.10 – GRÁFICO DA CONVERGÊNCIA DO ERRO PARA A REDE "XOR" COM DOIS NEURÔNIOS NA CAMADA INTERMEDIÁRIA SIMULADA NO HARDWARE C5510.

Porta "OU-EXCLUSIVO" com três neurônios na camada intermediária

Os testes efetuados com esta rede revelaram-se surpreendentes. Comparando-se o desempenho deste algoritmo com o algoritmo da porta "XOR" de dois neurônios testado anteriormente, notou-se que houve a necessidade de mais iterações com a utilização de um neurônio a mais na camada intermediária e a mesma taxa de aprendizado empregada. Isto nos leva novamente às dificuldades enfrentadas no projeto de redes neurais, pois exige a utilização de um procedimento heurístico na escolha de seus componentes (número de camadas, neurônios e conexões).

Comparando-se os testes no computador e no hardware ponto fixo, pode-se notar também que o teste efetuado no hardware atingiu a precisão pré-determinada com um número de iterações menor do que o teste efetuado no computador.

Tabela 3.8 – Padrões de entradas e saídas obtidas da rede "XOR" no computador.

Rede "XOR" em computador (1550 iterações)	
Pares de Entradas (x1 e x2)	Saída
0 e 0	0,044982
0 e 1	0,936134
1 e 0	0,90338
1 e 1	0,098622

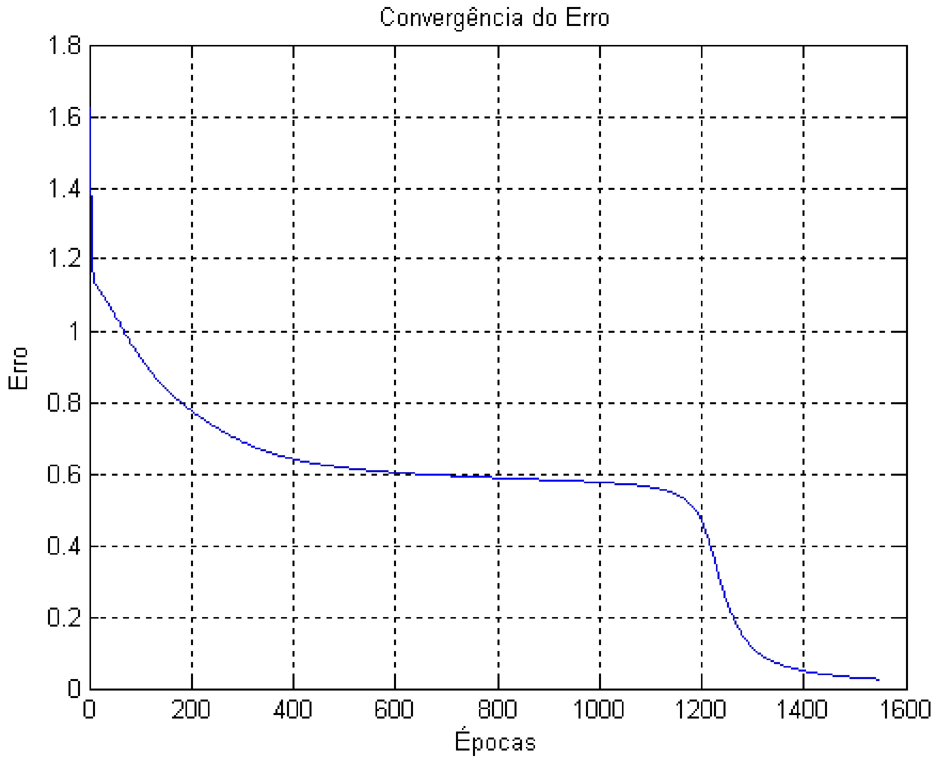

FIGURA 3.11 – GRÁFICO DA CONVERGÊNCIA DO ERRO PARA A REDE "XOR" COM TRÊS NEURÔNIOS NA CAMADA INTERMEDIÁRIA SIMULADA NO COMPUTADOR.

Tabela 3.9 – Padrões de entradas e saídas obtidas da rede "XOR" no hardware C5510.

Rede "XOR" em C5510 (1250 iterações)	
Pares de Entradas (x1 e x2)	Saída
0 e 0	0,053635
0 e 1	0,906639
1 e 0	0,938791
1 e 1	0,09018

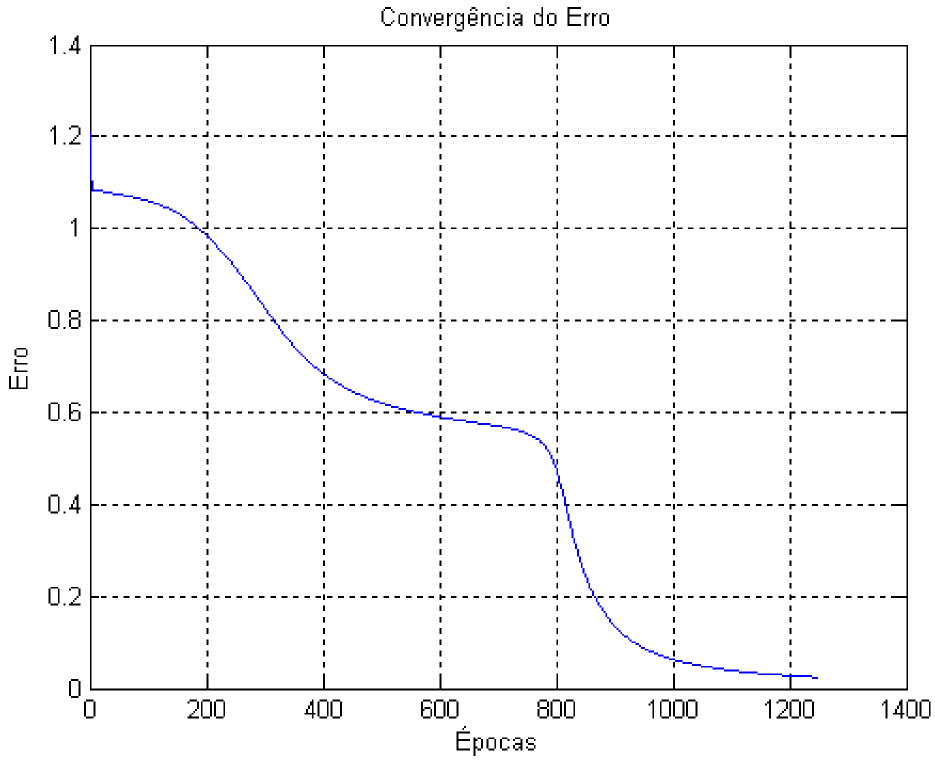

FIGURA 3.12 – GRÁFICO DA CONVERGÊNCIA DO ERRO PARA A REDE "XOR" COM TRÊS NEURÔNIOS NA CAMADA INTERMEDIÁRIA SIMULADA NO HARDWARE C5510.

Rede Codificadora

Valendo-se do mesmo procedimento utilizado até aqui foi testado o algoritmo de simulação da rede codificadora. A taxa de aprendizado foi mantida em 0,5. Nesta etapa, observou-se que o algoritmo no computador convergiu com um número menor de iterações que no hardware. Foi notado também o comportamento esperado ao se utilizar este tipo de rede. Em ambos os processos é possível verificar a padronização das saídas dos neurônios da camada intermediária, estabelecendo uma representação binária. Com o número utilizado de iterações, não foi possível estabelecer o mesmo critério de parada utilizado na camada de saída para as saídas dos neurônios na camada intermediária. Mas ainda assim é possível validar o processo de codificação.

Tabela 3.10 – Padrões de entradas e saídas obtidas da rede codificadora no computador.

Rede Codificadora implementada em computador (700 iterações)									
Entradas				Saídas				Comportamento Obtido	
x1	x2	x3	x4	Y1	y2	y3	y4	neurônio 1	neurônio 2
1	0	0	0	0,900469	0,091953	0,049827	0,000708	0,021401	0,676036
0	1	0	0	0,051516	0,909411	0,000128	0,070133	0,811002	0,975082
0	0	1	0	0,091087	0,001705	0,900327	0,065718	0,225839	0,013406
0	0	0	1	0,000462	0,057984	0,074387	0,911227	0,980723	0,185678

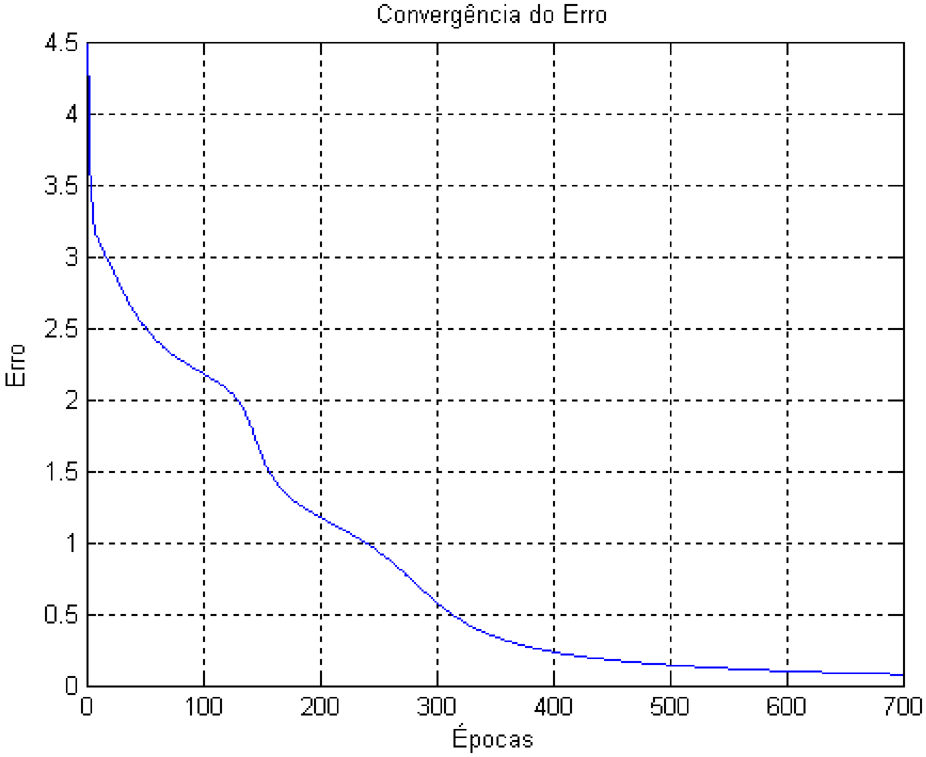

FIGURA 3.13 – GRÁFICO DA CONVERGÊNCIA DO ERRO PARA A REDE CODIFICADORA SIMULADA NO COMPUTADOR.

Tabela 3.11 – Padrões de entradas e saídas obtidas da rede codificadora no hardware C5510.

Rede Codificadora implementada em C5510 (900 iterações)									
Entradas				Saídas				Comportamento Obtido	
x1	x2	x3	x4	y1	y2	y3	y4	neurônio 1	neurônio 2
1	0	0	0	0,902569	0,074333	0,001574	0,062757	0,294429	0,010488
0	1	0	0	0,055783	0,914792	0,083898	0,000863	0,014051	0,56356
0	0	1	0	0,000122	0,044725	0,912559	0,062644	0,662791	0,979533
0	0	0	1	0,067452	0,000146	0,058663	0,919587	0,980858	0,290624

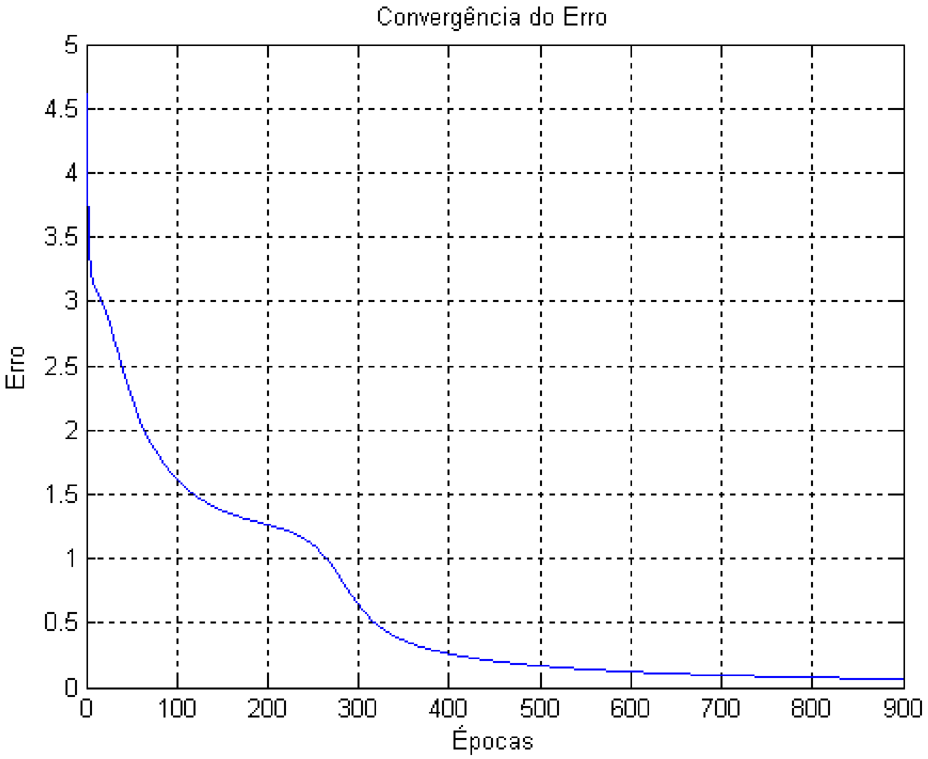

FIGURA 3.14 – GRÁFICO DA CONVERGÊNCIA DO ERRO PARA A REDE CODIFICADORA SIMULADA NO HARDWARE C5510.

3.3.2 REDE CLASSIFICADORA DE PADRÕES ÍRIS

Foram feitos três testes considerados relevantes para a análise de rede neural em hardware de ponto fixo. O primeiro teste foi realizado no computador com um número de apenas 600 iterações. O objetivo é ter um padrão de comparação para avaliar a rede implementada no kit DSP. Como já foi dito, foi realizado um treinamento com parte dos dados e depois a validação com a outra parte. Foram realizados o treinamento na placa e depois a validação utilizando o conjunto de teste. Para efeito de decisão da resposta da rede, foi considerado igual a 1 a maior das três saídas e 0 para as outras duas. Este procedimento foi feito para dois valores de taxa de aprendizado para verificar o seu efeito na convergência do erro: 0,5 e 0,1.

Procedimento para Taxa de Aprendizado de 0,5

Selecionado os conjuntos de treinamento e de validação, foi feita a simulação da rede tanto no computador quanto na placa. Em ambos os casos foi utilizado o número de

600 iterações, o que se mostrou suficiente. Com a rede treinada obteve-se então a taxa de acertos para o conjunto de treinamento.

Tabela 3.12 – Contabilização de acertos na classificação das espécies de Íris obtidos na simulação da rede no computador.

Conjunto de Treinamento - Computador		
Tipo de flor	acertos	total de acertos
Íris setosa	100%	97,78%
Íris versicolor	96,67%	
Íris virginica	96,67%	

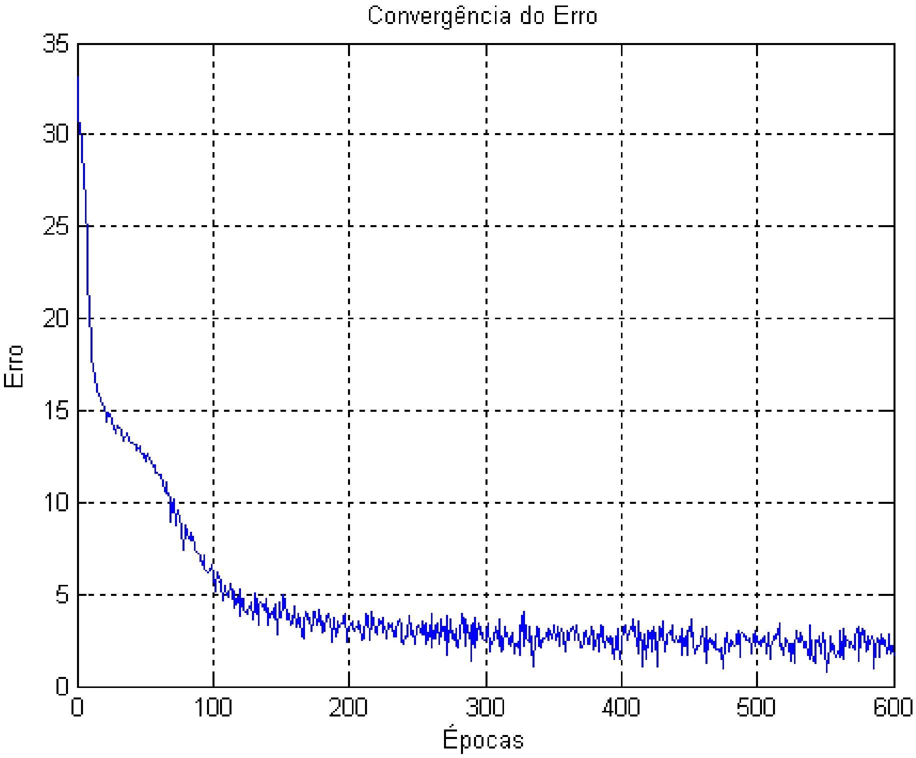

FIGURA 3.15 – GRÁFICO DA CONVERGÊNCIA DO ERRO PARA A REDE CLASSIFICADORA ÍRIS SIMULADA NO COMPUTADOR.

Tabela 3.13 – Contabilização de acertos na classificação das espécies de Íris obtidos na simulação da rede no C5510.

Conjunto de Treinamento - Hardware		
tipo de flor	acertos	total de acertos
Íris setosa	100%	94,44%
Íris versicolor	93,33%	
Íris virginica	90,00%	

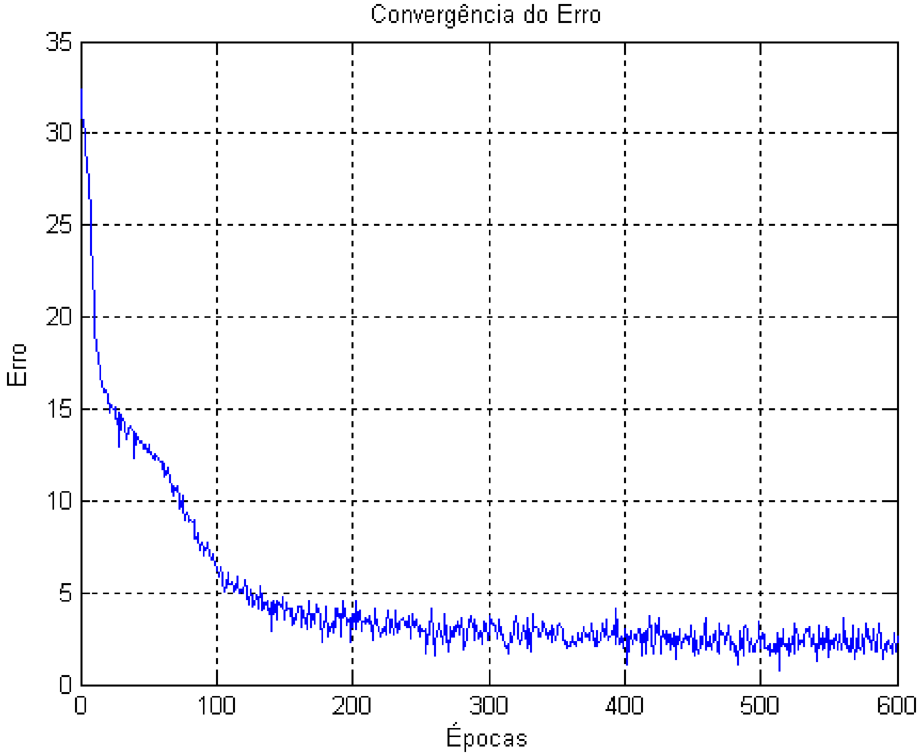

FIGURA 3.16 – GRÁFICO DA CONVERGÊNCIA DO ERRO PARA A REDE CLASSIFICADORA ÍRIS SIMULADA NO C5510.

Para testar com o conjunto de validação foram realizados três procedimentos: o teste da rede simulada no computador, o teste da rede simulada na placa com os pesos treinados no computador e o teste da rede simulada na placa com os pesos obtidos pelo treinamento nela própria.

Tabela 3.14 – Contabilização de acertos na classificação das espécies de Íris obtidos na simulação da rede no computador para o conjunto de teste.

Treinamento e Teste no Computador		
Tipo de flor	acertos	total de acertos
Íris setosa	100,00%	96,67%
Íris versicolor	95,00%	
Íris virginica	95,00%	

Tabela 3.15 – Contabilização de acertos na classificação das espécies de Íris obtidos na simulação da rede na placa com os pesos obtidos no treinamento no computador para o conjunto de teste.

Treinamento no Computador e Teste no Hardware		
Tipo de flor	acertos	total de acertos
Íris setosa	100,00%	95,00%
Íris versicolor	95,00%	
Íris virginica	90,00%	

Tabela 3.16 – Contabilização de acertos na classificação das espécies de Íris obtidos na simulação da rede na placa para o conjunto de teste.

Treinamento e Teste no Hardware		
Tipo de flor	acertos	total de acertos
Íris setosa	100,00%	98,33%
Íris versicolor	95,00%	
Íris virginica	100,00%	

De acordo com a porcentagem de acerto obtida em cada caso, percebeu-se que o caso com menor precisão foi o do teste da rede no hardware com o treinamento efetuado no computador. O caso com melhor precisão foi o do teste e treinamento na placa. Uma hipótese provável para o melhor funcionamento da rede com seu treinamento efetuado em ponto fixo se deve ao fato de que, ao ser treinada em ponto fixo, a rede se adapte de uma melhor maneira à menor precisão nos cálculos efetuados.

Procedimento para Taxa de Aprendizado de 0,1

Foi efetuado o mesmo procedimento anterior, apenas alterando o valor da variável taxa de aprendizado para 0,1 no programa de treinamento da rede.

Foram obtidos os seguintes gráficos de convergência de erro e tabelas de porcentagem de acerto:

Tabela 3.17 – Contabilização de acertos na classificação das espécies de Íris obtidos na simulação da rede no computador para o conjunto de treinamento.

Conjunto de Treinamento - Computador		
Tipo de flor	acertos	total de acertos
Íris setosa	100%	98,89%
Íris versicolor	96,67%	
Íris virginica	100%	

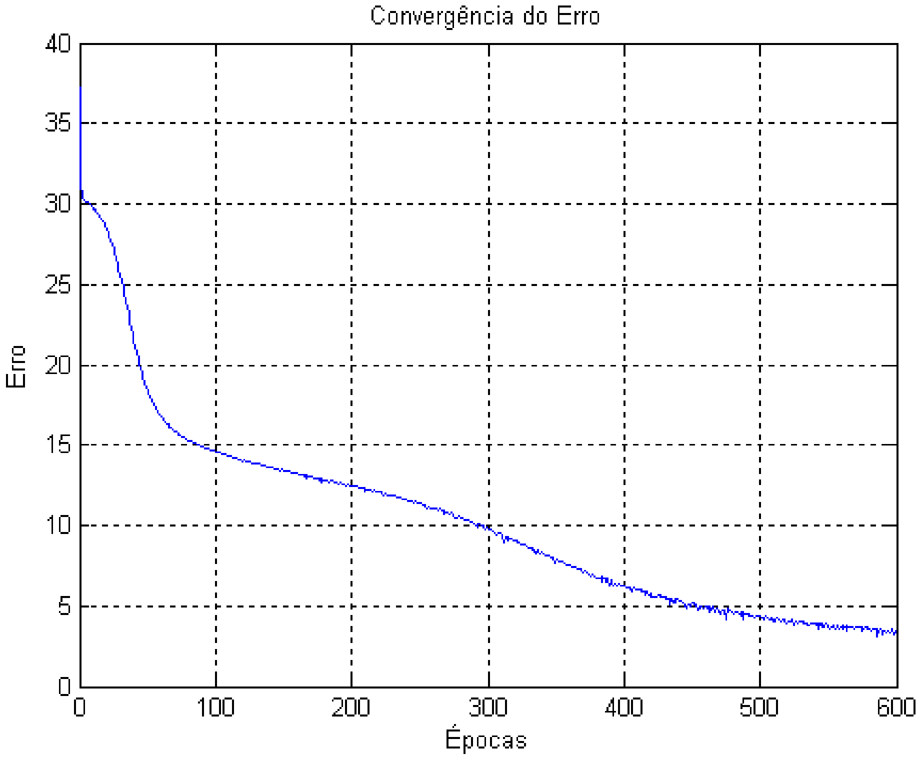

FIGURA 3.17 – GRÁFICO DA CONVERGÊNCIA DO ERRO PARA A REDE CLASSIFICADORA ÍRIS SIMULADA NO COMPUTADOR.

Tabela 3.18 – Contabilização de acertos na classificação das espécies de Íris obtidos na simulação da rede no C5510 para o conjunto de treinamento.

Conjunto de Treinamento - Hardware		
Tipo de flor	Acertos	total de acertos
Íris setosa	100%	97,78%
Íris versicolor	93,33%	
Íris virginica	100%	

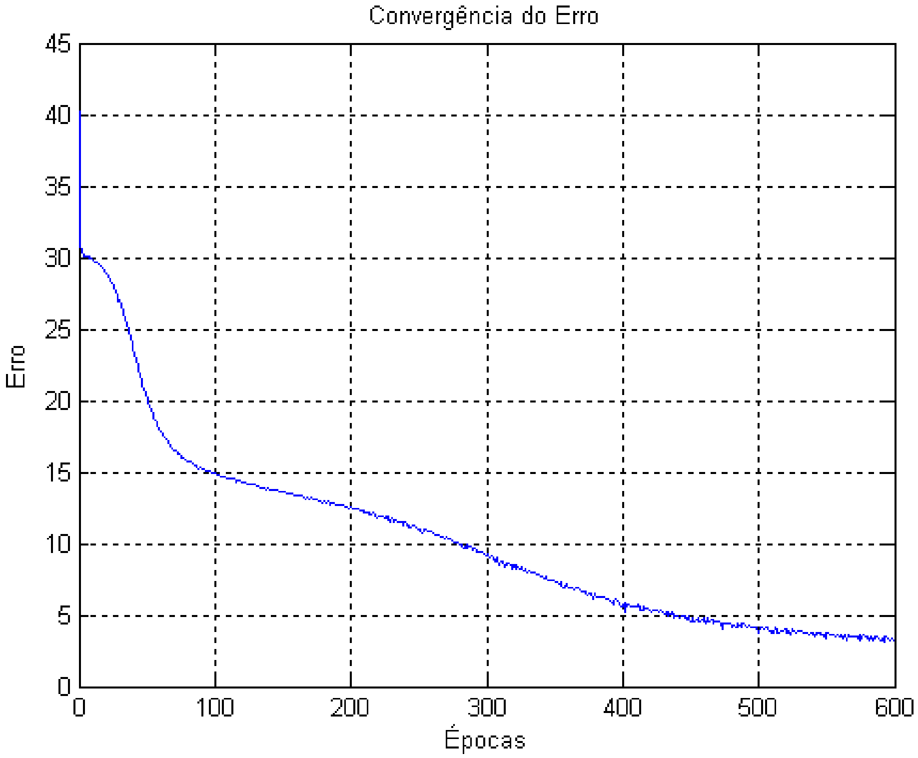

FIGURA 3.18 – GRÁFICO DA CONVERGÊNCIA DO ERRO PARA A REDE CLASSIFICADORA ÍRIS SIMULADA NO C5510.

É fácil observar que em ambas as simulações de treinamento, o erro convergiu de maneira menos oscilatória que no caso da taxa de aprendizado de 0,5, porém o valor mínimo do erro quadrático atingido depois de 600 iterações foi maior, isto é, para uma taxa de aprendizado menor, o erro converge de forma mais lenta. Se o número de iterações utilizado fosse um pouco maior, a rede com essa taxa de aprendizado menor poderia atingir um nível de erro quadrático igual ao caso anterior.

Em seguida, apresentou-se os resultados obtidos nos três procedimentos de testes mencionados acima:

Tabela 3.19 – Contabilização de acertos na classificação das espécies de Íris obtidos na simulação da rede no computador para o conjunto de teste.

Treinamento e Teste no Computador		
	acertos	total de acertos
Íris setosa	100,00%	98,33%
Íris versicolor	95,00%	
Íris virginica	100,00%	

Tabela 3.20 – Contabilização de acertos na classificação das espécies de Íris obtidos na simulação da rede na placa com os pesos obtidos no treinamento no computador para o conjunto de teste.

Treinamento no Computador e Teste no Hardware		
Tipos de flor	acertos	total de acertos
Íris setosa	100,00%	96,67%
Íris versicolor	95,00%	
Íris virginica	95,00%	

Tabela 3.21 – Contabilização de acertos na classificação das espécies de Íris obtidos na simulação da rede na placa para o conjunto de teste.

Treinamento e Teste no Hardware		
Tipo de flor	acertos	total de acertos
Íris setosa	100,00%	93,33%
Íris versicolor	95,00%	
Íris virginica	85,00%	

4 CONSIDERAÇÕES FINAIS

4.1 CONCLUSÃO

A comparação entre o funcionamento das redes neurais no hardware de ponto fixo e no computador e a verificação da viabilidade de implementação neste hardware foram os objetivos deste trabalho. Aspectos de otimização da rede como o uso do componente de momento e a influência do número de neurônios ou de camadas não foram levados em conta. Abordou-se de forma extensiva características importantes do treinamento feito através do algoritmo "backpropagation", como inicialização aleatória dos pesos e "bias" da rede, utilização de uma taxa de aprendizado conveniente, comportamento da convergência do erro quadrático.

Utilizando-se algoritmos estruturados em linguagem C, foram elaboradas diversas soluções para alguns problemas, os quais no começo eram bastante simplificados, com o intuito de validar a metodologia utilizada para simular o processamento paralelizado e distribuído de uma rede neural MLP. Em seguida, abordou-se um procedimento para a resolução de um problema mais complexo, a classificação de um banco de dados de entrada/saída clássico na teoria de Redes Neurais. Esta fase se caracterizou pela utilização de um programa em linguagem C mais flexível, pois estabelecia a simulação do funcionamento da rede neural através de rotinas matriciais e que possibilitava uma maior flexibilização da rede ao permitir a variação de padrões como número de entradas, neurônios na camada escondida e na camada de saída.

Fez-se uso também de um hardware de ponto fixo para implementar a rede, a um ponto que foi considerado satisfatório, inclusive com relação ao treinamento da rede implementado no hardware. O algoritmo de aprendizado adotado é bastante robusto, já que leva em consideração o erro proveniente da falta de precisão nos cálculos.

4.2 APLICAÇÕES E POSSIBILIDADES FUTURAS

A partir do trabalho realizado é possível criar experimentos para laboratório de disciplinas que estudem redes neurais. Também é viável dar continuidade ao trabalho no sentido de investigar a origem das diferenças numéricas entre a rede implementada no hardware de ponto fixo e no computador. Outro caminho seria otimizar as redes implementadas para resolver outros problemas específicos, investigando a influência de mais camadas na rede, da quantidade de neurônios em cada camada, utilizando-se recursos como o momento etc.

Em geral as aplicações de redes neurais exigem algum pré-processamento. O problema da íris, por exemplo, exigiu que se fizesse a normalização dos dados de entrada. No caso de se tentar reconhecer voz deve-se calcular a transformada de Fourier e lançar mão de um algoritmo como o Cepstrum para obter entradas para a rede. Uma possibilidade de continuação deste projeto é escrever também para o hardware de ponto fixo rotinas que realizem esse tipo de pré-processamento no hardware para que se façam aplicações realmente autônomas.

5 REFERÊNCIAS BIBLIOGRÁFICAS

[1] BRAGA, A.P., CARVALHO, A.C.P.L.F., LUDERMIR, T.B., **Redes Neurais Artificiais: Teoria e Aplicações.** 1ª Edição, Rio de Janeiro, RJ, Editora LTC, 2000.

[2] HAYKIN, Simon, **Redes Neurais: Princípios e prática.** 2ª Edição, São Paulo, SP, Bookman Companhia Editora, 2001.

[3] KOVÁCS, Z.L., **Redes Neurais Artificiais: Fundamentos e Aplicações.** 1.ª Edição, São Paulo, SP, Livraria da Física, 1996.

[4] DORF, Richard C., **The Electrical Engineering Handbook**. 4ª Edição, EUA, Editora CRC Press, 1997.

[5] CAUDILL, Maureen, BUTLER, Charles, **Understanding Neural Networks: Computer Explorations.** 1ª Edição, EUA, Massachusetts Institute of Technology, 1992.

[6] JAMSA, Kris, KLANDER, Lars, **Programando em C/C++ - A Bíblia.** 1ª Edição, São Paulo, SP, Editora MAKRON Books, 1999.

[7] DEITEL, H.M, DEITEL, P.J., **C How to Program.** 4ª Edição, EUA, Editora Addison Wesley, 2003.

[8] PATTERSON, D. A., HENNESSY, J. L., **Organização e Projeto de Computadores: A Interface Hardware/Software**. 2ª Edição, Rio de Janeiro, RJ, Editora LTC, 2000.

[9] LEE, H. KUO, M., **Real-Time Digital Signal Processing.** 1ª Edição, EUA, Editora John Wiley Professional, 2001.

[10] **TMS320VC5510 Technical Reference.** Spectrum Digital, Inc, 2004.

[11] BARRETO, Jorge M., **Introdução às Redes Neurais Artificiais.** Disponível em: <http://www.din.uem.br/ia/neurais/>, Acesso em: 12/12/2006, às 16:00.

[12] GORDON, Robert, **A Calculated Look at Fixed-Point Arithmetic.** Disponível em: <http://www.embedded.com/98/9804fe2.htm>, Acesso em 09/11/2006, às 10:00.

APÊNDICE

```
/************************************************************
  xor.c
  autores: Joel Fernando Jardim Martins
              Carlos Felipe Ávila Klein
  Implementação em rede neural MLP com dois neurônios na camada escondida
************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

float a1, a2, a3, b1, b2, b3, s1, s2, s3, erro3;
int x1, x2;
int i, n;
float w111, w112, w121, w122, w211, w221;
float step_size;

void inicializa()
{
    w111=rand();
    w112=rand();
    w121=rand();
    w122=rand();
    w211=rand();
    w221=rand();
    b1=rand();
    b2=rand();
    b3=rand();
    w111=((w111/32767)-0.5)/0.5;
    w112=((w112/32767)-0.5)/0.5;
    w121=((w121/32767)-0.5)/0.5;
    w122=((w122/32767)-0.5)/0.5;
    w211=((w211/32767)-0.5)/0.5;
    w221=((w221/32767)-0.5)/0.5;
    b1=((b1/32767)-0.5)/0.5;
    b2=((b2/32767)-0.5)/0.5;
    b3=((b3/32767)-0.5)/0.5;
```

```
    printf("\n %f", w111);
    printf("\n %f", w112);
    printf("\n %f", w121);
    printf("\n %f", w122);
    printf("\n %f", w211);
    printf("\n %f", w221);
    printf("\n %f", b1);
    printf("\n %f", b2);
    printf("\n %f", b3);

}

void forward()
{

    a1=w111*x1+w121*x2+b1;
    s1=1/(1+exp(-2*a1));
    a2=w112*x1+w122*x2+b2;
    s2=1/(1+exp(-2*a2));

    a3=s1*w211+s2*w221+b3;
    s3=1/(1+exp(-2*a3));

}

void atualiza_pesos()
{

    w211=w211+erro3*((s3)*(1-s3))*s1*step_size;
    w221=w221+erro3*((s3)*(1-s3))*s2*step_size;
    b3= b3+erro3*((s3)*(1-s3))*step_size;

    w111=w111+x1*((s1)*(1-s1))*erro3*((s3)*(1-s3))*w211*step_size;
    w112=w112+x1*((s2)*(1-s2))*erro3*((s3)*(1-s3))*w221*step_size;

    w121=w121+x2*((s1)*(1-s1))*erro3*((s3)*(1-s3))*w211*step_size;
    w122=w122+x2*((s2)*(1-s2))*erro3*((s3)*(1-s3))*w221*step_size;

    b1=b1+erro3*((s1)*(1-s1))*((s3)*(1-s3))*w211*step_size;
```

```
    b2=b2+erro3*((s2)*(1-s2))*((s3)*(1-s3))*w221*step_size;

}

void treinamento()
{
int i, n;
float somaerro;
FILE *cfPtr;

        printf("numero de iteracoes: ");
        scanf("%d", &n);


    cfPtr=fopen("somaerro.dat", "w");
        for (i=1; i<=n; i++)
        {
       somaerro=0;
             x1=0;
             x2=0;
             forward();
             erro3=0-s3;
             somaerro+=erro3*erro3;
             atualiza_pesos();

       x1=1;
             x2=1;
             forward();
             erro3=0-s3;
             somaerro+=erro3*erro3;
             atualiza_pesos();

             x1=0;
             x2=1;
             forward();
             erro3=1-s3;
             somaerro+=erro3*erro3;
             atualiza_pesos();

       x1=1;
```

```
            x2=0;
            forward();
            erro3=1-s3;
            somaerro+=erro3*erro3;
            atualiza_pesos();
        //printf("\nsomaerro= %f", somaerro);
           fprintf(cfPtr,"\nepoca %d\t %f", i, somaerro); ;

    }
    fclose(cfPtr);
}

void recebe_entradas()
{

    printf("x1: ");
    scanf("%d", &x1);
    printf("\nx2: ");
    scanf("%d", &x2);

}

int main()
{
char opcao, fluxo;

inicializa();
printf("\nstep size: ");
scanf("%f", &step_size);

while (fluxo!='d')
{

  printf("\n\nEscolha uma opcao:\na-Treinar rede\nb-Utilizar rede\n");
  scanf("\n%c", &opcao);
  if (opcao=='a') treinamento();
  if (opcao=='b')
  {

  recebe_entradas();
```

```
   forward();
   printf("\nsaida: %f", s3);

   }
   printf("\nc-continuar\nd-sair");
   scanf("\n%c", &fluxo);
}
return 0;
}


/**************************************************************************/


/************************************************************
  xor.c
  autores: Joel Fernando Jardim Martins
              Carlos Felipe Ávila Klein
  Implementação em rede neural MLP com dois neurônios na camada escondida
*************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define NUMEROPADROES 90
#define NUMEROENTRADAS 4
#define NUMEROESCONDIDA 6
#define NUMEROSAIDAS 3
#define NUMEROITERACOES 601
#define rando() ((float)rand()/RAND_MAX)



 int    i, j, k, p, np, op, ranpad[NUMEROPADROES+1], epoca;
 int      NumPadroes = NUMEROPADROES, NumEntrada = NUMEROENTRADAS, NumEscondida =
NUMEROESCONDIDA, NumSAIDA = NUMEROSAIDAS;
 float entrada[NUMEROPADROES+1][NUMEROENTRADAS+1];
 float target[NUMEROPADROES+1][NUMEROSAIDAS+1];
 float                                SomaEscondida[NUMEROPADROES+1][NUMEROESCONDIDA+1],
W12[NUMEROENTRADAS+1][NUMEROESCONDIDA+1],
Escondida[NUMEROPADROES+1][NUMEROESCONDIDA+1];
 float                                          SomaSaida[NUMEROPADROES+1][NUMEROSAIDAS+1],
W23[NUMEROESCONDIDA+1][NUMEROSAIDAS+1],
SAIDA[NUMEROPADROES+1][NUMEROSAIDAS+1];
```

```
float DeltaS[NUMEROSAIDAS+1], somaDWS[NUMEROESCONDIDA+1], DeltaE[NUMEROESCONDIDA+1];
float DeltaW12[NUMEROENTRADAS+1][NUMEROESCONDIDA+1] , DeltaW23[NUMEROESCONDIDA+1][NUMEROSAIDAS+1];
float Erro, eta = 0.1, alpha = 0, wmax = 1;

void recebe_entradas(void)
{

entrada[1][1] = 0.818031;
entrada[1][2] = 0.517530;
entrada[1][3] = 0.250418;
entrada[1][4] = 0.016695;
entrada[2][1] = 0.775300;
entrada[2][2] = 0.283046;
entrada[2][3] = 0.541480;
entrada[2][4] = 0.159983;
entrada[3][1] = 0.714865;
entrada[3][2] = 0.259951;
entrada[3][3] = 0.622026;
entrada[3][4] = 0.185679;
entrada[4][1] = 0.817338;
entrada[4][2] = 0.514620;
entrada[4][3] = 0.257310;
entrada[4][4] = 0.030272;
entrada[5][1] = 0.753849;
entrada[5][2] = 0.315246;
entrada[5][3] = 0.548254;
entrada[5][4] = 0.178183;
entrada[6][1] = 0.690259;
entrada[6][2] = 0.350979;
entrada[6][3] = 0.596665;
entrada[6][4] = 0.210588;
entrada[7][1] = 0.778674;
entrada[7][2] = 0.594624;
entrada[7][3] = 0.198208;
entrada[7][4] = 0.028315;
entrada[8][1] = 0.739235;
entrada[8][2] = 0.375882;
entrada[8][3] = 0.526235;
entrada[8][4] = 0.187941;
```

```
entrada[9][1] = 0.715626;
entrada[9][2] = 0.352308;
entrada[9][3] = 0.561492;
entrada[9][4] = 0.220193;
entrada[10][1] = 0.828133;
entrada[10][2] = 0.507020;
entrada[10][3] = 0.236609;
entrada[10][4] = 0.033801;
entrada[11][1] = 0.745498;
entrada[11][2] = 0.372749;
entrada[11][3] = 0.524178;
entrada[11][4] = 0.174726;
entrada[12][1] = 0.706006;
entrada[12][2] = 0.238392;
entrada[12][3] = 0.632655;
entrada[12][4] = 0.210885;
entrada[13][1] = 0.788895;
entrada[13][2] = 0.552226;
entrada[13][3] = 0.252446;
entrada[13][4] = 0.094667;
entrada[14][1] = 0.763019;
entrada[14][2] = 0.335266;
entrada[14][3] = 0.531801;
entrada[14][4] = 0.150292;
entrada[15][1] = 0.705589;
entrada[15][2] = 0.327230;
entrada[15][3] = 0.582878;
entrada[15][4] = 0.235196;
entrada[16][1] = 0.860939;
entrada[16][2] = 0.440035;
entrada[16][3] = 0.248716;
entrada[16][4] = 0.057396;
entrada[17][1] = 0.761852;
entrada[17][2] = 0.340112;
entrada[17][3] = 0.530575;
entrada[17][4] = 0.149649;
entrada[18][1] = 0.695899;
entrada[18][2] = 0.347949;
entrada[18][3] = 0.576291;
entrada[18][4] = 0.250089;
```

```
entrada[19][1] = 0.826998;
entrada[19][2] = 0.526271;
entrada[19][3] = 0.195472;
entrada[19][4] = 0.030073;
entrada[20][1] = 0.741433;
entrada[20][2] = 0.294219;
entrada[20][3] = 0.576670;
entrada[20][4] = 0.176532;
entrada[21][1] = 0.710669;
entrada[21][2] = 0.355335;
entrada[21][3] = 0.568535;
entrada[21][4] = 0.213201;
entrada[22][1] = 0.807796;
entrada[22][2] = 0.538530;
entrada[22][3] = 0.237587;
entrada[22][4] = 0.031678;
entrada[23][1] = 0.729924;
entrada[23][2] = 0.391031;
entrada[23][3] = 0.534409;
entrada[23][4] = 0.169447;
entrada[24][1] = 0.695956;
entrada[24][2] = 0.342784;
entrada[24][3] = 0.592082;
entrada[24][4] = 0.218135;
entrada[25][1] = 0.798370;
entrada[25][2] = 0.557353;
entrada[25][3] = 0.225954;
entrada[25][4] = 0.030127;
entrada[26][1] = 0.769869;
entrada[26][2] = 0.354140;
entrada[26][3] = 0.508113;
entrada[26][4] = 0.153974;
entrada[27][1] = 0.727662;
entrada[27][2] = 0.275331;
entrada[27][3] = 0.599829;
entrada[27][4] = 0.186832;
entrada[28][1] = 0.780109;
entrada[28][2] = 0.576603;
entrada[28][3] = 0.237425;
entrada[28][4] = 0.050877;
```

```
entrada[29][1] = 0.732396;
entrada[29][2] = 0.385472;
entrada[29][3] = 0.539660;
entrada[29][4] = 0.154189;
entrada[30][1] = 0.731225;
entrada[30][2] = 0.313382;
entrada[30][3] = 0.568730;
entrada[30][4] = 0.208921;
entrada[31][1] = 0.818031;
entrada[31][2] = 0.517530;
entrada[31][3] = 0.250418;
entrada[31][4] = 0.016695;
entrada[32][1] = 0.733509;
entrada[32][2] = 0.354530;
entrada[32][3] = 0.550132;
entrada[32][4] = 0.183377;
entrada[33][1] = 0.715295;
entrada[33][2] = 0.317909;
entrada[33][3] = 0.596079;
entrada[33][4] = 0.178824;
entrada[34][1] = 0.802185;
entrada[34][2] = 0.545486;
entrada[34][3] = 0.240655;
entrada[34][4] = 0.032087;
entrada[35][1] = 0.764442;
entrada[35][2] = 0.271254;
entrada[35][3] = 0.554837;
entrada[35][4] = 0.184946;
entrada[36][1] = 0.729654;
entrada[36][2] = 0.289545;
entrada[36][3] = 0.579090;
entrada[36][4] = 0.220054;
entrada[37][1] = 0.816094;
entrada[37][2] = 0.533600;
entrada[37][3] = 0.219718;
entrada[37][4] = 0.031388;
entrada[38][1] = 0.698580;
entrada[38][2] = 0.378891;
entrada[38][3] = 0.568336;
entrada[38][4] = 0.213126;
```

```
entrada[39][1] = 0.727126;
entrada[39][2] = 0.266613;
entrada[39][3] = 0.605938;
entrada[39][4] = 0.181781;
entrada[40][1] = 0.773811;
entrada[40][2] = 0.597328;
entrada[40][3] = 0.203635;
entrada[40][4] = 0.054303;
entrada[41][1] = 0.764350;
entrada[41][2] = 0.355818;
entrada[41][3] = 0.513959;
entrada[41][4] = 0.158141;
entrada[42][1] = 0.683079;
entrada[42][2] = 0.341540;
entrada[42][3] = 0.597694;
entrada[42][4] = 0.243957;
entrada[43][1] = 0.786991;
entrada[43][2] = 0.557452;
entrada[43][3] = 0.262330;
entrada[43][4] = 0.032791;
entrada[44][1] = 0.765219;
entrada[44][2] = 0.333914;
entrada[44][3] = 0.528696;
entrada[44][4] = 0.153044;
entrada[45][1] = 0.692552;
entrada[45][2] = 0.303751;
entrada[45][3] = 0.607502;
entrada[45][4] = 0.243001;
entrada[46][1] = 0.786090;
entrada[46][2] = 0.571702;
entrada[46][3] = 0.232254;
entrada[46][4] = 0.035731;
entrada[47][1] = 0.693334;
entrada[47][2] = 0.385186;
entrada[47][3] = 0.577778;
entrada[47][4] = 0.192593;
entrada[48][1] = 0.699970;
entrada[48][2] = 0.323867;
entrada[48][3] = 0.585050;
entrada[48][4] = 0.250736;
```

```
entrada[49][1] = 0.822106;
entrada[49][2] = 0.513816;
entrada[49][3] = 0.239781;
entrada[49][4] = 0.051382;
entrada[50][1] = 0.740886;
entrada[50][2] = 0.331740;
entrada[50][3] = 0.552900;
entrada[50][4] = 0.187986;
entrada[51][1] = 0.711712;
entrada[51][2] = 0.350022;
entrada[51][3] = 0.571703;
entrada[51][4] = 0.210013;
entrada[52][1] = 0.779649;
entrada[52][2] = 0.580915;
entrada[52][3] = 0.229308;
entrada[52][4] = 0.045862;
entrada[53][1] = 0.755193;
entrada[53][2] = 0.339290;
entrada[53][3] = 0.536296;
entrada[53][4] = 0.164172;
entrada[54][1] = 0.670541;
entrada[54][2] = 0.342113;
entrada[54][3] = 0.615803;
entrada[54][4] = 0.232637;
entrada[55][1] = 0.803274;
entrada[55][2] = 0.551267;
entrada[55][3] = 0.220507;
entrada[55][4] = 0.047251;
entrada[56][1] = 0.762630;
entrada[56][2] = 0.341869;
entrada[56][3] = 0.525952;
entrada[56][4] = 0.157786;
entrada[57][1] = 0.686190;
entrada[57][2] = 0.316703;
entrada[57][3] = 0.612293;
entrada[57][4] = 0.232249;
entrada[58][1] = 0.822250;
entrada[58][2] = 0.517713;
entrada[58][3] = 0.228403;
entrada[58][4] = 0.060907;
```

```
entrada[59][1] = 0.743148;
entrada[59][2] = 0.365055;
entrada[59][3] = 0.534545;
entrada[59][4] = 0.169490;
entrada[60][1] = 0.677679;
entrada[60][2] = 0.327156;
entrada[60][3] = 0.595890;
entrada[60][4] = 0.280419;
entrada[61][1] = 0.811209;
entrada[61][2] = 0.559454;
entrada[61][3] = 0.167836;
entrada[61][4] = 0.027973;
entrada[62][1] = 0.715249;
entrada[62][2] = 0.405308;
entrada[62][3] = 0.536437;
entrada[62][4] = 0.190733;
entrada[63][1] = 0.709537;
entrada[63][2] = 0.280080;
entrada[63][3] = 0.616177;
entrada[63][4] = 0.196056;
entrada[64][1] = 0.776114;
entrada[64][2] = 0.549747;
entrada[64][3] = 0.307212;
entrada[64][4] = 0.032338;
entrada[65][1] = 0.759117;
entrada[65][2] = 0.393114;
entrada[65][3] = 0.488004;
entrada[65][4] = 0.176224;
entrada[66][1] = 0.670175;
entrada[66][2] = 0.361682;
entrada[66][3] = 0.595711;
entrada[66][4] = 0.255305;
entrada[67][1] = 0.812284;
entrada[67][2] = 0.536107;
entrada[67][3] = 0.227439;
entrada[67][4] = 0.032491;
entrada[68][1] = 0.735443;
entrada[68][2] = 0.354589;
entrada[68][3] = 0.551582;
entrada[68][4] = 0.170728;
```

```
entrada[69][1] = 0.692991;
entrada[69][2] = 0.341996;
entrada[69][3] = 0.602992;
entrada[69][4] = 0.197997;
entrada[70][1] = 0.808466;
entrada[70][2] = 0.522134;
entrada[70][3] = 0.269489;
entrada[70][4] = 0.033686;
entrada[71][1] = 0.767011;
entrada[71][2] = 0.350634;
entrada[71][3] = 0.514993;
entrada[71][4] = 0.153402;
entrada[72][1] = 0.674671;
entrada[72][2] = 0.369981;
entrada[72][3] = 0.587616;
entrada[72][4] = 0.250281;
entrada[73][1] = 0.766939;
entrada[73][2] = 0.571445;
entrada[73][3] = 0.285722;
entrada[73][4] = 0.060152;
entrada[74][1] = 0.786675;
entrada[74][2] = 0.358834;
entrada[74][3] = 0.483046;
entrada[74][4] = 0.138013;
entrada[75][1] = 0.727852;
entrada[75][2] = 0.328707;
entrada[75][3] = 0.563498;
entrada[75][4] = 0.211312;
entrada[76][1] = 0.790965;
entrada[76][2] = 0.569495;
entrada[76][3] = 0.221470;
entrada[76][4] = 0.031639;
entrada[77][1] = 0.764673;
entrada[77][2] = 0.314865;
entrada[77][3] = 0.539769;
entrada[77][4] = 0.157433;
entrada[78][1] = 0.716539;
entrada[78][2] = 0.330710;
entrada[78][3] = 0.573231;
entrada[78][4] = 0.220474;
```

```
entrada[79][1] = 0.826474;
entrada[79][2] = 0.495885;
entrada[79][3] = 0.264472;
entrada[79][4] = 0.033059;
entrada[80][1] = 0.757281;
entrada[80][2] = 0.354212;
entrada[80][3] = 0.525211;
entrada[80][4] = 0.158785;
entrada[81][1] = 0.733379;
entrada[81][2] = 0.329489;
entrada[81][3] = 0.542063;
entrada[81][4] = 0.244460;
entrada[82][1] = 0.825123;
entrada[82][2] = 0.528079;
entrada[82][3] = 0.198030;
entrada[82][4] = 0.033005;
entrada[83][1] = 0.706319;
entrada[83][2] = 0.378385;
entrada[83][3] = 0.567578;
entrada[83][4] = 0.189193;
entrada[84][1] = 0.715765;
entrada[84][2] = 0.301964;
entrada[84][3] = 0.592743;
entrada[84][4] = 0.212493;
entrada[85][1] = 0.802124;
entrada[85][2] = 0.546903;
entrada[85][3] = 0.236991;
entrada[85][4] = 0.036460;
entrada[86][1] = 0.782581;
entrada[86][2] = 0.383618;
entrada[86][3] = 0.460341;
entrada[86][4] = 0.168792;
entrada[87][1] = 0.717182;
entrada[87][2] = 0.316404;
entrada[87][3] = 0.580073;
entrada[87][4] = 0.221483;
entrada[88][1] = 0.795948;
entrada[88][2] = 0.553703;
entrada[88][3] = 0.242245;
entrada[88][4] = 0.034606;
```

```
entrada[89][1] = 0.722330;
entrada[89][2] = 0.354829;
entrada[89][3] = 0.570260;
entrada[89][4] = 0.164742;
entrada[90][1] = 0.689149;
entrada[90][2] = 0.339431;
entrada[90][3] = 0.586291;
entrada[90][4] = 0.257145;


}

void recebe_target(void)
{

target[1][1] = 1.000000;
target[1][2] = 0.000000;
target[1][3] = 0.000000;
target[2][1] = 0.000000;
target[2][2] = 1.000000;
target[2][3] = 0.000000;
target[3][1] = 0.000000;
target[3][2] = 0.000000;
target[3][3] = 1.000000;
target[4][1] = 1.000000;
target[4][2] = 0.000000;
target[4][3] = 0.000000;
target[5][1] = 0.000000;
target[5][2] = 1.000000;
target[5][3] = 0.000000;
target[6][1] = 0.000000;
target[6][2] = 0.000000;
target[6][3] = 1.000000;
target[7][1] = 1.000000;
target[7][2] = 0.000000;
target[7][3] = 0.000000;
target[8][1] = 0.000000;
target[8][2] = 1.000000;
target[8][3] = 0.000000;
target[9][1] = 0.000000;
```

```
target[9][2] = 0.000000;
target[9][3] = 1.000000;
target[10][1] = 1.000000;
target[10][2] = 0.000000;
target[10][3] = 0.000000;
target[11][1] = 0.000000;
target[11][2] = 1.000000;
target[11][3] = 0.000000;
target[12][1] = 0.000000;
target[12][2] = 0.000000;
target[12][3] = 1.000000;
target[13][1] = 1.000000;
target[13][2] = 0.000000;
target[13][3] = 0.000000;
target[14][1] = 0.000000;
target[14][2] = 1.000000;
target[14][3] = 0.000000;
target[15][1] = 0.000000;
target[15][2] = 0.000000;
target[15][3] = 1.000000;
target[16][1] = 1.000000;
target[16][2] = 0.000000;
target[16][3] = 0.000000;
target[17][1] = 0.000000;
target[17][2] = 1.000000;
target[17][3] = 0.000000;
target[18][1] = 0.000000;
target[18][2] = 0.000000;
target[18][3] = 1.000000;
target[19][1] = 1.000000;
target[19][2] = 0.000000;
target[19][3] = 0.000000;
target[20][1] = 0.000000;
target[20][2] = 1.000000;
target[20][3] = 0.000000;
target[21][1] = 0.000000;
target[21][2] = 0.000000;
target[21][3] = 1.000000;
target[22][1] = 1.000000;
target[22][2] = 0.000000;
```

```
target[22][3] = 0.000000;
target[23][1] = 0.000000;
target[23][2] = 1.000000;
target[23][3] = 0.000000;
target[24][1] = 0.000000;
target[24][2] = 0.000000;
target[24][3] = 1.000000;
target[25][1] = 1.000000;
target[25][2] = 0.000000;
target[25][3] = 0.000000;
target[26][1] = 0.000000;
target[26][2] = 1.000000;
target[26][3] = 0.000000;
target[27][1] = 0.000000;
target[27][2] = 0.000000;
target[27][3] = 1.000000;
target[28][1] = 1.000000;
target[28][2] = 0.000000;
target[28][3] = 0.000000;
target[29][1] = 0.000000;
target[29][2] = 1.000000;
target[29][3] = 0.000000;
target[30][1] = 0.000000;
target[30][2] = 0.000000;
target[30][3] = 1.000000;
target[31][1] = 1.000000;
target[31][2] = 0.000000;
target[31][3] = 0.000000;
target[32][1] = 0.000000;
target[32][2] = 1.000000;
target[32][3] = 0.000000;
target[33][1] = 0.000000;
target[33][2] = 0.000000;
target[33][3] = 1.000000;
target[34][1] = 1.000000;
target[34][2] = 0.000000;
target[34][3] = 0.000000;
target[35][1] = 0.000000;
target[35][2] = 1.000000;
target[35][3] = 0.000000;
```

```
target[36][1] = 0.000000;
target[36][2] = 0.000000;
target[36][3] = 1.000000;
target[37][1] = 1.000000;
target[37][2] = 0.000000;
target[37][3] = 0.000000;
target[38][1] = 0.000000;
target[38][2] = 1.000000;
target[38][3] = 0.000000;
target[39][1] = 0.000000;
target[39][2] = 0.000000;
target[39][3] = 1.000000;
target[40][1] = 1.000000;
target[40][2] = 0.000000;
target[40][3] = 0.000000;
target[41][1] = 0.000000;
target[41][2] = 1.000000;
target[41][3] = 0.000000;
target[42][1] = 0.000000;
target[42][2] = 0.000000;
target[42][3] = 1.000000;
target[43][1] = 1.000000;
target[43][2] = 0.000000;
target[43][3] = 0.000000;
target[44][1] = 0.000000;
target[44][2] = 1.000000;
target[44][3] = 0.000000;
target[45][1] = 0.000000;
target[45][2] = 0.000000;
target[45][3] = 1.000000;
target[46][1] = 1.000000;
target[46][2] = 0.000000;
target[46][3] = 0.000000;
target[47][1] = 0.000000;
target[47][2] = 1.000000;
target[47][3] = 0.000000;
target[48][1] = 0.000000;
target[48][2] = 0.000000;
target[48][3] = 1.000000;
target[49][1] = 1.000000;
```

```
target[49][2] = 0.000000;
target[49][3] = 0.000000;
target[50][1] = 0.000000;
target[50][2] = 1.000000;
target[50][3] = 0.000000;
target[51][1] = 0.000000;
target[51][2] = 0.000000;
target[51][3] = 1.000000;
target[52][1] = 1.000000;
target[52][2] = 0.000000;
target[52][3] = 0.000000;
target[53][1] = 0.000000;
target[53][2] = 1.000000;
target[53][3] = 0.000000;
target[54][1] = 0.000000;
target[54][2] = 0.000000;
target[54][3] = 1.000000;
target[55][1] = 1.000000;
target[55][2] = 0.000000;
target[55][3] = 0.000000;
target[56][1] = 0.000000;
target[56][2] = 1.000000;
target[56][3] = 0.000000;
target[57][1] = 0.000000;
target[57][2] = 0.000000;
target[57][3] = 1.000000;
target[58][1] = 1.000000;
target[58][2] = 0.000000;
target[58][3] = 0.000000;
target[59][1] = 0.000000;
target[59][2] = 1.000000;
target[59][3] = 0.000000;
target[60][1] = 0.000000;
target[60][2] = 0.000000;
target[60][3] = 1.000000;
target[61][1] = 1.000000;
target[61][2] = 0.000000;
target[61][3] = 0.000000;
target[62][1] = 0.000000;
target[62][2] = 1.000000;
```

```
target[62][3] = 0.000000;
target[63][1] = 0.000000;
target[63][2] = 0.000000;
target[63][3] = 1.000000;
target[64][1] = 1.000000;
target[64][2] = 0.000000;
target[64][3] = 0.000000;
target[65][1] = 0.000000;
target[65][2] = 1.000000;
target[65][3] = 0.000000;
target[66][1] = 0.000000;
target[66][2] = 0.000000;
target[66][3] = 1.000000;
target[67][1] = 1.000000;
target[67][2] = 0.000000;
target[67][3] = 0.000000;
target[68][1] = 0.000000;
target[68][2] = 1.000000;
target[68][3] = 0.000000;
target[69][1] = 0.000000;
target[69][2] = 0.000000;
target[69][3] = 1.000000;
target[70][1] = 1.000000;
target[70][2] = 0.000000;
target[70][3] = 0.000000;
target[71][1] = 0.000000;
target[71][2] = 1.000000;
target[71][3] = 0.000000;
target[72][1] = 0.000000;
target[72][2] = 0.000000;
target[72][3] = 1.000000;
target[73][1] = 1.000000;
target[73][2] = 0.000000;
target[73][3] = 0.000000;
target[74][1] = 0.000000;
target[74][2] = 1.000000;
target[74][3] = 0.000000;
target[75][1] = 0.000000;
target[75][2] = 0.000000;
target[75][3] = 1.000000;
```

```
target[76][1] = 1.000000;
target[76][2] = 0.000000;
target[76][3] = 0.000000;
target[77][1] = 0.000000;
target[77][2] = 1.000000;
target[77][3] = 0.000000;
target[78][1] = 0.000000;
target[78][2] = 0.000000;
target[78][3] = 1.000000;
target[79][1] = 1.000000;
target[79][2] = 0.000000;
target[79][3] = 0.000000;
target[80][1] = 0.000000;
target[80][2] = 1.000000;
target[80][3] = 0.000000;
target[81][1] = 0.000000;
target[81][2] = 0.000000;
target[81][3] = 1.000000;
target[82][1] = 1.000000;
target[82][2] = 0.000000;
target[82][3] = 0.000000;
target[83][1] = 0.000000;
target[83][2] = 1.000000;
target[83][3] = 0.000000;
target[84][1] = 0.000000;
target[84][2] = 0.000000;
target[84][3] = 1.000000;
target[85][1] = 1.000000;
target[85][2] = 0.000000;
target[85][3] = 0.000000;
target[86][1] = 0.000000;
target[86][2] = 1.000000;
target[86][3] = 0.000000;
target[87][1] = 0.000000;
target[87][2] = 0.000000;
target[87][3] = 1.000000;
target[88][1] = 1.000000;
target[88][2] = 0.000000;
target[88][3] = 0.000000;
target[89][1] = 0.000000;
```

```
target[89][2] = 1.000000;
target[89][3] = 0.000000;
target[90][1] = 0.000000;
target[90][2] = 0.000000;
target[90][3] = 1.000000;


}

void exporta_pesos()
{

   int l, m;
   FILE *cfPtr;
   cfPtr=fopen("pesos.dat", "w");
   for (m=0; m<=NUMEROESCONDIDA+1; m++)
   {
      for (l=0; l<=NUMEROENTRADAS+1; l++)
         fprintf(cfPtr,"\nW12[%d][%d]=%f;", l, m, W12[l][m]);
         //fprintf(cfPtr, "\n");
   }



   for (m=0; m<=NUMEROSAIDAS+1; m++)
   {
      for (l=0; l<=NUMEROESCONDIDA+1; l++)
         fprintf(cfPtr,"\nW23[%d][%d]=%f;", l, m, W23[l][m]);
         //fprintf(cfPtr, "\n");
   }
   fclose(cfPtr);
}

int main(void) {

       FILE *cfPtr;
   recebe_entradas();
   recebe_target();
```

```
   cfPtr=fopen("erro.dat", "w");
   for( j = 1 ; j <= NumEscondida ; j++ ) {    /* inicializa W12 e DeltaW12 */
      for( i = 0 ; i <= NumEntrada ; i++ ) {
         DeltaW12[i][j] = 0.0 ;
         W12[i][j] = 2.0 * ( rando() - 0.5 ) * wmax ;
      }
   }
   for( k = 1 ; k <= NumSAIDA ; k ++ ) {    /* inicializa W23 e DeltaW23 */
      for( j = 0 ; j <= NumEscondida ; j++ ) {
         DeltaW23[j][k] = 0.0 ;
         W23[j][k] = 2.0 * ( rando() - 0.5 ) * wmax ;
      }
   }

   for( epoca = 0 ; epoca < NUMEROITERACOES ; epoca++) {     /* faz a iteração da atualização dos
pesos */
      for( p = 1 ; p <= NumPadroes ; p++ ) {    /* randomiza a ordem dos indivíduos */
         ranpad[p] = p ;
      }
      for( p = 1 ; p <= NumPadroes ; p++) {
         np = p + rando() * ( NumPadroes + 1 - p ) ;
         op = ranpad[p] ; ranpad[p] = ranpad[np] ; ranpad[np] = op ;
      }
      Erro = 0.0 ;
      for( np = 1 ; np <= NumPadroes ; np++ ) {    /* repete para todos os padrões de treinamento */
         p = ranpad[np];
         for( j = 1 ; j <= NumEscondida ; j++ ) {    /* computa as ativações da unidade escondida */
            SomaEscondida[p][j] = W12[0][j] ;
            for( i = 1 ; i <= NumEntrada ; i++ ) {
               SomaEscondida[p][j] += entrada[p][i] * W12[i][j] ;
            }
            Escondida[p][j] = 1.0/(1.0 + exp(-SomaEscondida[p][j])) ;
         }
         for( k = 1 ; k <= NumSAIDA ; k++ ) {    /* computa as unidades de ativação da saída e erros */
            SomaSaida[p][k] = W23[0][k] ;
            for( j = 1 ; j <= NumEscondida ; j++ ) {
               SomaSaida[p][k] += Escondida[p][j] * W23[j][k] ;
            }
            SAIDA[p][k] = 1.0/(1.0 + exp(-SomaSaida[p][k])) ;   /* Sigmoidal SAIDAs */
```

```
          Erro += 0.5 * (target[p][k] - SAIDA[p][k]) * (target[p][k] - SAIDA[p][k]) ;   /* SSE */
          DeltaS[k] = (target[p][k] - SAIDA[p][k]) * SAIDA[p][k] * (1.0 - SAIDA[p][k]) ;   /*erro saida sigmoide */

        }
        for( j = 1 ; j <= NumEscondida ; j++ ) {    /* retropropagação de erros para a camada escondida */
          somaDWS[j] = 0.0 ;
          for( k = 1 ; k <= NumSAIDA ; k++ ) {
            somaDWS[j] += W23[j][k] * DeltaS[k] ;
          }
          DeltaE[j] = somaDWS[j] * Escondida[p][j] * (1.0 - Escondida[p][j]) ;
        }
        for( j = 1 ; j <= NumEscondida ; j++ ) {     /* atualiza pesos w12 */
          DeltaW12[0][j] = eta * DeltaE[j] + alpha * DeltaW12[0][j] ;
          W12[0][j] += DeltaW12[0][j] ;
          for( i = 1 ; i <= NumEntrada ; i++ ) {
            DeltaW12[i][j] = eta * entrada[p][i] * DeltaE[j] + alpha * DeltaW12[i][j];
            W12[i][j] += DeltaW12[i][j] ;
          }
        }
        for( k = 1 ; k <= NumSAIDA ; k ++ ) {    /* atualiza pesos W23 */
          DeltaW23[0][k] = eta * DeltaS[k] + alpha * DeltaW23[0][k] ;
          W23[0][k] += DeltaW23[0][k] ;
          for( j = 1 ; j <= NumEscondida ; j++ ) {
            DeltaW23[j][k] = eta * Escondida[p][j] * DeltaS[k] + alpha * DeltaW23[j][k] ;
            W23[j][k] += DeltaW23[j][k] ;
          }
        }
      }
      fprintf(cfPtr, "\nepoca %-5d :   Erro = %f", epoca, Erro) ;
      if( Erro < 0.0004 ) break ;  /* pára o aprendizado quando o erro convergir para o valor descrito */
    }
    fclose(cfPtr);
    exporta_pesos();


    cfPtr = fopen("resultados.dat", "w");

    fprintf(cfPtr, "\n\nNETWORK DATA - epoca %d\n\nPat\t", epoca) ;   /* mostra as SAIDAs */
    for( i = 1 ; i <= NumEntrada ; i++ ) {
```

```
      fprintf(cfPtr, "ENTRADA%-4d\t", i) ;
    }
    for( k = 1 ; k <= NumSAIDA ; k++ ) {
      fprintf(cfPtr, "target%-4d\tSAIDA%-4d\t", k, k) ;
    }
    for( p = 1 ; p <= NumPadroes ; p++ ) {
    fprintf(cfPtr, "\n%d\t", p) ;
      for( i = 1 ; i <= NumEntrada ; i++ ) {
        fprintf(cfPtr, "%f\t", entrada[p][i]) ;
      }
      for( k = 1 ; k <= NumSAIDA ; k++ ) {
        fprintf(cfPtr, "%f\t%f\t", target[p][k], SAIDA[p][k]) ;
      }
    }
    fprintf(cfPtr, "\n\nPassar bem!\n\n") ;



    fclose(cfPtr);





    return 1 ;
  }
```