UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
BACHARELADO EM ENGENHARIA ELETRÔNICA

GUILHERME AUGUSTO PASCHOAL DA SILVA

IMPLEMENTAÇÃO DE REDES NEURAIS ARTIFICIAIS UTILIZANDO VHDL PARA APLICAÇÕES EM RECONHECIMENTO E CLASSIFICAÇÃO DE PADRÕES

TRABALHO DE CONCLUSÃO DE CURSO

CAMPO MOURÃO
2019

GUILHERME AUGUSTO PASCHOAL DA SILVA

IMPLEMENTAÇÃO DE REDES NEURAIS ARTIFICIAIS UTILIZANDO VHDL PARA APLICAÇÕES EM RECONHECIMENTO E CLASSIFICAÇÃO DE PADRÕES

Trabalho de Conclusão de Curso de Graduação, apresentado à disciplina de TCC2, do Curso Superior de Engenharia Eletrônica do Departamento Acadêmico de Eletrônica – DAELN, da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para obtenção do título de Engenheiro em Eletrônica.

Orientador: Prof.Dr. Marcio Rodrigues da Cunha

CAMPO MOURÃO

2019

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Campus Campo Mourão
Coordenação de Engenharia Eletrônica

TERMO DE APROVAÇÃO DO TRABALHO DE CONCLUSÃO DE CURSO INTITULADO

IMPLEMENTAÇÃO DE REDES NEURAIS ARTIFICIAIS UTILIZANDO VHDL PARA APLICAÇÕES EM RECONHECIMENTO E CLASSIFICAÇÃO DE PADRÕES

DO(A) DISCENTE

GUILHERME AUGUSTO PASCHOAL DA SILVA

Trabalho de Conclusão de Curso apresentado no dia 29 de novembro de 2019 ao Curso Superior de Engenharia Eletrônica da Universidade Tecnológica Federal do Paraná, Campus Campo Mourão. O discente foi arguido pela Comissão Examinadora composta pelos professores abaixo assinados. Após deliberação, a comissão considerou o trabalho aprovado com alterações.

Prof. André Luiz Regis Monteiro
UTFPR

Prof. Leandro Castilho Brolin
UTFPR

Prof. Marcio Rodrigues da Cunha
Orientador
UTFPR

RESUMO

O escopo deste trabalho é o desenvolvimento de RNAs em *hardware*, para resolver problemas de reconhecimento e classificação de padrões, utilizando a linguagem de descrição VHDL. Duas redes diferentes foram implementadas, com objetivo de abordar topologias distintas com diferentes funções de ativação. Primeiramente o projeto foi desenvolvido e treinado em *software*, utilizando o MATLAB®. Após o treinamento de ambas as redes e com a aquisição dos parâmetros de pesos e *bias*, partiu-se para a implementação em *hardware*, utilizando o mesmo fluxo de processamento que em *software*. A descrição do circuito de ambas as RNAs foram feitas utilizando o conceito totalmente paralelo. Após a implementação, realizou-se simulações e testes comparativos entre a implementação sequencial no MATLAB® e a paralela em VHDL. As duas redes descritas tiveram a mesma taxa de acerto que as em *software*, porém com um tempo mais rápido de execução e com pouco recurso de *hardware* utilizado.

Palavras-Chave: Redes Neurais Artificiais, VHDL, FPGA, classificação de padrões.

ABSTRACT

The scope of this paper is the development of artificial neural network in hardware, to solve problems of pattern recognition and classification, using VHDL. Two different networks were implemented, with the objective of showing two topologies and different activation function. Firstly, the project was developed and trained in software using MATLAB®. After the training of both networks and the acquisition of the parameters of weight and *bias*, started the implementation in hardware, using the same processing flow as in software. The circuit description of both ANNs was made using the trough parallel concept. After the implementation, simulations and comparative tests were made between the sequential implementation in MATLAB® and the parallel in VHDL. The two described networks had the same hit rate as those in software, but with a faster execution time and with little hardware resource utilization.

Keywords: Artificial Neural Networks, VHDL, FPGA, pattern classification.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS, SIGLAS E ACRÔNIMOS

IA	Inteligência Artificial
IAS	Inteligência Artificial simbólica
IAC	Inteligencia Artificial conexionista
RNA	Redes Neurais Artificiais
PLD	*Programmable Logic Device*s (Dispositivos Lógicos Programáveis)
FPGA	*Field Programmable Gate Array* (Matriz de portas programáveis em campo)
VHDL	VHSIC *Hardware Description Language* (VHSIC Linguagem de descrição de hardware)
MATLAB®	*Matrix Laboratory* (Laboratório de Matrizes)
CLB	*Configuration Logical Blocks* (Blocos Lógicos de configuração)
DSP	*Digital Signal Processor* (Processador Digital de Sinais)
VHSIC	*Very High Speed Integrated Circuit* (Circuito Integrado de alta velocidade)
IEEE	Institute of Eletrical and Eletronic Engineers (Instituto de Engenheiros Eletricistas e Eletrônicos)

SUMÁRIO

1 INTRODUÇÃO

Construir um dispositivo que possua autonomia e inteligência é um sonho antigo dos pesquisadores. Em particular, o campo da Inteligência Artificial (IA) procura entender como ocorre a percepção, compreensão, prevenção e manipulação das coisas do mundo através da experiência humana, e então, a partir deste conhecimento, construir dispositivos inteligentes com estudos que vão desde os processos de raciocínio, até o que se refere aos comportamentos dos seres vivos (RUSSEL; NORVIG, 2013).

No mundo moderno, as Redes Neurais Artificiais (RNAs), que fazem parte de uma ramificação da IA, estão sendo cada vez mais utilizadas em diversos campos de pesquisa, seja para desenvolvimento ou para aplicação dessas redes. Segundo Silva, Spatti e Flauzino (2010), as RNAs são modelos computacionais inspirados nas redes neuronais biológicas, e suas principais características são a capacidade de aprendizado e generalização. Sendo assim, essa tecnologia está presente em diversas aplicações, como: modelagem, análise de séries temporais, processamentos de sinais e controle, e reconhecimento e classificação de padrões.

Atualmente, têm-se as RNAs como uma tecnologia prática, com sucesso em diversas aplicações reais. A maioria destas aplicações envolvem resoluções de problemas relacionados a reconhecimento e classificação de padrões, onde se necessita classificar em categoria ou grupo um conjunto de entradas, como por exemplo: visão de máquina, reconhecimento de caracteres, diagnóstico computacional, reconhecimento de fala, entre outros (THEODORIDIS; KOUTROUMBAS, 2003).

Há correntes de pesquisas sobre interfaces de desenvolvimento para atuarem com RNA devido à grande utilização de tal ferramenta no ambiente científico. A aplicação de RNAs em *software* permite um alto nível de abstração, com isso, torna-se mais fácil sua implementação, porém o processamento é feito de forma sequencial. Já em *hardware,* permite-se o embarque de um circuito dedicado a realização das tarefas de uma RNA, sendo sua implementação mais difícil, porém o processamento dos dados são feitos de forma paralela (PRADO, 2011).

Para a aplicação da rede em *hardware* há ferramentas que possibilitam o desenvolvimento de sistemas digitais reconfiguráveis, que permitem prototipagem em circuitos, como os *Field Programmable Gate Arrays* (FPGAs). Estes circuitos integrados possibilitam a criação de projetos descritos em linguagem de descrição de *hardware*, como o VHDL e Verilog (PRADO, 2011). Estas linguagens, segundo Pedroni (2010) e D'Amore (2012), descrevem como deve ser o comportamento ou estrutura do circuito, e então, a partir de um compilador, o *hardware* físico é inferido.

Os FPGAs apontam novos rumos para o desenvolvimento das Redes Neurais Artificiais, pois através do paralelismo na execução das tarefas torna-se mais similar ao cérebro biológico, sendo isso impossível de se reproduzir pela implementação de *softwares* em processadores sequenciais (HARIPRASATH, PRABAKAR, 2012).

Neste contexto, o presente trabalho apresenta o processo de implementação e demonstração da aplicabilidade das Redes Neurais Artificiais em *hardware,* para problemas que envolvem reconhecimento e classificação de padrões, utilizando o VHDL e FPGA. Serão abordadas dois tipos de RNAs, cada uma com um tipo de complexidade diferente em relação a arquitetura da rede e função de ativação.

1.1 OBJETIVOS

Nesta seção são apresentados o objetivo geral e os objetivos específicos do trabalho.

1.1.1 OBJETIVO GERAL

Implementação de redes neurais artificiais utilizando a linguagem de descrição de *hardware* VHDL, em FPGA, para aplicações de reconhecimento e classificação de padrões.

1.1.2 OBJETIVOS ESPECÍFICOS

Abaixo são apresentados os objetivos específicos do trabalho:

- Implementar dois tipos de redes com função de ativação e arquiteturas diferentes;
- Desenvolver um programa para treinamento e simulação das redes neurais artificiais no *software* MATLAB®;
- Desenvolver o código em VHDL das redes com base nos parâmetros estabelecidos no programa desenvolvido no MATLAB®;
- Simular o *hardware* descrito utilizando o *software Quartus Prime*;
- Demonstrar a eficiência da rede em *hardware* através de testes comparativos entre o *hardware* e *software*;
- Validar a rede e sua eficiência em problemas envolvendo reconhecimento de classificação de padrões.

1.1.3 JUSTIFICATIVA

As redes neurais artificiais não são uma tecnologia recente, porém, atualmente, há diversos trabalhos sendo realizados para tornar sua implementação mais simples e aplicável no mundo real, sendo interessante estudar maneiras de contribuir para que isso ocorra.

Uma boa contribuição é estudar seu desenvolvimento utilizando VHDL para síntese em FPGA, pois sabendo que as RNAs são baseadas em redes neurais biológicas e que necessitam de alto nível de paralelismo, o mesmo possui grande quantidade de elementos lógicos que possibilitam a implementação de diferentes arquiteturas digitais, e permite execução das operações paralelas em nível de hardware, fazendo com que sua velocidade de processamento seja muito maior quando comparado com os computadores comuns, que operam de forma sequencial.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo será abordada, de forma explicativa, a revisão bibliográfica dos temas necessários para compreensão e desenvolvimento do trabalho, como redes neurais artificiais, formas de aprendizado, reconhecimento de padrões, FPGA e VHDL.

2.1 REDES NEURAIS ARTIFICIAIS

Quando se trata de inteligência artificial existem dois paradigmas, a inteligência artificial simbólica e a conexionista, que nasceram simultaneamente após o encontro no *Darth-mouth College*, em 1956, sendo o primeiro esforço conjunto para o estudo de IA, onde foi publicado o livro *Automata Studies*, no qual o primeiro artigo abordava as redes neurais artificiais como um paradigma da inteligência artificial (BARRETO, 2002).

Na IAS (inteligência artificial simbólica), a simulação do comportamento inteligente global é feita sem considerar os agentes responsáveis por tal comportamento. Na IAC (inteligência artificial conexionista) se acredita que construindo uma máquina que imite a estrutura do cérebro, ela apresentará comportamento inteligente. Então, com o passar do tempo, as duas correntes se separaram e os estudos sobre redes neurais, que faz parte da conexionista, andaram lentamente, enquanto, a corrente da IAS acelerou (BARRETO, 2002).

Em 1943, a primeira publicação sobre neurocomputação foi através do artigo elaborado por McCullosh & Pitts. Onde tratavam sobre o primeiro modelamento matemático inspirado no neurônio biológico, o que resultou na primeira concepção de neurônio artificial, sem preocupações com técnicas de aprendizado. Então, em 1949, foi proposto o primeiro método de treinamento para RNAs, sendo denominado como regra de aprendizado de Hebb, baseado em observações de caráter neurofisiológico (SILVA; SPATTI; FLAUZINO, 2010).

Entre 1957 e 1958, houve a concepção do modelo *Perceptron*, por Frank Rosenblatt, que desenvolveu o primeiro neurocomputador, chamado de *Mark I – Perceptron* (SILVA; SPATTI; FLAUZINO, 2010). Motivados por este modelo, Widrow e Hoff desenvolveram a rede *Adaline* que consistia na variação do algoritmo de aprendizagem do *Perceptron* (WIDROW; LEHR, 1990). Porém, em

1969, Minsky e Papert demonstraram as limitações destes modelos de única camada em analisar problemas não linearmente separáveis (HAYKIN, 2001).

Apesar da diminuição drástica nos estudos sobre redes neurais nas décadas de 70 e 80, um trabalho de grande relevância foi publicado por John Hopfield em 1982, que tratava de redes recorrentes, estimulando novamente o interesse dos pesquisadores pelo assunto (HAYKIN, 2001). Em 1986, através de Rumelhart, Hinton e Willians surgiu o algoritmo *backpropagation*, propiciando o treinamento das redes *Perceptron* Multi-camadas e resultando em uma grande capacidade de generalização (AGUIAR, 2010 apud RUMELHART et al., 1986).

Recentemente, diversos estudos em diferentes ramos do conhecimento têm permitido o avanço em relação à teoria sobre redes neurais artificiais, tendo como destaques a proposição de algoritmos de aprendizado baseados no método de Levenberg-Marquart, que permitem o treinamento das redes para diversas aplicações (SILVA; SPATTI; FLAUZINO, 2010 apud HAGAN; MENHAJ, 1994).

Diversas outras informações, relacionadas ao contexto histórico das redes neurais, explicado de forma detalhada, podem ser encontradas em Haykin (2001).

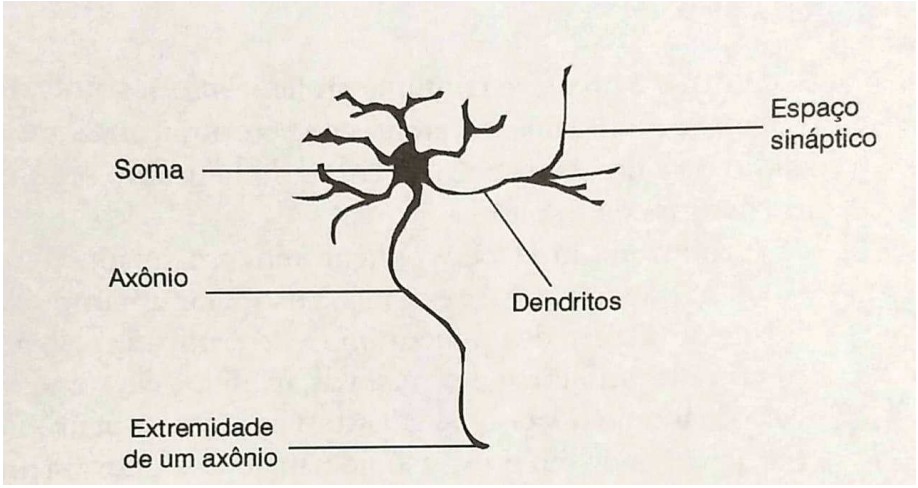
2.1.1 NEURÔNIO BIOLÓGICO

No cérebro humano, o processamento de informações é feito através de elementos biológicos operando em paralelo, com o objetivo de produzir ações apropriadas, como pensar e memorizar, a partir de um estímulo (SILVA; SPATTI; FLAUZINO, 2010).

O corpo celular do neurônio é chamado de soma, e possui diversas ramificações. As ramificações são chamadas de dendritos, que conduzem os sinais das extremidades para o corpo celular. Há também uma ramificação denominada axônio, com a função de conduzir os sinais do corpo da célula para suas extremidades. Suas extremidades são conectadas com outros neurônios através de ligações chamadas de sinapses. As sinapses possuem um papel fundamental na memorização da informação (BARRETO, 2002).

A Figura 1 representa de maneira ilustrativa um neurônio biológico, conforme citado acima.

Figura 1 - Representação de um neurônio biológico.



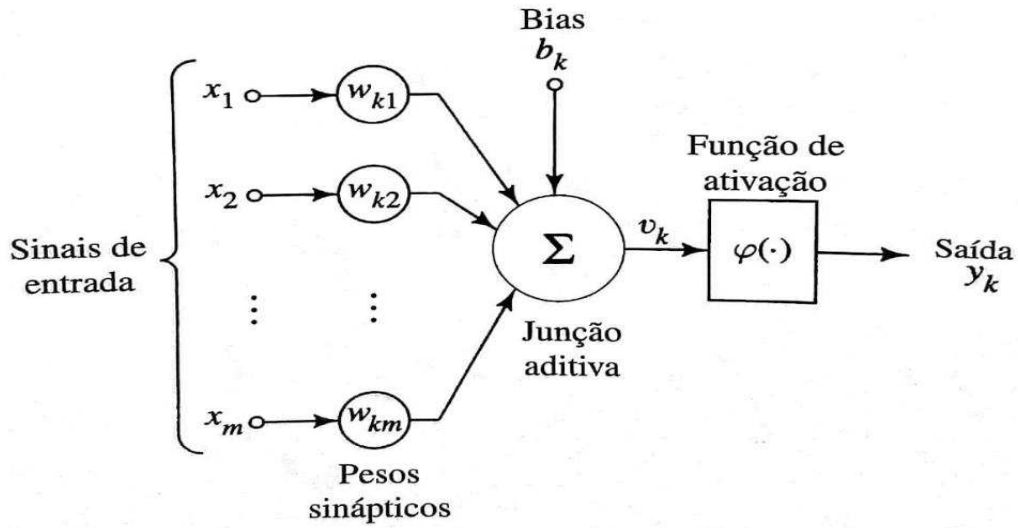
Fonte: Braga (2011, p.7).

2.1.2 NEURÔNIO ARTIFICIAL

Silva, Spatti e Flauzino (2010), dizem que os modelos computacionais chamados de neurônio artificial são modelos simplificados dos neurônios biológicos. Porém, segundo Haykin (2001) os neurônios artificiais, que são utilizados para construir as redes neurais, são extremamente primitivos em comparação aos encontrados no cérebro. O modelo mais simples de neurônio artificial e que engloba as principais características de uma rede neural biológica, foi proposto por McCulloch e Pitts em 1943 (SILVA; SPATTI; FLAUZINO, 2010).

Tal modelo funciona da seguinte maneira. Há *n* terminais de entrada que representam os dendritos, e um terminal de saída representando o axônio. Para simular as sinapses cada sinal de entrada é multiplicado pelo seu respectivo peso sináptico (WOLF, 2001). Um combinador linear faz a soma dos sinais de entrada e uma função de ativação restringe a amplitude de saída do neurônio a um valor finito. O *bias* tem a função de aumentar ou diminuir a entrada líquida da função de ativação, dependendo se ele é positivo ou negativo, respectivamente (HAYKIN, 2001). A Figura 2 representa o modelo básico citado neste parágrafo.

Figura 2 - Modelo básico de um neurônio.



Fonte: Haykin (2001, p.36).

Segundo Haykin (2001), pode-se escrever este neurônio em modelo matemático da seguinte maneira:

$$v_k = b_k + \sum_{m=1}^{n} w_m x_m \tag{2.1}$$

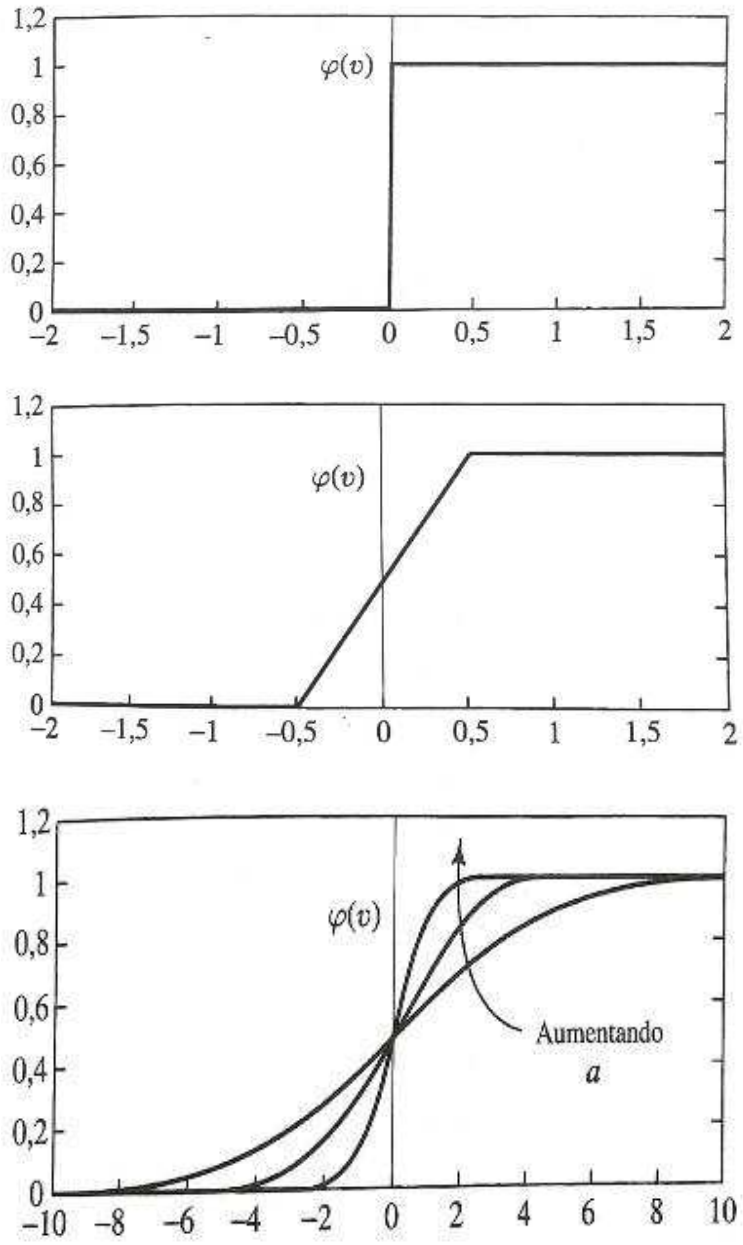
e

$$y_k = \varphi(v_k) \tag{2.2}$$

Onde x_m são os sinais de entrada. Os pesos sinápticos são representados por w_m. A saída do combinador linear é v_k. O *bias* é b_k. A função de ativação é representada por $\varphi_{(.)}$, e o sinal de saída por y_k.

Como consta em Haykin (2001), um exemplo de três tipos básicos de funções de ativação são: função de limiar, função linear por partes e função sigmóide, a função tangente hiperbólica é comumente utilizada no lugar da sigmóide. A Figura 3 representa os gráficos das funções de ativação, respectivamente.

Figura 3 - Funções de ativação: Função de limiar; Função linear por partes; Função sigmoide.



Fonte: Haykin (2001, p.39).

2.1.3 ARQUITETURAS DE RNAs

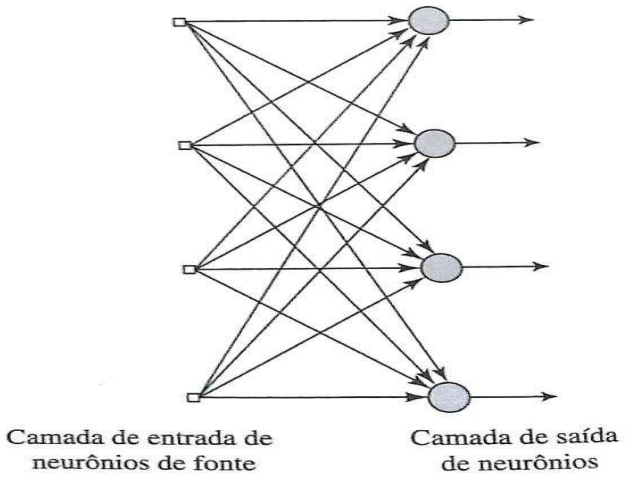
A forma como os neurônios estão interligados uns em relação aos outros é que define a arquitetura de uma rede neural artificial. A topologia é definida como sendo as diferentes formas estruturais que as redes podem assumir. Por exemplo, em uma arquitetura pode haver dois tipos de topologia com diferentes quantidades de neurônios e, ou diferentes tipos de função de ativação (SILVA; SPATTI; FLAUZINO, 2010).

Em geral, segundo Haykin (2001), podemos citar três classes diferentes de arquiteturas de rede. São elas: redes alimentadas diretamente com camada única, redes alimentadas diretamente com múltiplas camadas e redes recorrentes.

As redes alimentadas diretamente com camada única, ou também chamada de redes *feedforward* de camada simples, possui apenas uma camada de entrada e uma camada de neurônios, que é a própria camada de saída (SILVA; SPATTI; FLAUZINO, 2010). O termo “camada única” é referente a camada de saída, pois não é levado em conta a camada de entrada, por esta não realizar qualquer operação (HAYKIN, 2001). Segundo Silva, Spatti e Flauzino (2010), essas redes são comumente empregadas envolvendo classificação de padrões com dados linearmente separáveis e filtragem linear. A Figura 4 ilustra a arquitetura *feedfoward* de única camada.

As Redes Alimentadas Diretamente com Múltiplas Camadas, também chamadas de redes *feedforward* de Múltiplas Camadas, se diferenciam por ter uma ou mais camadas ocultas, o qual possuem nós chamados de neurônios ocultos. Ao adicionar mais camadas ocultas, a rede adquire maior capacidade de generalizar problemas, devido ao aumento das conexões sinápticas e das interações neurais (HAYKIN, 2001).

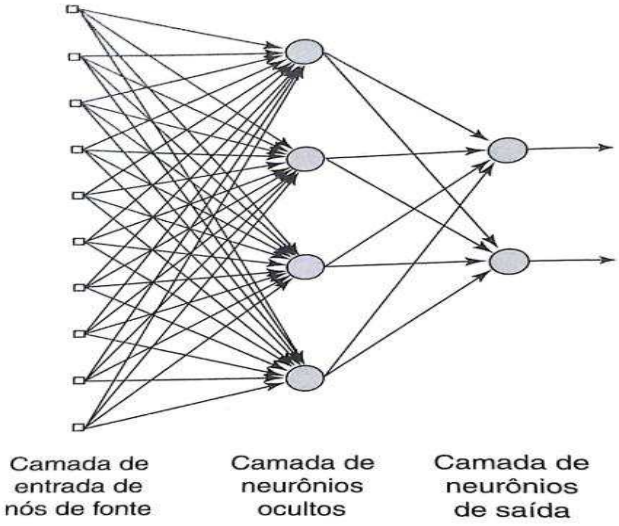
Figura 4 - Rede alimentada adiante com camada única.



Fonte: Haykin (2001, p.47).

A Figura 5 representa uma rede *feedforward* de múltiplas camadas, com dez neurônios de entrada, quatro neurônios ocultos e dois de saída.

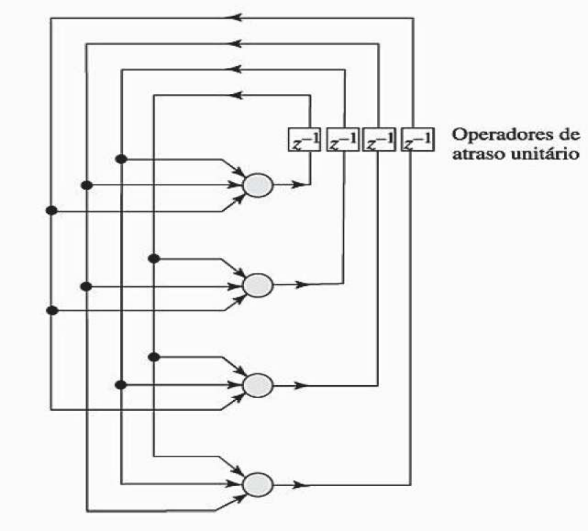
Figura 5 - Rede feedforward de múltiplas camadas.



Fonte: Haykin (2001, p.48).

A Figura 6 representa uma Rede Recorrente, que se diferencia das *feedfowards*, por ter pelo menos um laço de realimentação (HAYKIN, 2001). A realimentação torna a rede capaz de realizar processamento dinâmico, sendo assim, podem ser utilizadas em sistemas variantes em relação ao tempo e não lineares (SILVA; SPATTI; FLAUZINO, 2010).

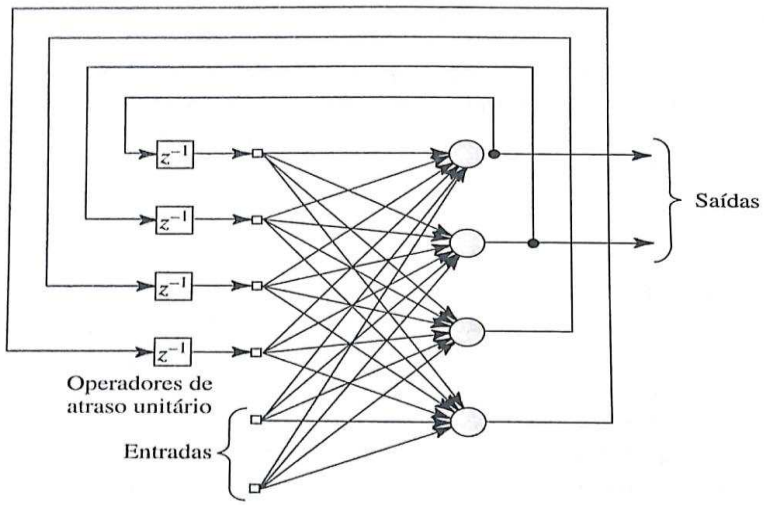
Figura 6 - Rede recorrente sem camadas ocultas.



Fonte: Haykin (2001, p.48).

A arquitetura seguinte, se difere da anterior pois possui camadas ocultas, assim como é apresentado na Figura 7. Tanto nas redes recorrentes de camada única, quanto nas recorrentes de múltiplas camadas, a presença de um laço de realimentação tem grande impacto na capacidade de aprendizagem e no desempenho da rede (HAYKIN, 2001). Tanto a arquitetura apresentada na Figura 6, quanto na Figura 7, são bastante utilizadas em sistemas de previsões temporais (SILVA; SPATTI; FLAUZINO, 2010).

Figura 7 - Rede recorrente com neurônios ocultos.



Fonte: Haykin (2001, p.49).

2.1.4 PROCESSOS DE APRENDIZAGEM

Para que uma rede neural seja utilizada para a solução de um problema, o passo inicial é a etapa de aprendizagem. Durante esta fase, a rede extrai informações do ambiente a qual está introduzida, e cria representações particulares na forma de pesos sinápticos, associados as conexões entre os neurônios (WOLF, 2001).

Segundo Braga (2011), existem basicamente dois paradigmas de aprendizado. O aprendizado supervisionado e o não supervisionado, também chamados de aprendizagem com um professor e aprendizagem sem um professor, respectivamente.

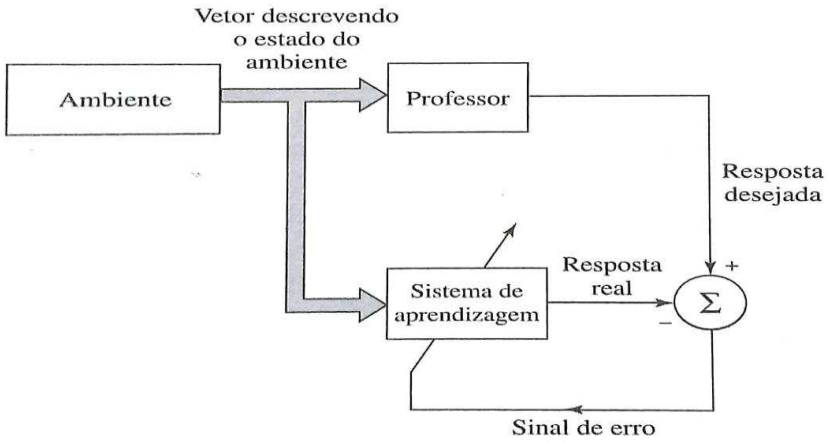
2.1.4.1 APRENDIZAGEM SUPERVISIONADA

O aprendizado supervisionado necessita de um agente denominado professor. O professor detém o conhecimento sobre o ambiente externo e apresenta a rede este conhecimento como um conjunto de amostras de entrada e sua respectiva saída desejada (BARRETO, 2002).

Através das amostras apresentadas a rede, os pesos sinápticos e limiares são ajustados continuamente, de forma iterativa, através de comparações feita pelo próprio algoritmo de aprendizagem, entre a resposta do sistema e a resposta desejada. Tal diferença é utilizada no processo de ajuste dos pesos até que a resposta obtida esteja dentro dos valores aceitáveis para tal solução (SILVA; SPATTI; FLAUZINO, 2010).

A Figura 8 representa um exemplo típico de aprendizado supervisionado, que é o aprendizado por correção de erros. Onde o objetivo é minimizar o erro da saída, aproximando assim da resposta desejada pelo professor (HAYKIN, 2001).

Figura 8 - Diagrama da aprendizagem com um professor.



Fonte: Haykin (2001, p.88).

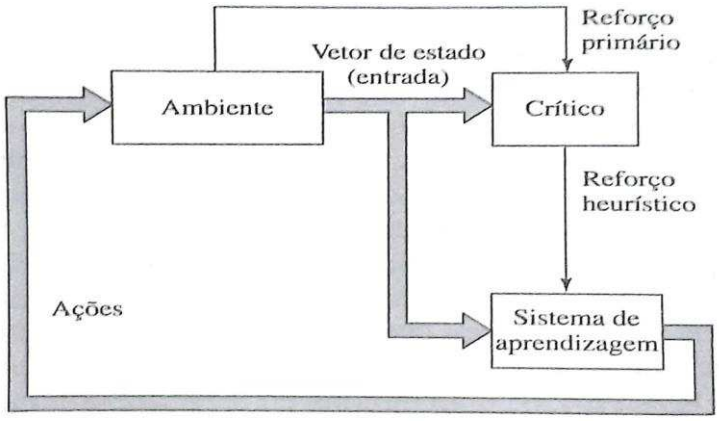
2.1.4.2 APRENDIZAGEM NÃO-SUPERVISIONADA

No paradigma de aprendizagem não-supervisionada, como o próprio nome já diz, não há um professor para supervisionar o processo de aprendizagem (HAYKIN, 2001). Neste processo, somente os padrões de entrada

são apresentados a rede, de forma contínua, e o aprendizado é obtido através da semelhança entre os dados. Sendo estas semelhanças e regularidades nas entradas, características essenciais para o aprendizado não-supervisionado (BRAGA, 2011).

A Figura 9 mostra o diagrama em blocos de um exemplo típico de aprendizagem não-supervisionada, chamado de aprendizagem por reforço, onde um mapeamento da entrada e saída é feito de forma iterativa, contínua com o ambiente, com o objetivo de minimizar um índice escalar de desempenho. O sistema converte um sinal de reforço primário, recebido pelo ambiente em um sinal de reforço de melhor qualidade, chamado de reforço heurístico (HAYKIN, 2001).

Figura 9 - Diagrama da aprendizagem sem um professor.



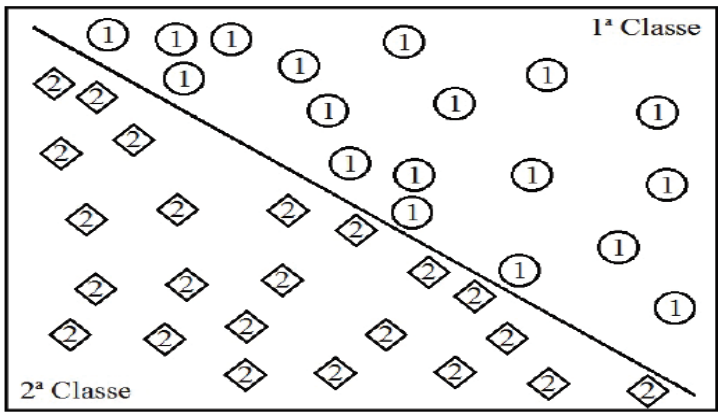
Fonte: Haykin (2001, p.90).

2.1.5 REDE *PERCEPTRON*

O *Perceptron* é a rede mais simples usada para classificação de padrões linearmente separáveis. Consiste em um único neurônio com pesos sinápticos e função de ativação ajustáveis. O algoritmo de treinamento foi desenvolvido por Rosenblatt, que provou que se os dados utilizados para treinar a rede são retirados de classes linearmente separáveis, então o algoritmo converge (HAYKIN, 2001).

A grande importância de estudar especificamente esta rede é que a mesma serve como base para às diversas outras redes existentes. Por ser uma rede simples, sua resposta de processamento tende a ser rápida, porém é limitada à problemas linearmente separáveis, como mostrado na Figura 10 (SOUSA; TORRES, 2011). O *Perceptron* constituído de apenas um neurônio é limitado a realizar classificação de padrões com apenas duas classes. Para resolver problemas de classificação com mais de duas classes, expande-se a camada de saída do *Perceptron* para mais de um neurônio (HAYKIN, 2001).

Figura 10 - Exemplo de um conjunto linearmente separável.



Fonte: Sousa; Torres (2011, p. 53).

O treinamento da rede é feito apresentando em sua entrada o conjunto de exemplos distintos de duas classes. Ao se adquirir o conjunto de treinamento formado pelos grupos de entrada e saída apresentados, utiliza-se o algoritmo proposto por Rosemblant, que tende a ajustar os pesos sinápticos para classificação das duas classes (SOUSA; TORRES, 2011).

O ajuste dos pesos é feito através da observação da saída da rede comparando-a com a saída desejada. Se o valor da saída for coincidente com o valor desejado, os pesos são incrementados. Caso não coincida, eles serão decrementados. Tal processo é executado, repetitivamente, até que a saída seja similar a saída desejada de cada amostra. Em termos matemáticos, as regras de ajuste dos pesos sinápticos e dos limiares podem ser expressas pelas seguintes expressões:

$$w_i^{atual} = w_i^{anterior} + n.\left(d^{(k)} - y\right).x^{(k)} \quad (2.3)$$

$$\theta_i^{atual} = \theta_i^{anterior} + n.\left(d^{(k)} - y\right).x^{(k)} \quad (2.4)$$

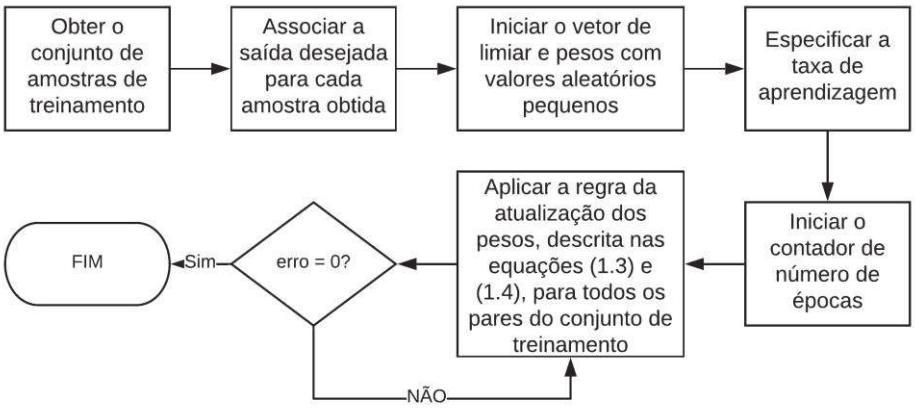
Onde w é o vetor que contêm o limiar e os pesos, $x^{(k)}$ é a k-ésima amostra de treinamento, $d^{(k)}$ é o valor desejado para a k-ésima amostra de treinamento, y é o valor da saída produzida pela rede, e n é uma constante que define a taxa de aprendizagem da rede escolhida, geralmente no intervalo $0 < n < 1$ (SILVA; SPATTI; FLAUZINO, 2010).

Os exemplos apresentados à rede devem ser escolhidos de forma a apresentar as várias formas que cada classe possa assumir, para que a rede seja capaz de generalizar o problema e classificar um exemplo que não tenha sido apresentado no conjunto de treinamento (SOUSA; TORRES, 2011).

Geralmente, nas redes *Perceptron*, se utiliza a função degrau como função de ativação, assim como mostrada na Figura 3, já apresentada. A função tem o seguinte comportamento, se a soma dos valores ponderados de entrada gerar valores maiores que zero, o neurônio envia positivo à saída, e se a soma gerar valores menores ou iguais a zero, é enviado negativo à saída (SOUSA; TORRES, 2011).

Segundo Silva, Spatti e Flauzino (2010), o passo a passo do treinamento do *Perceptron* pode ser representado como mostra a Figura 11.

Figura 11 - Fluxograma do algoritmo de treinamento.



Fonte: Autoria Própria (2019).

2.1.6 REDE *PERCEPTRON* MULTICAMADAS

As redes *Perceptron* Multicamadas, também chamadas pelo seu nome em inglês *Multi Layer Perceptron* (MLP), são redes compostas pela presença de pelo menos uma camada intermediária de neurônios, ou seja, no total esta arquitetura deve ter pelo menos duas camadas (HAYKIN, 2001). Portanto, trata-se de uma rede igual a já apresentada na Figura 7.

As redes MLP têm sido utilizada com sucesso em diversas aplicações que envolvem dados não linearmente separáveis, graças ao seu algoritmo de treinamento supervisionado chamado de *error back-propagation* (SILVA; SPATTI; FLAUZINO, 2010). O modelo de cada neurônio da rede possui uma função de ativação não linear, porém essa não linearidade tem de ser suave. Uma forma bastante utilizada são as funções sigmoide e tangente hiperbólica, já apresentadas (HAYKIN, 2001).

O *back-propagation*, ou retropropagação, é um algoritmo baseado na regra de aprendizagem por correção do erro. Esta aprendizagem basicamente consiste em dois passos através das camadas da rede. Um passo é dado para frente, a propagação, onde as entradas são aplicadas à rede e produzem um sinal de erro na saída, e outro passo é dado para trás, retropropagação, onde os pesos são ajustados com intuito do valor real se aproximar ao máximo do valor desejado (HAYKIN, 2011).

2.1.7 RECONHECIMENTO E CLASSIFICAÇÃO DE PADRÕES

Os seres humanos possuem extrema facilidade em reconhecimento de padrões. Como por exemplo, identificar um membro da família apenas ao ouvir sua voz por telefone, mesmo que a ligação esteja ruim (HAYKIN, 2001).

Para uma rede neural reconhecer padrões, primeiramente é passado pelo processo de treinamento. Neste processo é apresentado a rede, de forma repetitiva, um conjunto de padrões de entrada, juntamente com a classe que cada padrão pertence. Assim, a rede deverá adaptar seus pesos de modo a mapear os dados de entrada com as classes correspondentes. Após o

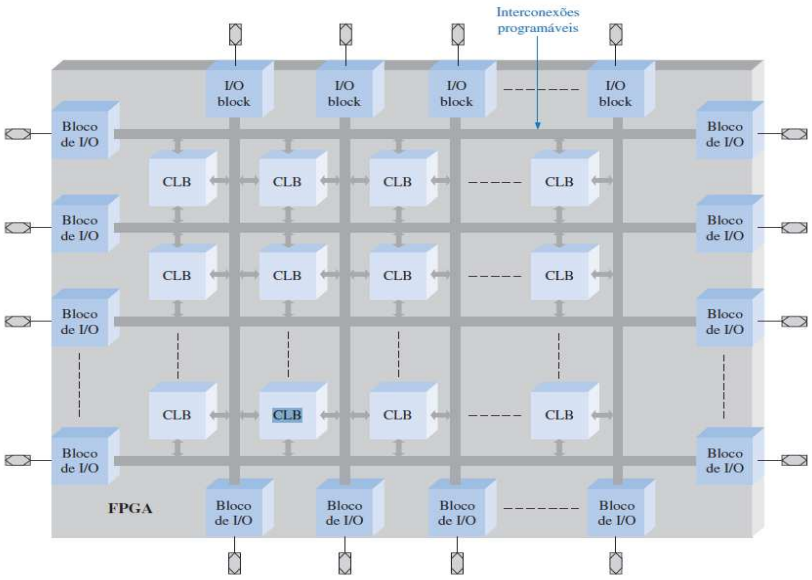
treinamento, é apresentado a rede outros conjuntos de padrões pertencentes as mesmas classes dos apresentados no treinamento, porém com características diferentes (BRAGA, 2011).

O reconhecimento de padrões é de natureza estatística, sendo os padrões representados por pontos em um espaço de decisão multidimensional. É no processo de treinamento que são delimitadas as fronteiras de decisão (HAYKIN, 2001).

2.2 FPGA

Em meados da década de 1980, a empresa Xilinx lançou uma nova arquitetura de circuitos integrados chamada de FPGA. Tal arquitetura é baseada em lógica programável, contendo uma matriz bidirecional de blocos lógicos programáveis chamados de CLB (*Configuration Logical Blocks),* blocos de entrada e saída, e interconexões reconfiguráveis (FLOYD 2007). A Figura 12 ilustra a estrutura básica de um FPGA.

Figura 12 - Estrutura básica de um FPGA.

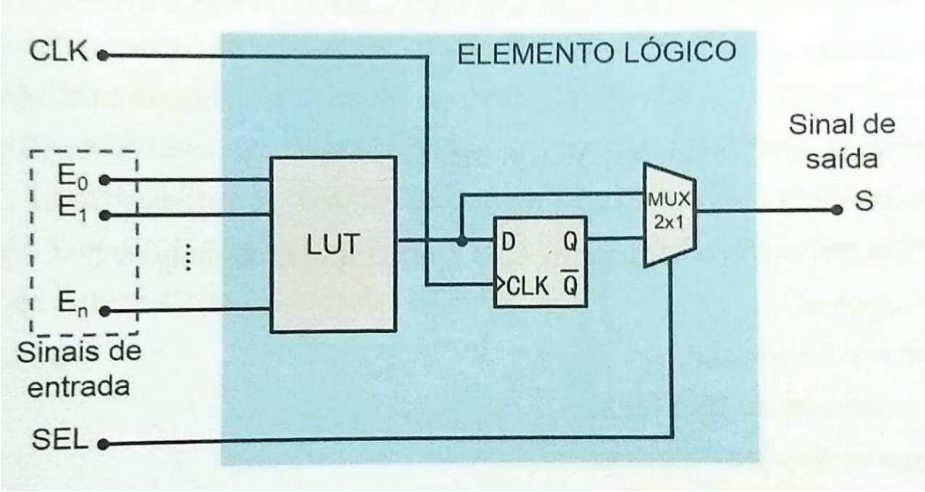


Fonte: Floyd (2007, p. 645).

Os blocos CLB são constituídos pela união de *flip-flops* e a utilização de lógica combinacional, assim como representado na Figura 13. Além dos CLBs, outros blocos de *hardware* com configurações específicas podem ser

incorporados ao FPGA, tais como memórias, multiplicadores, DSPs e até microcontroladores (DANTAS, 2014).

Figura 13 - Estrutura interna do CLB.



Fonte: Dantas (2014, p.457).

Os FPGAs permitem o projeto de circuitos paralelos, ou seja, uma parte de sua estrutura interna pode executar operações de forma independente de outras partes da mesma estrutura (SOUSA; TORRES, 2011).

Segundo Samame (2015), as principais vantagens dos FPGAs são:

- Velocidade de processamento devido as implementações baseadas em fluxo de dados ao invés de fluxo de instruções, e executadas diretamente em hardware;
- Flexibilidade em *hardware* devido ao alto poder de ajustes a funções específicas, mediante reconfiguração estática ou dinâmica;
- Baixo custo;
- Rápido desenvolvimento de protótipos.

2.3 VHDL

O VHDL (*VHSIC Hardware Description Language)* é uma linguagem de descrição de *hardware* que não depende de tecnologias e de fabricantes. Seu código descreve a parte comportamental ou estrutural que se deseja

implementar. A partir disso, um circuito físico, correspondente com tal descrição, é inferido pelo compilador (PEDRONI, 2010).

Esta linguagem foi criada pelo Departamento de Defesa dos Estados Unidos, em meados da década de 1980, com o objetivo de se ter uma linguagem compacta de descrever projetos de circuitos integrados de altíssima velocidade (*VHSIC).* O sucesso do VHDL se deve a padronização pelo IEEE, identificada pelo padrão IEEE 1076-2008, sendo esta sua última revisão (DANTAS, 2014).

A estrutura de um código em VHDL consiste em três partes básicas. As declarações das bibliotecas e pacotes, entidade e arquitetura. Na primeira parte deve ser declarada uma lista com todas as bibliotecas e pacotes necessários para o compilador processar o projeto.

Na entidade se encontra duas seções de código denominadas *PORT* e *GENERIC,* sendo o *PORT* uma lista com especificações das portas do circuito, e *GENERIC* a declaração das constantes genéricas, sendo esta opcional. A arquitetura, última parte da estrutura, contém o código propriamente dito, iniciado por um comando *BEGIN* e finalizado por um comando *END* (PEDRONI, 2010).

A linguagem permite a utilização de códigos sequenciais. Para isso, os comandos sequenciais não são empregados na região de código concorrente. As regiões específicas para esse código são chamadas de subprogramas e processos (D'AMORE, 2012).

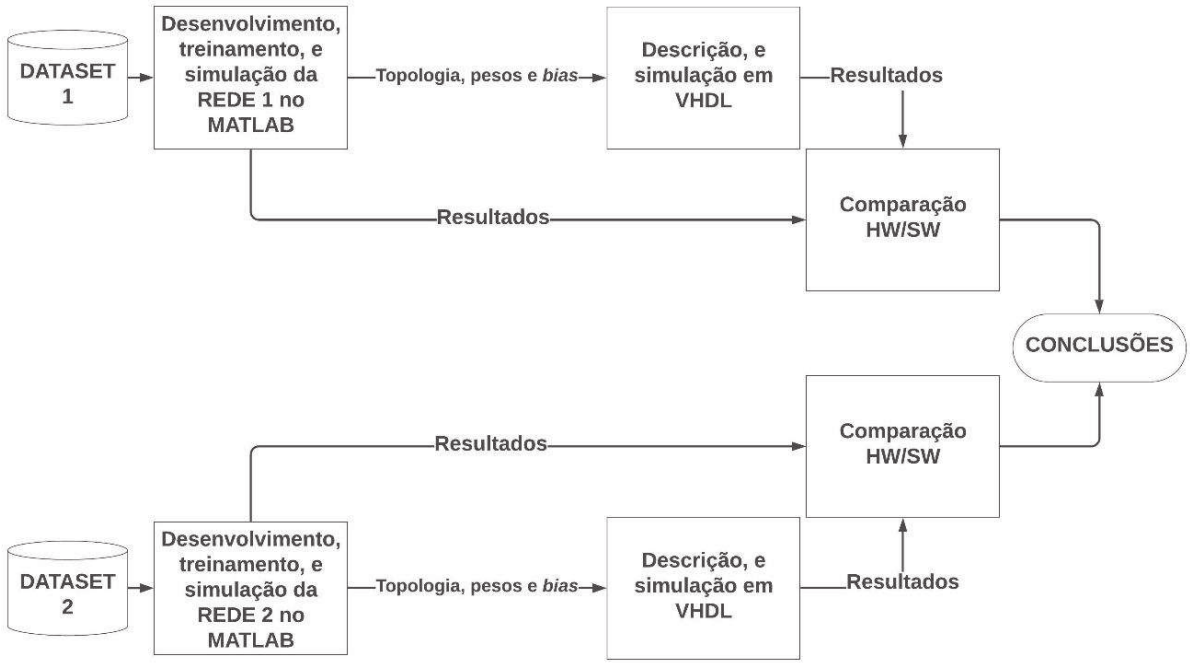
O VHDL suporta projetos com múltiplas hierarquias, ou seja, o projeto pode ser dividido em módulos separados que serão interligados para compor o sistema completo. Dividindo assim a complexidade do problema em problemas menores (ALBERTIN, 2016 apud PEDRONI, 2010).

3 DESENVOLVIMENTO

Neste capítulo são abordados todos os passos para o desenvolvimento do projeto, como *hardware*, *software* e *datasets* utilizados, implementação das redes e treinamento das mesmas no MATLAB® e implementação em VHDL. Todas as etapas de desenvolvimento serão divididas para a primeira rede neural e para a segunda rede neural.

Para melhor compreensão do desenvolvimento, a Figura 14 mostra o diagrama de blocos contendo a ideia geral de como foi desenvolvido este trabalho.

Figura 14 - Diagrama de blocos do desenvolvimento.



Fonte: Autoria Própria (2019).

3.1 DESCRIÇÃO DAS FERRAMENTAS DE *SOFTWARE* E *HARDWARE*

Para a utilização de ferramentas de *software* foi utilizado um notebook com processador Intel Core i7-7500U 2.9 GHz, 8 GB de memória RAM, rodando um sistema operacional Windows 10.

O desenvolvimento, treinamento e simulações das RNAs em *software* foram feitos no MATLAB®, onde adquiriu-se os parâmetros necessários para implementação em VHDL. O *Quartus Prime* foi utilizado para descrever e simular os circuitos de *hardware* das redes em VHDL.

3.2 *DATASETS* UTILIZADOS

Os *datasets* são bancos de dados que contém os conjuntos de entradas e suas respectivas saídas utilizadas para o treinamento da rede. A seguir são apresentados os dois conjuntos de dados utilizados neste trabalho.

3.2.1 *DATASET 1*

Para o treinamento da primeira RNA, foi utilizado um *dataset* muito comum, consideravelmente simples e com poucos dados, que representa os caracteres das vogais como uma matriz binária de tamanho 5x4, para tentar simular a construção gráfica de letras *pixel* a *pixel*, onde bit 0 seria *pixel* branco, e bit 1 o *pixel* preto, assim como mostra a Figura 15.

Figura 15 - Representação das Letras em forma matricial.

Letra A			
0	1	1	0
1	0	0	1
1	1	1	1
1	0	0	1
1	0	0	1

Letra E			
1	1	1	1
1	0	0	0
1	1	1	0
1	0	0	0
1	1	1	1

Letra I			
0	1	0	0
0	1	0	0
0	1	0	0
0	1	0	0
0	1	0	0

Letra O			
0	1	1	0
1	0	0	1
1	0	0	1
1	0	0	1
0	1	1	0

Letra U			
1	0	0	1
1	0	0	1
1	0	0	1
1	0	0	1
0	1	1	0

Fonte: Autoria Própria (2019).

Com o intuito de adaptar estes dados para serem utilizados como entrada da rede, cada letra será representada por um vetor de uma única coluna e 20 linhas sendo os pixels em forma binária, assim como mostra o Quadro 1 disponível no Apêndice A do trabalho.

O conjunto de dados de saída desse sistema também são vetores binários de 5 bits que representam qual é a vogal a partir da posição do bit em nível alto, assim como mostra a Tabela 1.

Tabela 1 - Dados de saída para cada vogal.

A	E	I	O	U
1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

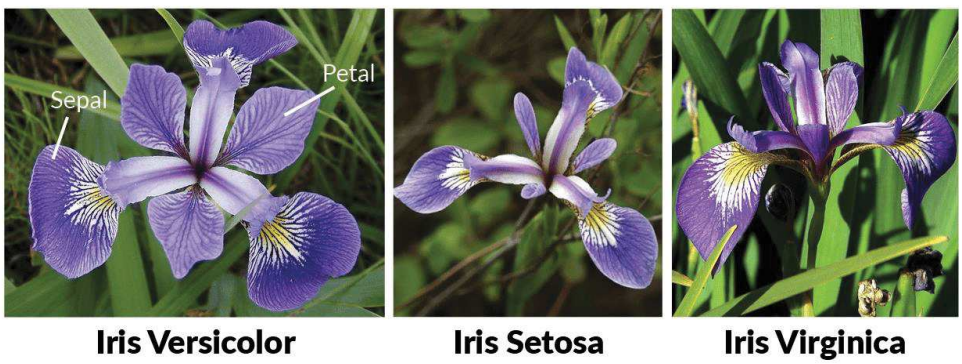
Fonte: Autoria Própria (2019).

3.2.2 *DATASET* 2

Na segunda rede apresentada neste trabalho, foi utilizado outro dataset muito usado para reconhecimento de padrões, porém, um pouco mais complexo que o anterior, pois seus dados originais não são números inteiros e seu conjunto não é linearmente separável.

Este conjunto de dados contém três classes que representam diferentes espécies da planta Íris: Íris setosa, Íris Versicolor, Íris virginica. Nesse conjunto há quatro atributos de entradas, sendo eles: comprimento da sépala, largura da sépala, comprimento da pétala, e largura da pétala. A Figura 16 mostra as três espécies da flor.

Figura 16 - Representação das três espécies citadas.



Fonte: Willems (2018).

Para cada classe há cinquenta diferentes conjuntos de atributos para serem apresentados à rede na etapa de treinamento.

Os dados de saída são representados por vetores binários de três bits, onde cada combinação representa uma classe, assim como mostra a Tabela 2.

Tabela 2 - Dados de saída para cada classe do segundo banco de dados.

Classe 1	Classe 2	Classe 3
1	0	0
0	1	0
0	0	1

Fonte: Autoria Própria (2019).

3.3 IMPLEMENTAÇÃO NO MATLAB®

A criação e treinamento da rede neural foi feita no MATLAB®. Para isso, utilizou-se uma ferramenta chamada *Neural Network Toolbox*, pois ela fornece algoritmos, funções e aplicativos com intuito de criar, treinar, visualizar e simular as redes neurais, auxiliando o desenvolvimento da mesma. Como serão implementadas duas redes diferentes, a primeira será chamada de Rede 1, e a segunda de Rede 2.

3.3.1 REDE 1

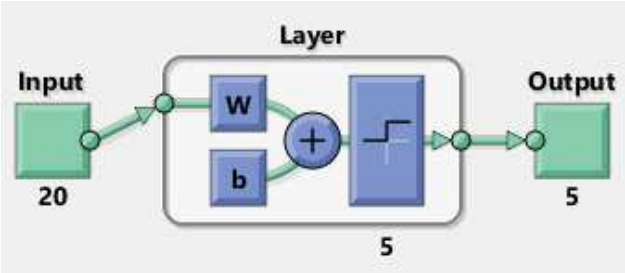
Para a criação desta rede foi utilizado o banco de dados das letras, descrito anteriormente. Por ser uma rede simples com uma fácil convergência, utilizou-se os parâmetros de treinamento indicados pelo próprio MATLAB®, sem ser necessário muitas alterações.

Por ser uma rede mais simples, utilizou-se o *nntool*, que é um aplicativo pertencente ao *toolbox* de redes neurais. A arquitetura utilizada foi a *Perceptron*, ou seja, possui apenas uma camada de neurônios.

A função de ativação escolhida foi o degrau unitário, pois o vetor de saída desejado é binário, e esta função limita o resultado em apenas 0 ou 1, e por ser facilmente implementada em *hardware*. O algoritmo utilizado para a atualização dos pesos e *bias* foi o *Cyclical order weight/ bias training*, pois é

selecionado automaticamente pelo MATLAB® neste caso. A quantidade de neurônios da rede foi definida pelo número de *bits* do vetor de saída, ou seja, 5 neurônios. Portanto, a rede terá 5 neurônios processando os dados, com 20 entradas em cada neurônio, assim como mostra a Figura 17.

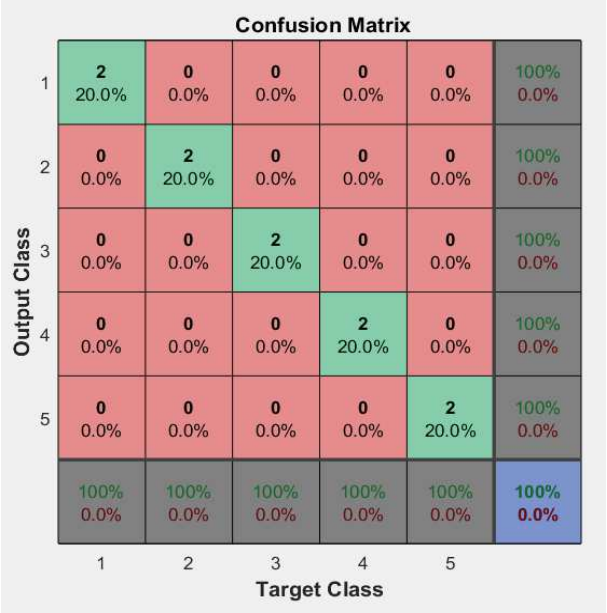
Figura 17 - Arquitetura da Rede Neural.



Fonte: Autoria Própria (2019).

Após o treinamento da rede criou-se um conjunto de testes que não foi apresentado para a rede na etapa de treinamento, com intuito de analisar sua taxa de acerto. Para cada letra, dois vetores com ruído foram criados, onde mudou-se um *pixel* de cada imagem. O Quadro 2 disponível no Apêndice A mostra o conjunto de vetores ruidosos utilizado para o teste. A partir da simulação do comportamento da rede gerou-se a matriz de confusão, apresentada na Figura 18, que nos mostra o quanto a rede foi assertiva em sua classificação, à partir das imagens ruidosas.

Figura 18 - Matriz de confusão da primeira RNA.



Fonte: Autoria Própria (2019).

Esta matriz nos mostra a relação de acerto na classificação, ou seja, se a classe desejada na saída foi igual a classe real. Quando há algum erro na classificação, ela nos apresenta com qual classe a saída foi confundida. Na imagem anterior pode-se observar que todas as saídas reais foram iguais as saídas desejadas, ou seja, todos os vetores criados para teste foram classificados de forma correta.

Os parâmetros de pesos e *bias* provenientes do treinamento foram anotados para posterior utilização no circuito em VHDL.

3.3.2 REDE 2

A segunda RNA foi criada e treinada a partir do segundo banco de dados também descrito anteriormente.

Para o desenvolvimento desta rede foi elaborado um *script* no MATLAB® utilizando as funções do *Toolbox* de redes neurais, pois por se tratar de um banco de dados mais complexo em relação ao anterior, alguns parâmetros de treinamento foram definidos visando uma convergência mais rápida e o menor erro possível.

Como se trata de um problema em que os dados não são linearmente separáveis, utilizou-se uma arquitetura chamada *Multi Layer Perceptron* (MLP), portanto, há mais de uma camada de neurônios.

A função de ativação utilizada foi a tangente hiperbólica, pois foi a que apresentou melhor desempenho no treinamento em relação às outras funções e também porque há vários estudos abordando sua implementação em *hardware*. Para atualização dos pesos e *bias*, o algoritmo *Levenberg-Marquardt backpropagation* foi utilizado, pois apresentou convergência mais rápida e melhores resultados em relação ao *Scaled Conjugate Gradiente.* Para executar o treinamento, o banco de dados foi dividido de forma aleatória em três partes: 60% dos dados para treinamento, 20% para validação, 20% para teste.

Para a escolha da quantidade de neurônios na camada oculta, avaliou-se a taxa de assertividade, apresentada no gráfico de confusão, para a rede treinada com 2, 4, 8, 20 neurônios. A Tabela 3 mostra a quantidade de acerto em porcentagem para cada classe.

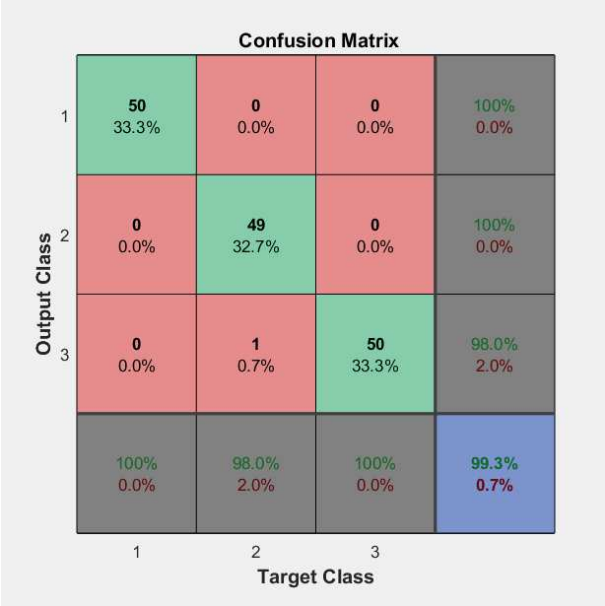
Tabela 3 - Porcentagem de acerto das redes em relação ao número de neurônios na camada oculta.

	2 neurônios	**4 neurônios**	**8 neurônios**	**20 neurônios**
Classe 1	0%	100%	100%	100%
Classe 2	94%	98%	98%	98%
Classe 3	100%	100%	100%	98%

Fonte: Autoria Própria (2019).

A partir dessas informações, escolheu-se uma topologia com 4 neurônios na camada oculta, pois obteve a melhor taxa de acerto em relação ao custo computacional, pensando na implementação em *hardware*. A Figura 19 nos mostra o gráfico de confusão para topologia escolhida.

Figura 19 - Matriz de confusão para rede com 4 neurônios na camada oculta.

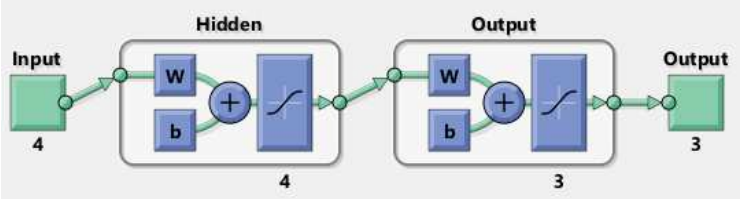


Fonte: Autoria Própria (2019).

Como pode-se observar na matriz de confusão da figura anterior, apenas uma amostra foi confundida. A saída desejada (eixo *Target* Class) para aquela amostra era a classe 2 e foi confundida com a classe 3 na saída real (eixo *Output Class*), por isso a taxa de 98% de acerto e 2% de erro, apresentada na imagem anterior, para esta classe. Todas as outras amostras utilizadas tiveram sua classificação correta.

Portando, a rede desenvolvida possui 7 neurônios para processamento dos dados, sendo 4 na camada oculta e 3 na camada de saída, assim como a Figura 20 mostra.

Figura 20 - Topologia utilizada para a segunda RNA desenvolvida.



Fonte: Autoria Própria (2019).

Após a validação da rede em *software* e coletados os parâmetros de pesos e *bias*, partiu-se para a etapa da implementação em *hardware*.

3.4 IMPLEMENTAÇÃO EM VHDL

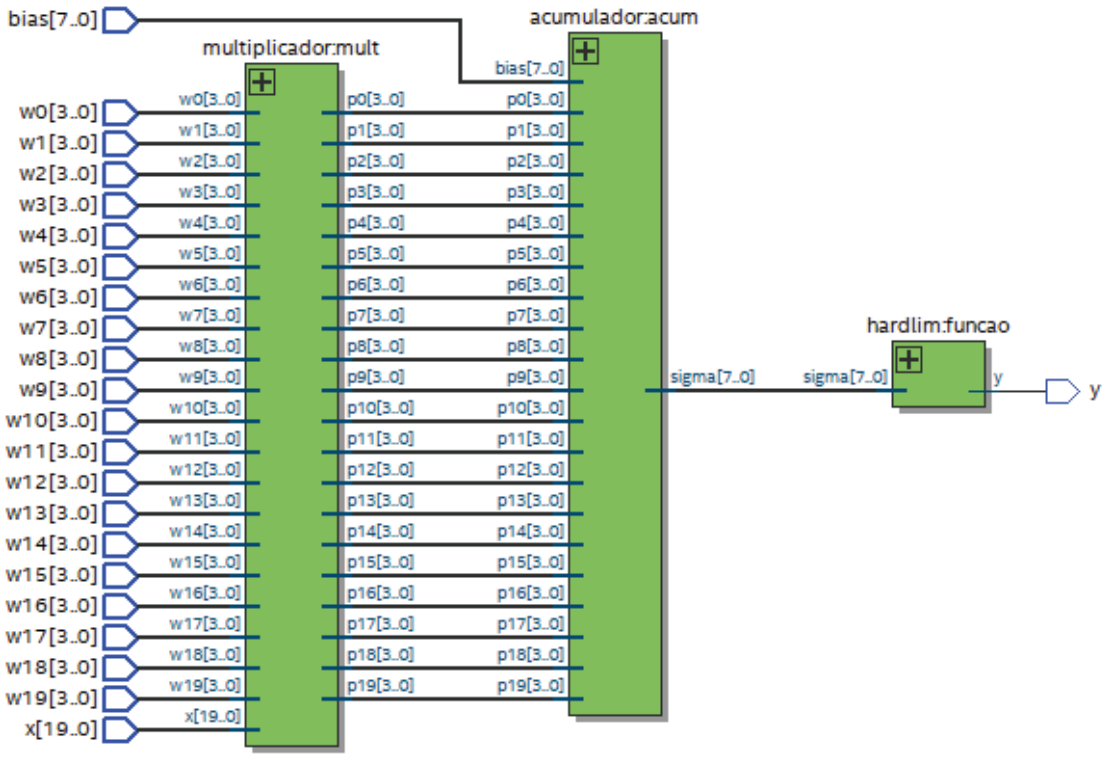
A partir do treinamento de ambas as redes no MATLAB®, gerou-se um *script*, disponível no Apêndice F e Apêndice G, para simular as redes em *software*, separadamente. Utilizando estes *scripts* como base pode-se descrever as redes em VHDL. A descrição dos *hardwares* foram feitas utilizando o conceito de projeto hierárquico, onde são feitos blocos separados para resolver problemas menores, e então une-se os blocos para o problema em sua totalidade.

3.4.1 IMPLEMENTAÇÃO EM VHDL – REDE 1

A arquitetura da primeira rede neural proposta é extremamente simples, onde é composta por 5 blocos de neurônio. Um vetor binário de 20 *bits* como entrada da rede representa cada pixel da imagem da vogal, chamado de *x.* Os pesos e *bias* foram declarados como constantes em um pacote denominado *neuron_types.* Os pesos são números de 4 bits e os *bias* são números de 8 *bits*, ambos do tipo *signal*, portando o *bit* mais significativo é o sinal e números negativos são escritos na forma de complemento de dois.

Cada bloco de neurônio é composto por 3 blocos: bloco multiplicador, bloco acumulador e bloco função de ativação, assim como mostra a Figura 21.

Figura 21 - Arquitetura do bloco neurônio.



Fonte: Autoria Própria (2019).

O bloco multiplicador recebe a entrada da rede e efetua a multiplicação de cada posição do vetor com seu peso correspondente. Como cada posição é um bit, sendo ele 0 ou 1, fez-se um comparador, onde compara se o bit de entrada for 1 a saída correspondente recebe o valor do peso, se for 0, recebe 0 em sua saída, tendo então o mesmo comportamento se fosse utilizado o multiplicador interno do FPGA para este caso.

O produto de cada *bit* de entrada com seu peso é a entrada do bloco acumulador. Nele é feito a soma de forma acumulativa de todos os produtos mais o *bias* correspondente daquele neurônio através do laço chamado de *for generate*. Sua saída é um vetor de 8 *bits* com sinal denominado *sigma*.

Sigma então é a entrada do bloco denominado *hardlim*, que é a função de ativação do sistema. Sua implementação é muito simples, pois como se trata de uma função identidade, bastou-se fazer um comparador onde, se a entrada do bloco for menor que 0, sua saída é 0, e se maior ou igual a 0 sua saída recebe 1.

E então, com a criação do bloco neurônio, bastou replicá-lo 5 vezes no bloco denominado *rede_neural* e através do *port map* fazer a ligação das entradas denominadas *x*, pesos sendo de *w0* a *w19*, *bias* sendo de *bias0* a *bias4* e as saídas dos neurônios foram armazenadas em um vetor de 5 bits chamado de *y.*

E assim se encerra a implementação da primeira rede em VHDL.

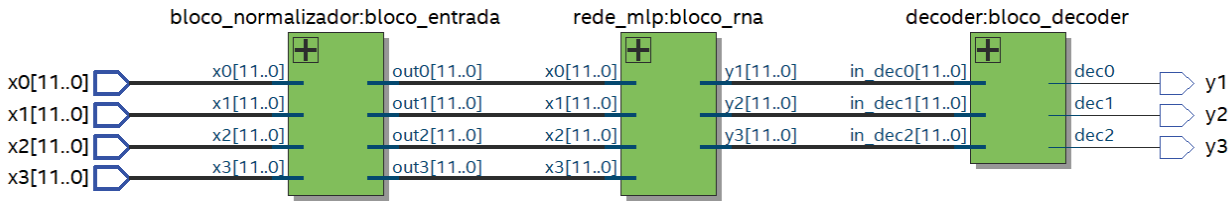
3.4.2 IMPLEMENTAÇÃO EM VHDL – REDE 2

A segunda rede é um pouco mais complexa de ser implementada, pois a arquitetura definida possui neurônios na camada oculta, e também seus dados de entrada não são números inteiros. Portanto, sua implementação será mais detalhada.

Os dados foram representados em ponto fixo com sinal, pois é menor o seu custo computacional e mais fácil sua implementação. Os dados de entrada e os pesos foram representados com 12 bits, sendo o bit de sinal o mais significativo, os três próximos a parte inteira e o restante a parte fracionária. Os dados dos *bias* foram representados com números de 24 bits, sendo o mais significativo o bit de sinal, os próximos 7 bits à parte inteira e o 16 bits para parte fracionária.

O algoritmo de treinamento escolhido na implementação no MATLAB® faz uma normalização dos dados antes e depois de serem expostos ao sistema da RNA em si. Portanto, o sistema como um todo possui 3 blocos: um bloco chamado *bloco_normalizador*, o da rede neural em si chamado *rede_mlp*, e um de decodificação chamado *decoder*. A Figura 22 mostra a topologia do sistema total.

Figura 22 - Blocos do sistema.

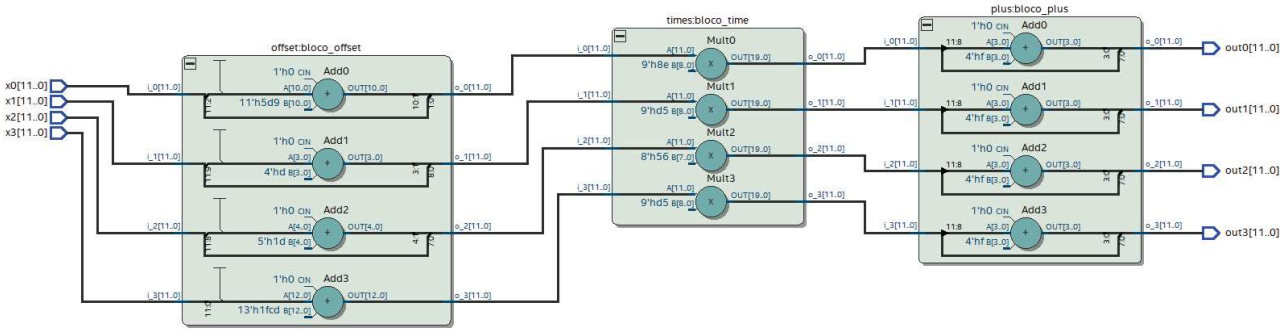


Fonte: Autoria Própria (2019).

3.4.2.1 MÓDULO DA ENTRADA

O *bloco_normalizador* é o bloco de entrada do sistema. Ele é composto por três outros blocos que fazem operações como subtração, multiplicação e soma, eles são: *offset*, *times* e *plus*, assim como mostra a Figura 23. Esta normalização é feita com intuito de diminuir a discrepância entre os dados na etapa de treinamento, otimizando a convergência.

Figura 23 - Blocos que compõem o bloco de entrada.



Fonte: Autoria Própria (2019).

O bloco chamado *offset* é composto por 4 somadores, que realizam a subtração de cada entrada com um valor predefinido. Depois os dados passam pelo bloco *times*, que fará a operação de multiplicação, e 4 multiplicadores constituem esse bloco. E posteriormente os dados serão somados pelo bloco *plus*, ou seja, este bloco é constituído por 4 somadores. Os parâmetros utilizados em cada bloco e para cada entrada estão apresentados na Tabela 4.

Tabela 4 - Parâmetros utilizados nas operações de normalização.

	Entrada 1	**Entrada 2**	**Entrada 3**	**Entrada 4**
Offset	4,3	2	1	0,1
Times	0,5555555	0,8333333	0,33898305	0,8333333
Plus	-1	-1	-1	-1

Fonte: Autoria Própria (2019).

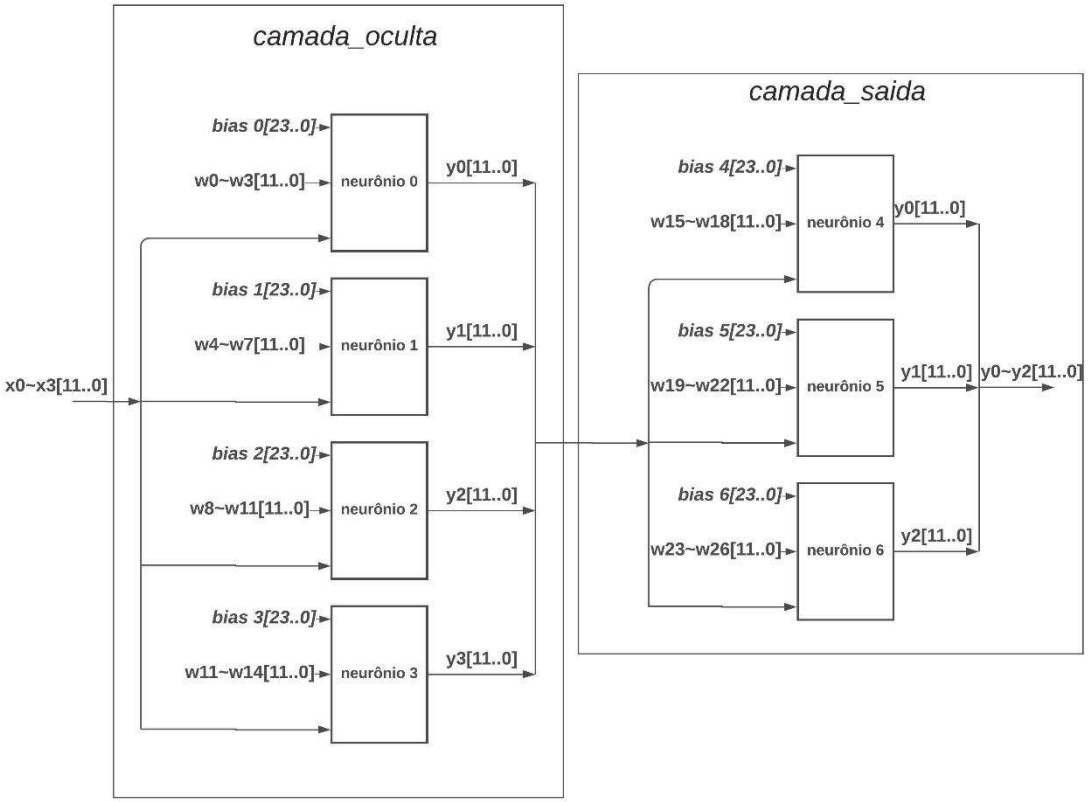
A escolha destes parâmetros é feita pelo MATLAB® na etapa de treinamento da rede. Por padrão é feita a normalização dos dados para que não haja problemas no treinamento em relação a convergência da rede.

O próximo bloco é o da RNA em si, portanto para melhor compreensão foi separado um item apenas para explicação dessa etapa.

3.4.2.2 IMPLEMENTAÇÃO DO BLOCO *REDE_MLP*

A arquitetura desse bloco possui 4 neurônios na camada oculta e 3 na camada de saída, onde estão localizados em módulos chamados de *camada_oculta* e *camada_saida*, assim como mostra a Figura 24.

Figura 24 - Arquitetura do bloco *rede_mlp.*



Fonte: Autoria Própria (2019).

Os figura com os blocos tirados do *Quartus Prime* está disponível no Apêndice E.

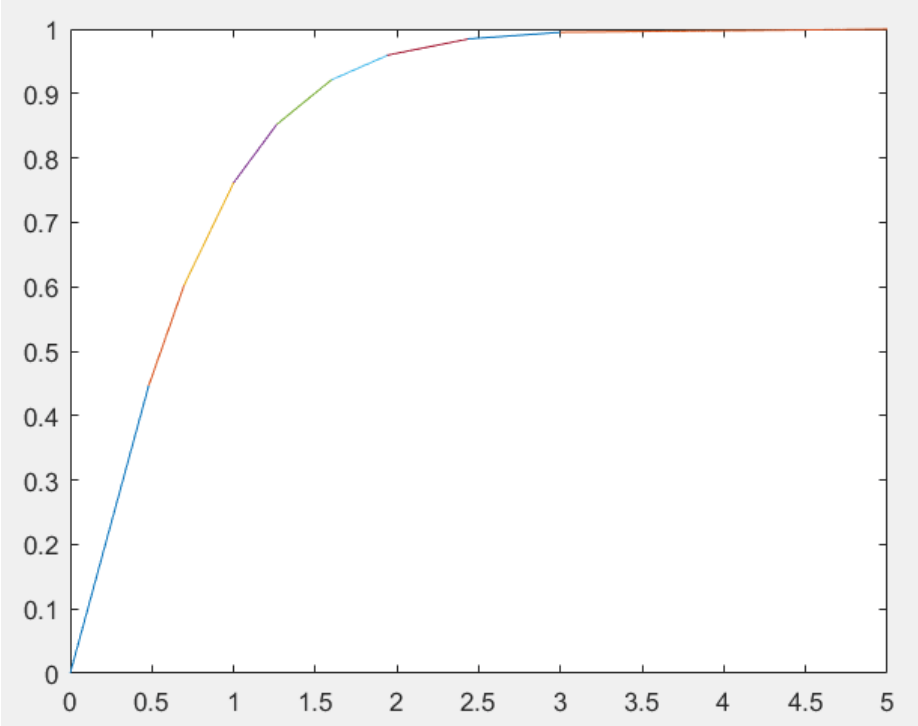
Todos os neurônios possuem a mesma arquitetura interna, onde possuem 2 módulos principais: *operações_lineares* e *tangente_hiperbólica*.

O bloco das operações lineares possui multiplicadores e um acumulador. Então, faz a multiplicação das entradas com seus respectivos pesos, similar ao da primeira rede, porém utiliza-se os multiplicadores internos do FPGA, fazendo com que saída dos multiplicadores tenham o dobro do tamanho da entrada. Isso justifica o fato do tamanho dos dados dos *bias* também possuírem o dobro dos dados dos pesos, para não haver problema na hora da soma no acumulador. E então sua saída é conectada a entrada do bloco da função de ativação.

A implementação da função tangente hiperbólica foi feita linearizando partes da função, levando em conta apenas o intervalo de -5 a +5. Como se trata de uma função ímpar, seu lado negativo é igual ao positivo porém com sinal invertido. Sabendo disso, bastou linearizar só o lado positivo da função, e quando

se tratar de números negativos na entrada inverte-se o sinal da saída. Utilizou-se 9 equações de primeiro grau para obter uma boa precisão. Portanto, a função linearizada é representada pela Figura 25.

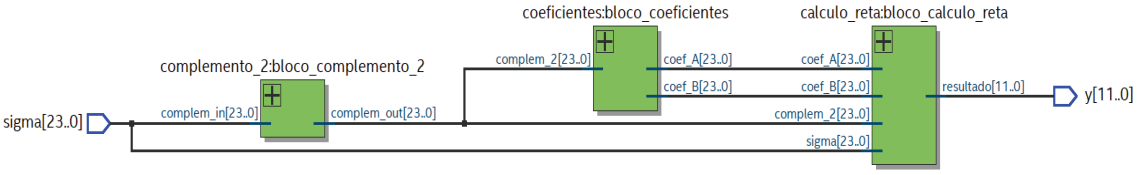
Figura 25 - Linearização da parte positiva da função tangente hiperbólica.



Fonte: Autoria Própria (2019).

Para isso, foram descritos 3 módulos em VHDL, apresentados na Figura 26, para aproximar a função tangente hiperbólica, sendo eles *complemento_2*, *coeficientes* e *calculo_reta.*

Figura 26 - Módulos da função tangente hiperbólica.



Fonte: Autoria Própria (2019).

O bloco *complemento_2*, verifica o bit mais significativo do vetor com o intuito de saber seu sinal. Então, se o número for negativo, faz-se o complemento

de dois do vetor multiplicando-o por -1, tendo o mesmo número com sinal positivo. Se o número já for positivo, apenas coloca-o na saída.

O próximo bloco é o *coeficientes*, que através de LUTs, utilizadas para tabelar os valores dos intervalos em que a entrada está, localiza a qual das retas o número da entrada pertence e retorna os coeficientes A e B para o cálculo da mesma.

Por fim, o bloco *calculo_reta* faz a multiplicação da entrada com o coeficiente A e depois a soma do resultado com o coeficiente B, tendo assim o resultado da função. Por último, este bloco compara se a entrada do bloco da função era negativa, se sim, faz o complemento de dois novamente para obter o resultado real da função.

Todos os parâmetros dos pesos e *bias* foram definidos como constantes em um pacote chamado *neuron_types*.

3.4.2.3 MÓDULO DE SAÍDA

O último módulo do sistema é o da saída, chamado de *decoder*. Nele apenas é feita a comparação de qual valor de saída entre os três neurônios eram maiores. Após a comparação é enviado um nível lógico alto para a saída correspondente ao maior valor, e as outras saídas recebem nível baixo. Identificando então a qual classe o padrão de entrada do sistema pertence.

4 RESULTADOS E DISCUSSÕES

Este capítulo apresenta os resultados do desenvolvimento das duas RNAs implementadas, tanto em *hardware* quanto em *software*. Todos os códigos e elementos utilizados para o desenvolvimento estão disponíveis para consulta em anexo a este trabalho.

4.1 CONSUMO DE RECURSOS

A partir da síntese lógica dos dois sistemas desenvolvidos neste trabalho, pode-se analisar o consumo de recurso de ambos. A Figura 27 mostra o consumo da primeira RNA. Seu consumo é muito pequeno pelo fato da rede ser extremamente simples.

Figura 27 - Compilation Report da Rede 1.

Flow Status	Successful - Thu Jul 11 19:56:23 2019
Quartus Prime Version	17.1.0 Build 590 10/25/2017 SJ Lite Edition
Revision Name	neural_network
Top-level Entity Name	rede_neural
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	156 / 114,480 (< 1 %)
Total registers	0
Total pins	28 / 529 (5 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Fonte: Autoria Própria (2019).

O total de elementos lógicos utilizados foi de 156, sendo menor que 1% do total neste modelo de FPGA. O total de pinos foi 28, sendo 5% do total de pinos disponíveis.

O consumo da segunda rede é apresentado na Figura 28. Por ser uma rede um pouco mais complexa, consumiu mais recursos do dispositivo.

Figura 28 - Compilation Report da Rede 2.

Flow Status	Successful - Thu Jul 11 20:04:32 2019
Quartus Prime Version	17.1.0 Build 590 10/25/2017 SJ Lite Edition
Revision Name	teste_01
Top-level Entity Name	teste_01
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	3,864 / 114,480 (3 %)
Total registers	0
Total pins	51 / 529 (10 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	28 / 532 (5 %)
Total PLLs	0 / 4 (0 %)

Fonte: Autoria Própria (2019).

O consumo de elementos lógicos foi muito maior que em relação a primeira rede. Utilizou-se 3864 elementos, ou seja, apenas 3% do total disponível. Os pinos utilizados representam apenas 10% do total. Nesse sistema, utilizou-se 28 multiplicadores internos do FPGA, representando 5% do total.

A quantidade de recursos lógicos utilizados pela implementação das funções de ativação dizem muito sobre a diferença entre o consumo total dos dois sistemas completos. As Figuras 29 e 30 mostram o consumo do bloco da função *hardlim* e *tanh*, respectivamente.

Figura 29 - Consumo de recursos lógicos da função degrau.

Total logic elements	0 / 114,480 (0 %)
Total registers	0
Total pins	9 / 529 (2 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Fonte: Autoria Própria (2019).

Figura 30 - Consumo de recursos função tangente hiperbólica.

Total logic elements	253 / 114,480 (< 1 %)
Total registers	0
Total pins	36 / 529 (7 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	4 / 532 (< 1 %)
Total PLLs	0 / 4 (0 %)

Fonte: Autoria Própria (2019).

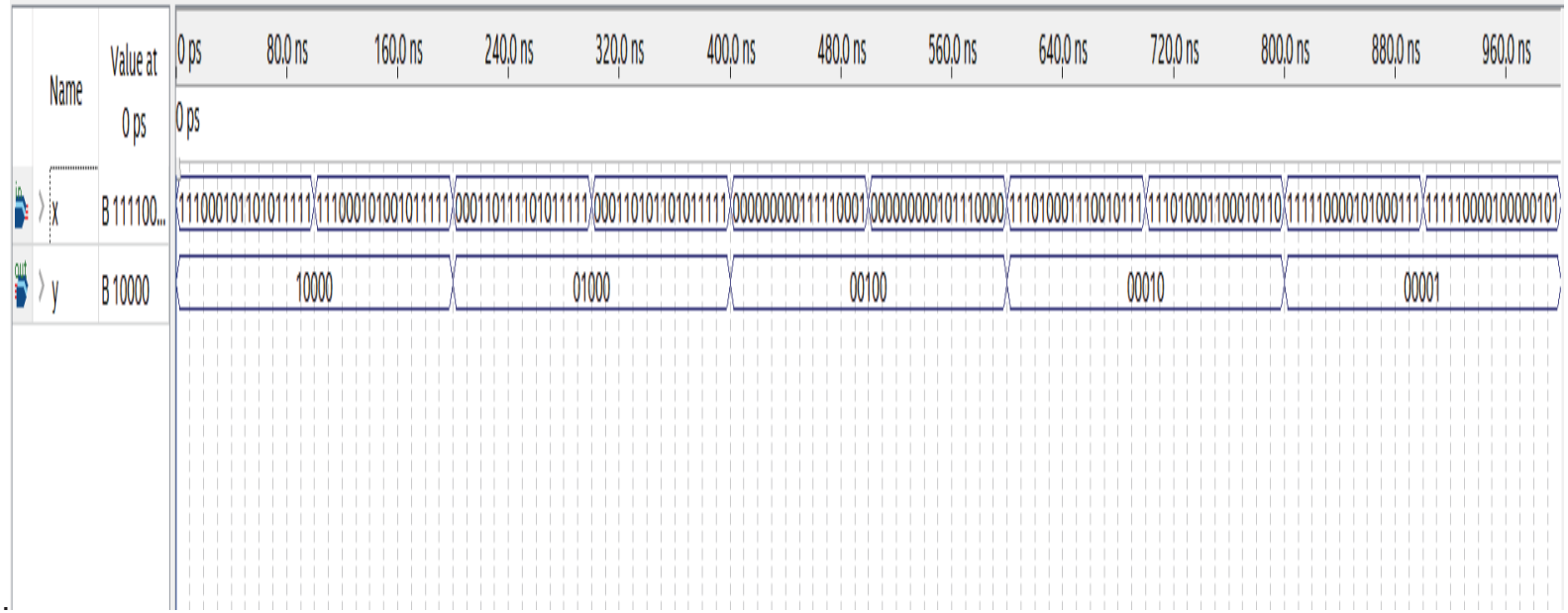
Na primeira função não temos consumo de elementos lógicos, pelo fato de haver apenas um comparador. Já na segunda função, presente em cada neurônio, há 253 elementos lógicos utilizados, e 4 multiplicadores internos pois a função foi aproximada por várias retas sendo necessário realizar operações de multiplicação para calcular seu valor.

4.2 COMPARAÇÕES HW x SW

Para saber se a rede se comportou de forma semelhante ao projeto feito no MATLAB®, foram feitos testes comparativos entre os resultados *hardware* e *software*.

A Figura 31 mostra a simulação da primeira RNA implementada em VHDL. Como entrada foram utilizados os mesmos vetores com ruídos criados para o teste da rede e citados no item 3.3.1. Relembrando, estes dados não foram utilizados no processo de treinamento, são apenas para teste.

Figura 31 - Simulação RNA 1.

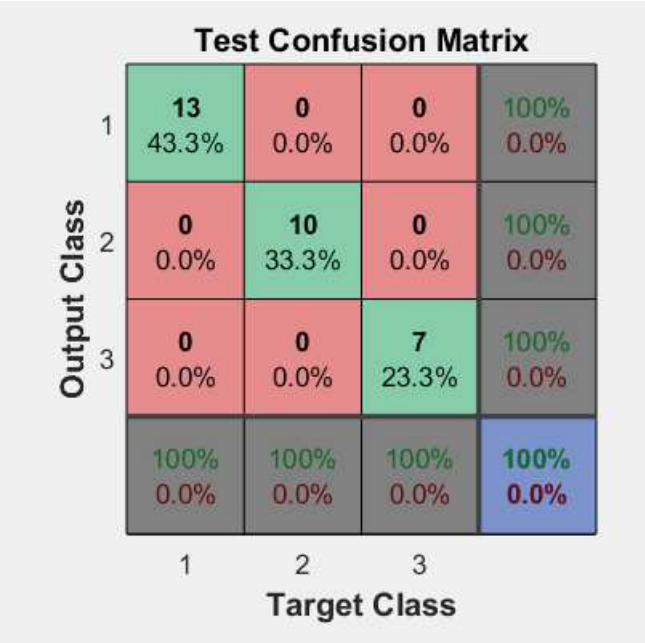


Fonte: Autoria Própria (2019).

A primeira RNA obteve 100% de acerto com a simulação do *hardware*, ou seja, a mesma taxa de acerto que a rede em *software*.

A segunda rede foi simulada utilizando os dados da fase de teste da etapa de treinamento do MATLAB®. São 30 conjuntos de amostras, sendo 13 da classe 1, 10 da classe 2 e 7 da classe 3. Em *software* obteve-se 100% de acerto utilizando estes dados, assim como mostra a matriz de confusão da Figura 32.

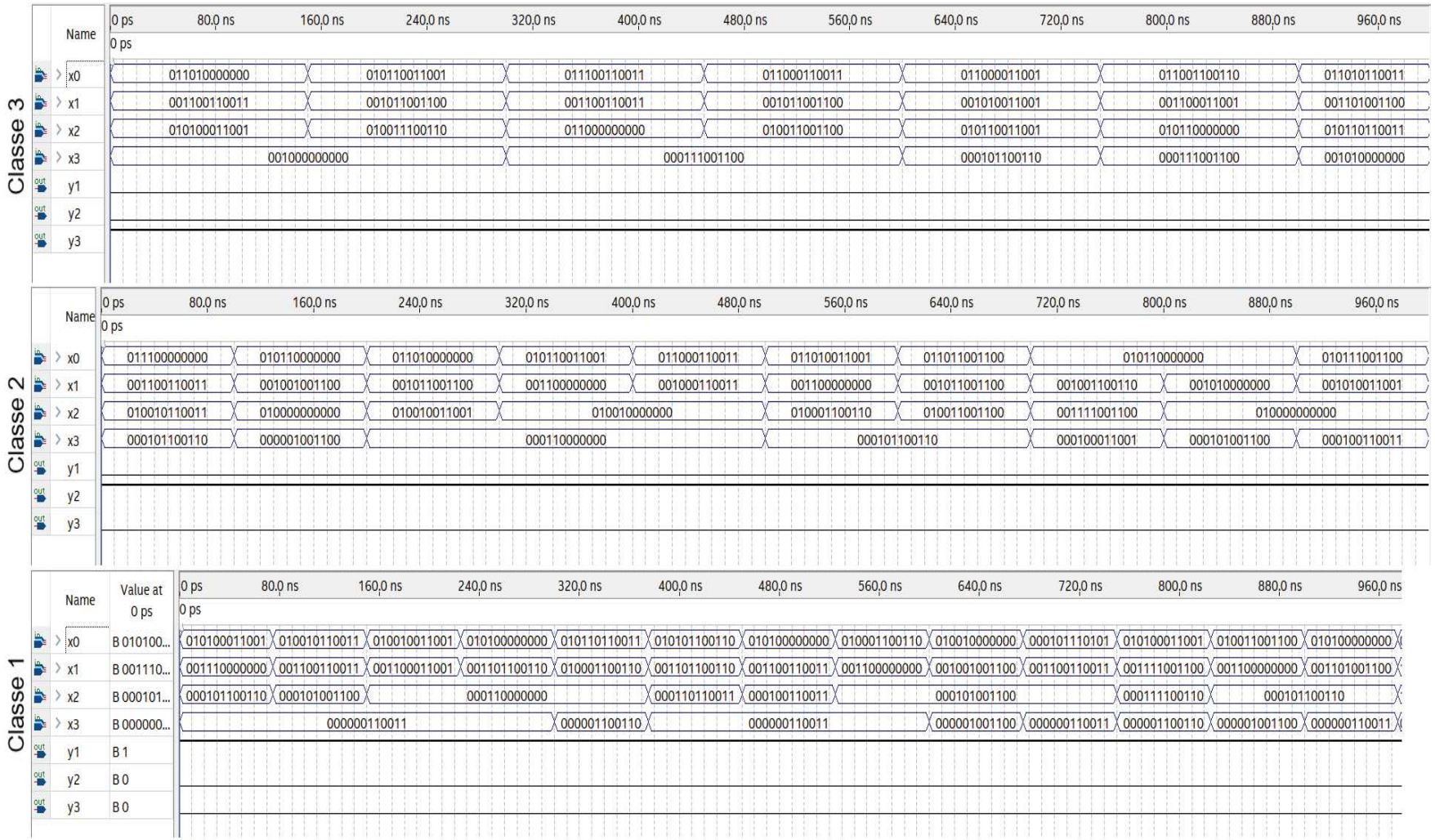
Figura 32 - Matriz de confusão utilizando os dados de teste.



Fonte: Autoria Própria (2019).

Estes dados não são apresentados para a rede na fase de atualização dos pesos e *bias*, portanto, utilizá-los torna maior a confiabilidade dos resultados. A Figura 33, mostra os resultados das simulações do circuito em VHDL. A partir da simulação tem-se que a taxa de acerto, utilizando os dados de teste, foram de 100%, igual a rede implementada em *software*.

Figura 33 - Simulação RNA 2.



Fonte: Autoria Própria (2019).

Outra comparação feita foi em relação ao tempo de execução das redes tanto em *hardware* quanto em *software*. Em *hardware* foi medido através do tempo de atraso da resposta do sistema, já em *software* foi através da função *tic toc* do MATLAB®*,* onde também mede o tempo de execução do programa até se obter a resposta. A Tabela 5 mostra os resultados tanto para a primeira rede quanto para a segunda.

Tabela 5 - Resultado do tempo de execução.

	Hardware	**Software**
Rede 1	15,83 ns	343 us
Rede 2	78,83 ns	1,76 ms

Fonte: Autoria Própria (2019).

5 CONCLUSÕES

Este trabalho tinha como objetivo desenvolver, implementar e simular uma arquitetura em *hardware* de redes neurais artificiais para reconhecimento e classificação de padrões. Portanto, este objetivo foi alcançado, com o desenvolvimento de duas topologias de RNA.

O comparativo entre a simulação em *hardware* e *software* foi bastante satisfatória, podendo mostrar o ótimo resultado apresentado pelos circuitos descritos. O custo computacional dos *hardwares* foi baixo em relação ao total disponível no FPGA escolhido. O tempo de execução dos circuitos descritos foram extremamente menores que as simulações em *software*, porém sua taxa de acerto foi a mesma, igual a 100%. Isso mostra que estes circuitos podem ser eficazes em problemas que necessitam de alta velocidade no reconhecimento de padrões, como em robôs móveis, por exemplo.

A diferença entre as duas redes em relação ao consumo de elementos lógicos se deve a implementação da função de ativação. Na primeira rede a função não fez uso de nenhum elemento, utilizando apenas um comparador. Na segunda foram consumidos 256 elementos.

O *software* MATLAB® foi de suma importância no desenvolvimento deste projeto, pois com a utilização das funções presentes no *Neural Network Toolbox*, pôde-se fazer o treinamento, validação e simulação das redes de forma mais rápida, sendo utilizado como base para a descrição em VHDL, evitando perda de tempo com possíveis erros nessas etapas.

O modelo de neurônio artificial descrito neste trabalho foi desenvolvido de forma bastante genérica, podendo ser utilizado em outras topologias, conforme a necessidade. Porém, a declaração dos pesos e *bias* não foram feitas da forma menos trabalhosa possível. Isso se deve a um erro no simulador, que impossibilitou a declaração dos pesos em arranjos de vetores, sendo necessário a declaração de forma individual como constantes, no pacote de dados.

Um ganho expresso com este trabalho foi a implementação da função de ativação tangente hiperbólica, pois apesar do erro devido a representação em ponto fixo e da aproximação com a linearização das partes da função, seu resultado foi satisfatório e sua arquitetura pôde ser utilizada no problema apresentado onde os dados não são linearmente separáveis. Podendo-se utilizar o bloco desta função para diversos problemas onde uma função não linear seja necessária.

Apesar do consumo de recursos ter sido baixo na implementação em *hardware*, a implementação de forma paralela em um único FPGA do mesmo

modelo utilizado não seria possível caso o número de entradas e, ou neurônios fosse alto. Sendo assim, a implementação de forma sequencial seria mais interessante. Como é o caso de um banco de dados denominado MNIST, que representa diversas imagens, de tamanho 28x28 pixels, de letras manuscritas. Se fosse implementado uma arquitetura para este caso, necessitaria de 784 entradas na rede, sendo necessário uma implementação de forma sequencial para os dados de entrada. A implementação deste trabalho teve inicio com este banco de dados, porém ao se deparar com a limitação do número de pinos do FPGA utilizado, e a necessidade de uma implementação de forma sequencial para este caso, migrou-se para outras bases de dados para que a implementação de forma totalmente paralela fosse possível.

Para trabalhos futuros algumas propostas são:

- Realizar a implementação de um algoritmo de treinamento e interface com usuário em *hardware*.
- Implementar a rede de forma genérica, podendo-se alterar a topologia da rede de forma mais fácil.
- Implementar um módulo para comunicação serial para o envio dos pesos e *bias* da rede treinada no MATLAB®.
- Explorar a implementação de outros tipos de redes, como a Hopfield.

REFERÊNCIAS

AGUIAR, F. G. **Utilização de Redes Neurais Artificiais para detecção de padrões de vazamento em dutos**. 2010. 95 f. Dissertação (Mestrado em Engenharia Mecânica) – Escola de engenharia de São Carlos da Universidade de São Paulo. São Paulo, 2010.

ALBERTIN, C. R. **Implementação de redes neurais artificiais utilizando FPGA para aplicações de controle inteligente em robótica**. Trabalho de Conclusão de Curso (Bacharelado em Engenharia Eletrônica) – Universidade Tecnológica Federal do Paraná – DAELN. Campo Mourão, 2016.

BARRETO, J. M. **Introdução às Redes Neurais Artificiais**. [2002]. Disponível em: <http://www.inf.ufsc.br/~j.barreto/tutoriais/Survey.pdf >. Acesso em: 27 mai. 2018.

D'AMORE, R. **VHDL**: descrição e síntese de circuitos digitais. 2. ed. Rio de Janeiro: LTC, 2012.

DANTAS, L. P. **Eletrônica digital**: técnicas digitais e dispositivos lógicos programáveis. São Paulo: SENAI-SP Editora, 2014.

FLOYD, T. L. **Sistemas digitais**: fundamentos e aplicações. 9. ed. Porto Alegre: Bookman, 2007.

HARIPRASATH, S; PRABAKAR, T. N. FPGA Implementation of Multilayer Feed Forward Neural Network Architecture Using VHDL. **International Conference on Computing and Applications**, Dindigul, p. 1-6, feb. 2012.

HAYKIN, S. **Redes Neurais**: princípios e prática. 2. ed. Porto Alegre: Bookman, 2001.

PEDRONI, V. A. **Eletrônica digital moderna com VHDL.** Rio de Janeiro: Elsevier, 2010.

PRADO, R. N. A. **Desenvolvimento de uma arquitetura em hardware prototipada em FPGA para aplicações genéricas utilizando redes neurais artificiais embarcadas**. 2011. 94 f. Dissertação (Mestrado em Ciências) – Programa de Pós Graduação em Engenharia Elétrica e de Computação – UFRN. Natal, 2011.

RUSSEL, S.; NORVIG P. **Inteligência Artificial**. 3. ed. Rio de janeiro: Elsevier, 2013.

SAMAME, L. F. C. **Arquitetura em hardware do filtro de Kalman estendido para localização de robôs móveis autônomos implementada em FPGA**. Dissertação (Mestrado em Sistemas Mecatrônicos) – Faculdade de tecnologia da Universidade de Brasília. Brasília, 2015.

SILVA, I. N.; SPATTI, D. H.; FLAUZINO, R. A. **Redes Neurais Artificiais para engenharia e ciências aplicadas:** curso prático. São Paulo: Artiliber, 2010.

SOUSA, M. A. A.; TORRES, T. F. J. Implementação direta de uma rede neural artificial em hardware e sua aplicabilidade no reconhecimento de padrões para seleção de frutas. **Sinergia**, São Paulo, v. 12, n. 1, p. 50-58, Jan. 2011.
THEODORIDIS, S.; KOUTROUMBAS, K.; **Pattern Recognition.** 2. ed. Elsivier, 2003.

WIDROW, B; LEHR, M. A. 30 years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation. **Proceedings of the IEEE**, v. 78, n. 9, p. 1-28, Sep. 1990.

WOLF, D. F. **Projeto de uma rede neural usada no reconhecimento de gestos por robôs móveis utilizando-se computação reconfigurável**. 2001. 106 f. Dissertação (Mestrado em Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e Computação - ICMC-USP. São Paulo, 2001.

APÊNDICE A – QUADROS DE DADOS

Quadro 1 – Representação dos dados de entrada em forma de vetores.

A	E	I	O	U
0	1	0	0	1
1	1	0	1	1
1	1	0	1	1
1	1	0	1	1
1	1	0	0	0
1	1	1	1	0
0	0	1	0	0
1	1	1	0	0
0	0	1	0	0
0	1	1	1	1
1	1	0	1	0
0	0	0	0	0
1	1	0	0	0
0	0	0	0	0
0	1	0	1	1
0	1	0	0	1
1	0	0	1	1
1	0	0	1	1
1	0	0	1	1
1	1	0	0	0

Fonte: Autoria própria (2019).

Quando 2 – Dados de entrada com ruídos utilizado para teste.

A	A	E	E	I	I	O	O	U	U
0	0	1	1	0	0	0	0	1	1
1	1	1	1	**1**	0	1	**0**	1	1
1	1	1	1	0	0	1	1	1	**0**
1	1	1	1	0	0	1	1	1	1
1	1	1	1	0	0	0	0	0	0
1	1	1	1	1	1	1	1	0	0
0	0	0	0	1	1	0	0	0	0
1	1	1	1	1	1	0	0	**1**	0
0	0	0	0	1	**0**	1	0	0	0
1	0	1	1	1	1	1	1	1	1
1	1	1	1	0	0	1	1	0	0
0	0	1	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0

0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	1	1	1
0	0	1	1	0	0	0	0	1	1
1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	1	1	1	1
1	**0**	1	**0**	0	0	0	0	0	0

Fonte: Autoria Própria (2019).

APÊNDICE B – SCRIPT PARA SIMULAÇÃO DA REDE 1

```
function [y1] = perceptron_alfabeto2(x1)

tic %% inicio função que conta o tempo de execução

%% Declaração dos pesos e bias
b1 = [0;
 -1;
 0;
 -1;
 0];

IW1_1 = [-1 0 0 0 1 0 0 1 0 -2 0 0 1 0 -2 -1 1 1 1 1;
 1 0 0 0 0 -1 -1 -1 -1 0 0 0 0 0 1 1 -1 -1 -1 0;
 0 -1 -1 -1 -1 0 1 0 1 1 -1 0 -1 0 0 0 -1 -1 -1 -1;
 -3 0 0 0 -2 2 -1 -3 -1 1 3 0 -2 0 2 -3 0 0 0 -2;
 2 0 0 0 -1 -2 0 -1 0 1 -2 0 -1 0 1 2 0 0 0 -1];

%% Simulação

Q = size(x1,2);

a1 = hardlim_apply(repmat(b1,1,Q) + IW1_1*x1); %
realiza as operações da rede

%% Saída
y1 = a1;

toc % fim da função de tempo
end

%% Declaração da Função de ativação
function a = hardlim_apply(n,~)
  a = double(n >= 0);
  a(isnan(n)) = nan;
end
```

APÊNDICE C – SCRIPT PARA CRIAÇÃO E TREINAMENTO DA REDE 2

```
load iris_dataset;
x = irisInputs;
t = irisTargets;

%% Criação da RNA
trainFcn = 'trainlm';                              %
Definição do algoritmo de treinamento utilizado

hiddenLayerSize =4;                                %
quantidade de neurônios na camada oculta

net = feedforwardnet(hiddenLayerSize);             %
declaração da rna

%% Divisão dos dados para serem utilizados no
treinamento
net.divideFcn = 'dividerand';
net.divideMode = 'sample';
net.divideParam.trainRatio = 60/100;
net.divideParam.valRatio = 20/100;
net.divideParam.testRatio = 20/100;

%% Seleção da função de ativação utilizada nos
neurônios de ambas camadas
net.layers{1}.transferFcn = 'tansig';
net.layers{2}.transferFcn = 'tansig';

%% Plots disponiveis no fim do treinamento
net.plotFcns =
{'plotperform','plottrainstate','ploterrhist', ...
    'plotconfusion', 'plotroc','plotregression' };


%% Treinamento e teste
[net,tr] = train(net,x,t);       % treinamento da rede
com os dados selecionados

% Teste da rede
y = net(x);
e = gsubtract(t,y);
performance = perform(net,t,y)
tind = vec2ind(t);
yind = vec2ind(y);
```

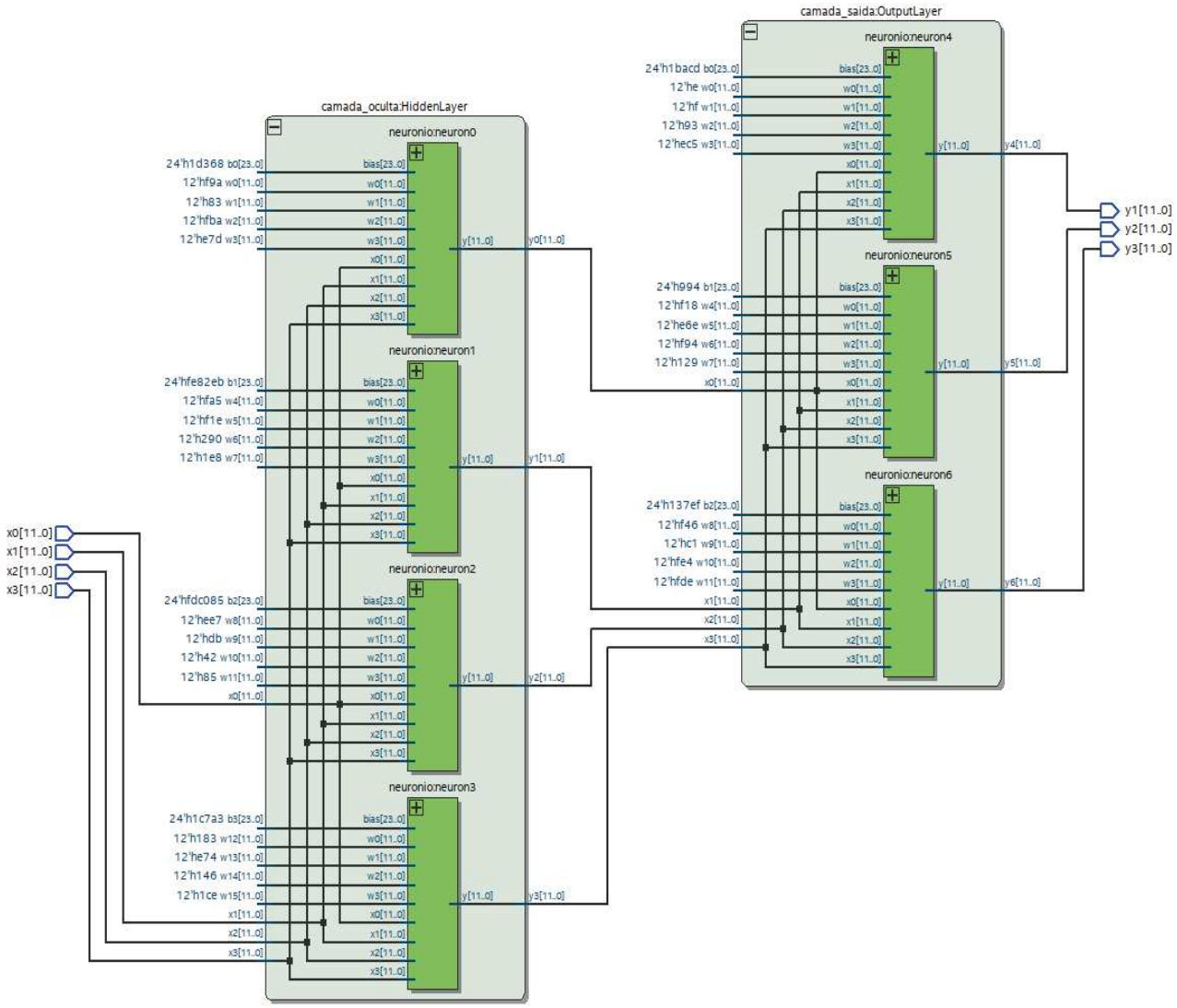
```
percentErrors = sum(tind ~= yind)/numel(tind);

% Recalcular treinamento, validação e performance
trainTargets = t .* tr.trainMask{1};
valTargets = t .* tr.valMask{1};
testTargets = t .* tr.testMask{1};
trainPerformance = perform(net,trainTargets,y)
valPerformance = perform(net,valTargets,y)
testPerformance = perform(net,testTargets,y)

% Vizualização da arquitetura
view(net)

% Plots
%figure, plotperform(tr)
%figure, plottrainstate(tr)
%figure, ploterrhist(e)
figure, plotconfusion(t,y)
%figure, plotroc(t,y)
%figure, plotregression(t,y)

%% Gera uma função para aquisição dos pesos e bias
if (false)
    genFunction(net,'funcao_rna');
    y = funcao_rna(x);
end
```

APÊNDICE D – SCRIPT PARA SIMULAÇÃO DA REDE 2

```
function [y1] = funcao_rna(x1)

%% Parametros da normalização
x1_step1.xoffset = [4.3;2;1;0.1];
x1_step1.gain =
[0.555555555555555;0.833333333333333;0.338983050847458
;0.833333333333333];
x1_step1.ymin = -1;

%% Declaração dos pesos e bias
% Camada oculta
b1 = [1.8258194965448227;-1.4886130953284491;-
2.2479821654260501;1.7798399265897364];
IW1_1 = [-0.39901580427715266 0.51481888277430354 -
0.27403640667323409 -1.5128889528954255;-
0.35789067085553977 -0.883160262792451412
2.5649534164184655 1.9068078926737739;-
1.098605870640331 0.8587945803042214
0.26023795153693213
0.52162032580764517;1.5137701384459961 -
1.5491426870981619 1.2752523556207633
1.8068551816289158];

% Camada de saída
b2 =
[1.7297043950244477;0.037428729205477455;1.21849171315
31301];
LW2_1 = [0.055348544261399714 0.058987318278496903
0.57771621003380391 -1.2310609743526915;-
0.907243601320327 -1.5740265321696392 -
0.42282283933239267 1.1602577928464461;-
0.72839507295651085 0.75431267705990379 -
0.11168872738357107 -0.13374493780096891];

% Paramentros para saída
y1_step1.ymin = -1;
y1_step1.gain = [2;2;2];
y1_step1.xoffset = [0;0;0];

%% Simulação

Q = size(x1,2);
```

```
xp1 = mapminmax_apply(x1,x1_step1); % Normaliza os
dados de entrada


a1 = tansig_apply(repmat(b1,1,Q) + IW1_1*xp1);    %
Operações realizadas na camada oculta

a2 = tansig_apply(repmat(b2,1,Q) + LW2_1*a1);     %
Operações na camada de saída


y1 = mapminmax_reverse(a2,y1_step1);              %
Saída
end

%% Funções utilizadas

% Processamento da entrada
function y = mapminmax_apply(x,settings)
  y = bsxfun(@minus,x,settings.xoffset);
  y = bsxfun(@times,y,settings.gain);
  y = bsxfun(@plus,y,settings.ymin);
end

% Tangente Hiperbolica
function a = tansig_apply(n,~)
  a = 2 ./ (1 + exp(-2*n)) - 1;
end

% Processamento da saída
function x = mapminmax_reverse(y,settings)
  x = bsxfun(@minus,y,settings.ymin);
  x = bsxfun(@rdivide,x,settings.gain);
  x = bsxfun(@plus,x,settings.xoffset);
end
```

APÊNDICE E – BLOCOS QUE COMPÕE O MÓDULO *rede_mlp*



APÊNDICE F – DESCRIÇÃO DA REDE 1 EM VHDL

```
------------------------------------------------
--Multiplicador
------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.neuron_types.all;

entity multiplicador is
	port (x		:		in std_logic_vector (input_size-1 downto
0);

	w0,w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12,w13,w14,w15,w16,w1
7,w18,w19	:	in signed (weight_length-1 downto 0);

	p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14,p15,p16,p17,p18,
p19	:	out signed (weight_length-1 downto 0)
	);

end multiplicador;

architecture hardware of multiplicador is

type weight_type is array (input_size-1 downto 0) of signed (weight_length-1
downto 0);
signal weight_vector: weight_type;
signal prod_vector  : weight_type;

begin
weight_vector(0)<=w0;
weight_vector(1)<=w1;
weight_vector(2)<=w2;
```

```
weight_vector(3)<=w3;
weight_vector(4)<=w4;
weight_vector(5)<=w5;
weight_vector(6)<=w6;
weight_vector(7)<=w7;
weight_vector(8)<=w8;
weight_vector(9)<=w9;
weight_vector(10)<=w10;
weight_vector(11)<=w11;
weight_vector(12)<=w12;
weight_vector(13)<=w13;
weight_vector(14)<=w14;
weight_vector(15)<=w15;
weight_vector(16)<=w16;
weight_vector(17)<=w17;
weight_vector(18)<=w18;
weight_vector(19)<=w19;

gen0: for i in 0 to input_size-1 generate
                    prod_vector(i)<= weight_vector(i) when x(i) ='1' else
                    (others=>'0');
                    end generate;

p0<=prod_vector(0);
p1<=prod_vector(1);
p2<=prod_vector(2);
p3<=prod_vector(3);
p4<=prod_vector(4);
p5<=prod_vector(5);
p6<=prod_vector(6);
p7<=prod_vector(7);
p8<=prod_vector(8);
p9<=prod_vector(9);
p10<=prod_vector(10);
```

```
p11<=prod_vector(11);
p12<=prod_vector(12);
p13<=prod_vector(13);
p14<=prod_vector(14);
p15<=prod_vector(15);
p16<=prod_vector(16);
p17<=prod_vector(17);
p18<=prod_vector(18);
p19<=prod_vector(19);
end hardware;

---------------------------------
-- Acumulador
---------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.neuron_types.all;

entity acumulador is
        port
(p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14,p15,p16,p17,p18,p19
        :       in signed (weight_length-1 downto 0);
                        bias

                                                                :   in  signed
(2*weight_length-1 downto 0);
                        sigma

                                                        :       out     signed
(2*weight_length-1 downto 0)
);
end acumulador;
```

```
architecture hardware of acumulador is
type weight_type is array (input_size-1 downto 0) of signed (weight_length-1 downto 0);
type bias_type   is array (input_size   downto 0) of signed (2*weight_length-1 downto 0);
signal prod_vector  : weight_type;
signal bias_vector  : weight_type;
signal aux_acum :bias_type;

begin
prod_vector(0)<=p0;
prod_vector(1)<=p1;
prod_vector(2)<=p2;
prod_vector(3)<=p3;
prod_vector(4)<=p4;
prod_vector(5)<=p5;
prod_vector(6)<=p6;
prod_vector(7)<=p7;
prod_vector(8)<=p8;
prod_vector(9)<=p9;
prod_vector(10)<=p10;
prod_vector(11)<=p11;
prod_vector(12)<=p12;
prod_vector(13)<=p13;
prod_vector(14)<=p14;
prod_vector(15)<=p15;
prod_vector(16)<=p16;
prod_vector(17)<=p17;
prod_vector(18)<=p18;
prod_vector(19)<=p19;

aux_acum(0)<=bias;
gen1: for i in 0 to input_size-1 generate
                    aux_acum(i+1)<= prod_vector(i) + aux_acum(i);
```

```
                    end generate;

sigma<= aux_acum(input_size);

end hardware;

---------------------------------
--Função de ativação
-------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.neuron_types.all;

entity hardlim is
        port (sigma :   in signed (2*weight_length-1 downto 0);
                        y               :       out     std_logic
);
end hardlim;

architecture hardware of hardlim is

begin

        y<= '1' when sigma >=0 else
                '0';

end hardware;

-------------------------
--neuronio
-------------------------
library ieee;
use ieee.std_logic_1164.all;
```

```
use ieee.numeric_std.all;
use work.neuron_types.all;

entity neuronio is
        port (x         :           in      std_logic_vector    (input_size-1
downto 0);

        w0,w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12,w13,w14,w15,w16,w1
7,w18,w19: in signed (weight_length-1 downto 0);
                        bias    :       in signed (2*weight_length-1 downto 0);
                        y : out std_logic
);
end neuronio;

architecture ligacoes of neuronio is

component multiplicador is
        port (x         :           in std_logic_vector (input_size-1 downto
0);

        w0,w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12,w13,w14,w15,w16,w1
7,w18,w19       :       in signed (weight_length-1 downto 0);

        p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14,p15,p16,p17,p18,
p19     :       out signed (weight_length-1 downto 0)
        );

end component;

component acumulador is
        port
(p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14,p15,p16,p17,p18,p19
        :       in signed (weight_length-1 downto 0);
```

```
                    bias

                                                    :    in    signed
(2*weight_length-1 downto 0);
                    sigma

                                                    :    out    signed
(2*weight_length-1 downto 0)
);
end component;

component hardlim is
        port (sigma :    in signed (2*weight_length-1 downto 0);
                    y              :    out    std_logic
);
end component;

signal pd0, pd1, pd2, pd3, pd4, pd5, pd6, pd7, pd8, pd9, pd10, pd11, pd12, pd13,
pd14, pd15, pd16, pd17, pd18, pd19    :    signed (weight_length-1
downto 0);
signal aux_sigma : signed (2*weight_length-1 downto 0);

begin

mult    :    multiplicador    PORT    MAP    (x,
w0,w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12,w13,w14,w15,w16,w17,w18,w
19, pd0, pd1, pd2, pd3, pd4, pd5, pd6, pd7, pd8, pd9, pd10, pd11, pd12, pd13,
pd14, pd15, pd16, pd17, pd18, pd19);
acum    :  acumulador    PORT MAP (pd0, pd1, pd2, pd3, pd4, pd5, pd6, pd7,
pd8, pd9, pd10, pd11, pd12, pd13, pd14, pd15, pd16, pd17, pd18, pd19, bias,
aux_sigma);
funcao  :  hardlim              PORT MAP (aux_sigma, y);

end ligacoes;
```

```
---------------------------------
--Rede neural
---------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.neuron_types.all;

entity rede_neural is
        port (x         :       in std_logic_vector (0 to input_size-1);
                        y       :       out  std_logic_vector  (0
to 4)
);
end rede_neural;

architecture ligacoes of rede_neural is
component neuronio is
        port (x         :       in      std_logic_vector    (input_size-1
downto 0);

        w0,w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12,w13,w14,w15,w16,w1
7,w18,w19: in signed (weight_length-1 downto 0);
                        bias    :       in signed (2*weight_length-1 downto 0);
                        y : out std_logic
);
end component;

begin

neuron0:        neuronio        PORT        MAP        (x,
pw0,pw1,pw2,pw3,pw4,pw5,pw6,pw7,pw8,pw9,pw10,pw11,pw12,pw13,pw14,p
w15,pw16,pw17,pw18,pw19, bias0, y(0));
```

```
neuron1: neuronio PORT MAP (x,
pw20,pw21,pw22,pw23,pw24,pw25,pw26,pw27,pw28,pw29,pw30,pw31,pw32,p
w33,pw34,pw35,pw36,pw37,pw38,pw39, bias1, y(1));
neuron2: neuronio PORT MAP (x,
pw40,pw41,pw42,pw43,pw44,pw45,pw46,pw47,pw48,pw49,pw50,pw51,pw52,p
w53,pw54,pw55,pw56,pw57,pw58,pw59, bias2, y(2));
neuron3: neuronio PORT MAP (x,
pw60,pw61,pw62,pw63,pw64,pw65,pw66,pw67,pw68,pw69,pw70,pw71,pw72,p
w73,pw74,pw75,pw76,pw77,pw78,pw79, bias3, y(3));
neuron4: neuronio PORT MAP (x,
pw80,pw81,pw82,pw83,pw84,pw85,pw86,pw87,pw88,pw89,pw90,pw91,pw92,p
w93,pw94,pw95,pw96,pw97,pw98,pw99, bias4, y(4));

end ligacoes;
```

APÊNDICE G – DESCRIÇÃO DA REDE 2 EM VHDL

```
----------------------------------------------
--Pacote de dados com pesos e bias
----------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package neuron_types is
        constant input_size             :       natural         :=4;
                                --quantidade de entradas de cada neuronio
        constant input_length   :       natural         :=12;
                        --tamanho dos dados de entrada
        constant output_size            :       natural         :=3;
                                --quantidade de saidas de cada neuronio
        constant output_length  :       natural         :=12;
                        --tamanho dos dados de saida
        constant n_camada_oculta:       natural         :=4;
                        -- quantidade de neuronios na camada oculta
        constant n_camada_saida :       natural         :=3;
                        -- quantidade de neuronios na camada de saida;



        -- declaração dos pesos e bias : ***CAMADA OCULTA***----------------
        -- neuronio 0:
        CONSTANT w_0 :      signed          (input_length-1         downto
0):="111110011010";
        CONSTANT w_1 :      signed          (input_length-1         downto
0):="000010000011";
```

```
        CONSTANT w_2 :      signed          (input_length-1          downto
0):="111110111010";
        CONSTANT w_3 :      signed          (input_length-1          downto
0):="111001111101";
        -- neuronio 1:
        CONSTANT w_4 :      signed          (input_length-1          downto
0):="111110100101";
        CONSTANT w_5 :      signed          (input_length-1          downto
0):="111100011110";
        CONSTANT w_6 :      signed          (input_length-1          downto
0):="001010010000";
        CONSTANT w_7 :      signed          (input_length-1          downto
0):="000111101000";
        -- neuronio 2:
        CONSTANT w_8 :      signed          (input_length-1          downto
0):="111011100111";
        CONSTANT w_9 :      signed          (input_length-1          downto
0):="000011011011";
        CONSTANT w_10:      signed          (input_length-1          downto
0):="000001000010";
        CONSTANT w_11: signed (input_length-1 downto 0):="000010000101";
        -- neuronio 3:
        CONSTANT w_12 :     signed          (input_length-1          downto
0):="000110000011";
        CONSTANT w_13 :     signed          (input_length-1          downto
0):="111001110100";
        CONSTANT w_14 :     signed          (input_length-1          downto
0):="000101000110";
        CONSTANT w_15 :     signed          (input_length-1          downto
0):="000111001110";

        -- bias :
        CONSTANT b_0 :      signed   (input_length+output_length-1   downto
0):="000000011101001101101000";
```

```
        CONSTANT b_1 :      signed  (input_length+output_length-1  downto
0):="11111110100000101110101 1";
        CONSTANT b_2 :      signed  (input_length+output_length-1  downto
0):="11111101110000001000010 1";
        CONSTANT b_3 :      signed  (input_length+output_length-1  downto
0):="00000001110001111010001 1";

        -- declaração dos pesos e bias : ***CAMADA DE SAIDA***----------------
        -- neuronio 4:
        CONSTANT w_16 :     signed          (input_length-1         downto
0):="000000001110";
        CONSTANT w_17 :     signed          (input_length-1         downto
0):="000000001111";
        CONSTANT w_18 :     signed          (input_length-1         downto
0):="000010010011";
        CONSTANT w_19 :     signed          (input_length-1         downto
0):="111011000101";
        -- neuronio 5:
        CONSTANT w_20 :     signed          (input_length-1         downto
0):="111100011000";
        CONSTANT w_21 :     signed          (input_length-1         downto
0):="111001101110";
        CONSTANT w_22 :     signed          (input_length-1         downto
0):="111110010100";
        CONSTANT W_23 :     signed          (input_length-1         downto
0):="000100101001";
        -- neuronio 6:
        CONSTANT w_24 :     signed          (input_length-1         downto
0):="111101000110";
        CONSTANT w_25 :     signed          (input_length-1         downto
0):="000011000001";
        CONSTANT w_26 :     signed          (input_length-1         downto
0):="111111100100";
```

```
        CONSTANT w_27 : signed (input_length-1 downto
0):="111111011110";
                -- bias :
        CONSTANT b_4 : signed (input_length+output_length-1 downto
0):="000000011011101011001101";
        CONSTANT b_5 : signed (input_length+output_length-1 downto
0):="000000000000100110010100";
        CONSTANT b_6 : signed (input_length+output_length-1 downto
0):="000000010011011111101111";

end neuron_types;
```

```
----------------------------------
--bloco offset na entrada
--------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.neuron_types.all;

-- ESSE BLOCO SIMULA A FUNÇÃO BSXFUN UTILIZANDO A FUNÇÃO
@PLUS DO MATLAB
entity offset is
        port ( i_0, i_1, i_2, i_3 : in signed
(input_length-1 downto 0);
                o_0, o_1, o_2, o_3 : out
                signed (input_length-1 downto 0)
);
end offset;
```

```
architecture hardware of offset is
constant aux0 : signed (input_length-1 downto 0):="010001001100";
constant aux1 : signed (input_length-1 downto 0):="001000000000";
constant aux2 :   signed (input_length-1 downto 0):="000100000000";
constant aux3 :   signed (input_length-1 downto 0):="000000011001";

signal sig0 : signed (input_length-1 downto 0);
signal sig1 : signed (input_length-1 downto 0);
signal sig2 : signed (input_length-1 downto 0);
signal sig3 : signed (input_length-1 downto 0);



begin
sig0<=i_0-aux0;
sig1<=i_1-aux1;
sig2<=i_2-aux2;
sig3<=i_3-aux3;

o_0<=sig0;
o_1<=sig1;
o_2<=sig2;
o_3<=sig3;

end hardware;

---------------------------------
--bloco times
---------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.neuron_types.all;
```

```
entity times is
        port ( i_0, i_1, i_2, i_3          :          in          signed
(input_length-1 downto 0);
                        o_0, o_1, o_2, o_3          :          out
                signed (input_length-1 downto 0)
);
end times;

architecture hardware of times is
constant aux0 : signed (input_length-1 downto 0):="000010001110";
constant aux1 : signed (input_length-1 downto 0):="000011010101";
constant aux2 :   signed (input_length-1 downto 0):="000001010110";
constant aux3 :   signed (input_length-1 downto 0):="000011010101";

signal sig0 : signed (2*input_length-1 downto 0);
signal sig1 : signed (2*input_length-1 downto 0);
signal sig2 : signed (2*input_length-1 downto 0);
signal sig3 : signed (2*input_length-1 downto 0);

signal sig_0 : signed (input_length-1 downto 0);
signal sig_1 : signed (input_length-1 downto 0);
signal sig_2 : signed (input_length-1 downto 0);
signal sig_3 : signed (input_length-1 downto 0);

begin
sig0<=i_0*aux0;
sig_0<=sig0(19) & sig0(18 downto 16) & sig0(15 downto 8);

sig1<=i_1*aux1;
sig_1<=sig1(19) & sig1(18 downto 16) & sig1(15 downto 8);

sig2<=i_2*aux2;
sig_2<=sig2(19) & sig2(18 downto 16) & sig2(15 downto 8);
```

```
sig3<=i_3*aux3;
sig_3<=sig3(19) & sig3(18 downto 16) & sig3(15 downto 8);

o_0<=sig_0;
o_1<=sig_1;
o_2<=sig_2;
o_3<=sig_3;
end hardware;

-------------------------------------
--bloco plus
-------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.neuron_types.all;

entity plus is
        port ( i_0, i_1, i_2, i_3          :          in          signed
(input_length-1 downto 0);
                        o_0, o_1, o_2, o_3          :          out
                signed (input_length-1 downto 0)
);
end plus;

architecture hardware of plus is
constant aux0 : signed (input_length-1 downto 0):="111100000000";
constant aux1 : signed (input_length-1 downto 0):="111100000000";
constant aux2 :   signed (input_length-1 downto 0):="111100000000";
constant aux3 :   signed (input_length-1 downto 0):="111100000000";

signal sig0 : signed (input_length-1 downto 0);
signal sig1 : signed (input_length-1 downto 0);
signal sig2 : signed (input_length-1 downto 0);
```

```
signal sig3 : signed (input_length-1 downto 0);

begin
sig0<=i_0+aux0;
sig1<=i_1+aux1;
sig2<=i_2+aux2;
sig3<=i_3+aux3;

o_0<=sig0;
o_1<=sig1;
o_2<=sig2;
o_3<=sig3;

end hardware;



--------------------------------
-- bloco normalizador
---------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.neuron_types.all;

entity bloco_normalizador is
        port ( x0, x1, x2, x3                   :           in
        signed (input_length-1 downto 0);
                        out0, out1, out2, out3  :           out     signed
(output_length-1 downto 0)
);
end bloco_normalizador;

architecture ligacoes of bloco_normalizador is
component offset is
```

```
        port ( i_0, i_1, i_2, i_3          :          in          signed
(input_length-1 downto 0);
                        o_0, o_1, o_2, o_3          :          out
                        signed (input_length-1 downto 0)
);
end component;

component times is
        port ( i_0, i_1, i_2, i_3          :          in          signed
(input_length-1 downto 0);
                        o_0, o_1, o_2, o_3          :          out
                signed (input_length-1 downto 0)
);
end component;

component plus is
        port ( i_0, i_1, i_2, i_3          :          in          signed
(input_length-1 downto 0);
                        o_0, o_1, o_2, o_3          :          out
                signed (input_length-1 downto 0)
);
end component;

-------------------------------------------------------------------------------
signal sig_off0, sig_off1, sig_off2, sig_off3      :      signed    (input_length-1
downto 0);
signal sig_tim0, sig_tim1, sig_tim2, sig_tim3      :      signed    (input_length-1
downto 0);
-------------------------------------------------------------------------------
begin
bloco_offset :    offset PORT MAP (x0, x1, x2, x3, sig_off0, sig_off1, sig_off2,
sig_off3);
bloco_time       : times    PORT MAP (sig_off0, sig_off1, sig_off2, sig_off3,
sig_tim0, sig_tim1, sig_tim2, sig_tim3);
```

```
bloco_plus : plus PORT MAP (sig_tim0, sig_tim1, sig_tim2, sig_tim3,
out0, out1, out2, out3);
end ligacoes;


-------------------------------------
--Multiplicador
-------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.neuron_types.all;

entity multiplicador is
        port    (x0, x1, x2, x3                                 :
        in              signed (input_length -1 downto 0);
                         w0, w1, w2, w3                                 :
                in              signed (output_length-1 downto 0);
                         prod0, prod1, prod2, prod3     :               out
        signed (input_length+output_length-1 downto 0)
                );
end multiplicador;

architecture hardware of multiplicador is

type input_type_vector          is array (input_size-1 downto 0) of signed
(input_length-1 downto 0);
type buffer_type                                is array (input_size-1 downto 0)
of signed (input_length+output_length-1 downto 0);
type prod                                               is array (input_size-1
downto 0) of signed(input_length+output_length-1 downto 0);

signal aux_x, aux_w :   input_type_vector;
signal aux_prod  :  buffer_type:=((others=>(others=>'0')));
```

```
begin
        aux_x(0)<=x0;
        aux_x(1)<=x1;
        aux_x(2)<=x2;
        aux_x(3)<=x3;

        aux_w(0)<=w0;
        aux_w(1)<=w1;
        aux_w(2)<=w2;
        aux_w(3)<=w3;

        gen0: FOR i IN 0 TO input_size-1 GENERATE
                aux_prod(i)<= aux_x(i)*aux_w(i);
        end generate;

        prod0<=aux_prod(0);
        prod1<=aux_prod(1);
        prod2<=aux_prod(2);
        prod3<=aux_prod(3);
end hardware;

-----------------------------------------------
--Acumulador
-----------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.neuron_types.all;

entity acumulador is
        port (prod0, prod1, prod2, prod3                :               in
                signed (input_length+output_length-1 downto 0);
```

```
                bias
                :           in          signed
(input_length+output_length-1 downto 0);
                acum
                :           out     signed
(input_length+output_length-1 downto 0)
);
end acumulador;

architecture hardware of acumulador is

TYPE buffer_type    is ARRAY (input_size downto 0)      of      signed
(input_length+output_length-1 downto 0);
TYPE prod_type          is ARRAY (input_size-1 downto 0) of signed
(input_length+output_length-1 downto 0);

signal aux_acum     :   buffer_type :=((others=>(others=>'0')));
signal aux_prod:  prod_type     :=((others=>(others=>'0')));

begin
        aux_prod(0)<=prod0;
        aux_prod(1)<=prod1;
        aux_prod(2)<=prod2;
        aux_prod(3)<=prod3;

        aux_acum(0)<=bias;

        gen1: FOR i IN 0 TO input_size-1 GENERATE
                aux_acum(i+1)<=aux_prod(i)+aux_acum(i);
        end generate;

        acum<=aux_acum(input_size);
end hardware;
```

```
------------------------------------
--bloco operações lineares
------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.neuron_types.all;

entity operacoes_lineares is
        port    (x0, x1, x2, x3                                 :
        in              signed (input_length -1 downto 0);
                        w0, w1, w2, w3                                  :
                in              signed (output_length-1 downto 0);
                        bias
                        :               in              signed
(input_length+output_length-1 downto 0);
                        sigma
                :               out     signed  (input_length+output_length-1
downto 0)
);
end operacoes_lineares;

architecture hardware of operacoes_lineares is

component multiplicador is
        port    (x0, x1, x2, x3                                 :
        in              signed (input_length -1 downto 0);
                        w0, w1, w2, w3                                  :
                in              signed (output_length-1 downto 0);
                        prod0, prod1, prod2, prod3      :               out
        signed (input_length+output_length-1 downto 0)
                );
end component;
```

```
component acumulador is
        port (prod0, prod1, prod2, prod3 : in
                signed (input_length+output_length-1 downto 0);
                bias
                : in signed
(input_length+output_length-1 downto 0);
                acum
                : out signed
(input_length+output_length-1 downto 0)
);
end component;

-----------------------------------------------------------------------------------------------

signal s_prod0, s_prod1, s_prod2, s_prod3 : signed (input_length+output_length-
1 downto 0);

-----------------------------------------------------------------------------------------------

BEGIN

bloco_multiplicador: multiplicador PORT MAP (x0, x1, x2, x3, w0, w1, w2, w3,
s_prod0, s_prod1, s_prod2, s_prod3);

bloco_acumulador: acumulador PORT MAP (s_prod0,
s_prod1, s_prod2, s_prod3, bias, sigma);

end hardware;


-------------------------------------
--complemento de 2
-------------------------------------
library ieee;
```

```
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.neuron_types.all;

entity complemento_2 is
	port ( complem_in		:		in
	signed (input_length+output_length-1 downto 0);
			complem_out		:		out	signed
(input_length+output_length-1 downto 0)
);
end complemento_2;

architecture hardware of complemento_2 is
signal aux_complem0		:		signed
((2*(input_length+output_length))-1 downto 0);
signal aux_complem1		:		signed
(input_length+output_length-1 downto 0);

begin

	aux_complem0<= complem_in* "111111110000000000000000";

		-- multiplica por -1
	aux_complem1<= aux_complem0(39) & aux_complem0(38 downto 32)
& aux_complem0(31 downto 16); -- salva apenas o necessário da multiplicação
(complemento de dois)

	complem_out <= complem_in when complem_in(23) ='0' else
						aux_complem1		when
complem_in(23)= '1' else

	"000000000000000000000000";

end hardware;
```

```
----------------------------------------
--calculo dos coeficientes
----------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.neuron_types.all;

entity coeficientes is
        port (complem_2          :          in          signed
(input_length+output_length-1 downto 0);
                        coef_A          :          out      signed
(input_length+output_length-1 downto 0);
                        coef_B          :          out      signed
(input_length+output_length-1 downto 0)
);
end coeficientes;

architecture hardware of coeficientes is

begin
coef_A              <="000000001110111000000111"          when
complem_2>"000000000000000000000000"                      AND
complem_2<="000000001111101100001000" else -- reta 1
                    "000000001011100000100100"            when
complem_2>"000000001111101100001000"                      AND
complem_2<="000000001011001010010101" else -- reta 2
                    "000000001000011001011001"            when
complem_2>"000000001011001010010101"                      AND
complem_2<="000000010000000000000000" else -- reta 3
                    "000000000101011111010101"            when
complem_2>"000000010000000000000000"                      AND
complem_2<="000000010100001110010101" else -- reta 4
```

```
                    "000000000011010011110000"          when
complem_2>"000000010100001110010101"                    AND
complem_2<="000000011001100110011001" else -- reta 5
                    "000000000001110010000100"          when
complem_2>"000000011001100110011001"                    AND
complem_2<="000000011111000010101010" else -- reta 6
                    "000000000000110100001110"          when
complem_2>"000000011111000010101010"                    AND
complem_2<="000000100110111100011010" else -- reta 7
                    "000000000000010010100010"          when
complem_2>"000000100110111100011010"                    AND
complem_2<="000000110000000000000000" else -- reta 8
                    "000000000000000010011101"          when
complem_2>"000000110000000000000000"                    AND
complem_2<="000001010000000000000000" else -- reta 9
                    "000000000000000000000000";

coef_B              <="000000000000000000000000"        when
complem_2>="000000000000000000000000"                   AND
complem_2<="000000000111101100001000" else -- reta 1
                    "000000000001100111011011"          when
complem_2>"000000000111101100001000"                    AND
complem_2<="000000001011001010010101" else -- reta 2
                    "000000000011110010011000"          when
complem_2>"000000001011001010010101"                    AND
complem_2<="000000010000000000000000" else -- reta 3
                    "000000000110101100011100"          when
complem_2>"000000010000000000000000"                    AND
complem_2<="000000010100001110010101" else -- reta 4
                    "000000001001011100111000"          when
complem_2>"000000010100001110010101"                    AND
complem_2<="000000011001100110011001" else -- reta 5
```

```
                    "00000000101111100011101l"          when
complem_2>"00000001100110011001l"                       AND
complem_2<="00000001111100001010101O" else -- reta 6
                    "00000000110111000100100l"          when
complem_2>"00000001111100001010101O"                    AND
complem_2<="00000100110111100011010" else -- reta 7
                    "00000000111100001101000l"          when
complem_2>"00000100110111100011010"                     AND
complem_2<="00000110000000000000000" else -- reta 8
                    "00000000111111001110011l"          when
complem_2>"00000110000000000000000"                     AND
complem_2<="00001010000000000000000" else -- reta 9
                    "00000001000000000000000"; -- alterei aqui , antes
era tudo zero
end hardware;

----------------------------------------
--calculo da reta
----------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.neuron_types.all;

entity calculo_reta is
        port ( sigma        :       in      signed
(input_length+output_length-1 downto 0);
                    complem_2   :       in      signed
(input_length+output_length-1 downto 0);
                    coef_A      :       in      signed
(input_length+output_length-1 downto 0);
                    coef_B      :       in      signed
(input_length+output_length-1 downto 0);
```

```
                resultado : out signed
(output_length-1 downto 0)
);
end calculo_reta;

architecture hardware of calculo_reta is

signal aux_mult0 : signed
(2*(input_length+output_length)-1 downto 0);
signal aux_mult1 : signed (input_length+output_length-1
downto 0);
signal sum_coef : signed (input_length+output_length-1
downto 0);
signal aux_resultado0 : signed (input_length-1 downto 0);
signal aux_resultado1 : signed (input_length+output_length-1 downto 0);
signal aux_resultado2 : signed (input_length-1 downto 0);

begin


        aux_mult0 <= coef_A*complem_2;
        aux_mult1 <= aux_mult0(39) & aux_mult0(38 downto 32) &
aux_mult0(31 downto 16);
        sum_coef <= aux_mult1+coef_B;

        aux_resultado0<= sum_coef(19) & sum_coef(18 downto 16) &
sum_coef(15 downto 8);                  -- resultado positivo se numero
de entrada positivo

        aux_resultado1<= aux_resultado0 * "111100000000";
        aux_resultado2<= aux_resultado1(19) & aux_resultado1(18 downto 16)
& aux_resultado1(15 downto 8);          -- complemento de 2 se sinal de entrada
negativo
```

```
        resultado<= aux_resultado0 when sigma(input_length+output_length-1)
='0' else
                                        aux_resultado2          when
sigma(input_length+output_length-1) ='1' else
                                        (others=>'0');



end hardware;

-------------------------------
--bloco função tangente hiperbólica
-------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.neuron_types.all;

entity tangente_hiperbolica is
        port ( sigma                    :           in
        signed (input_length+output_length-1 downto 0);
                        y                                       :
                out     signed (output_length-1 downto 0)
);
end tangente_hiperbolica;

architecture hardware of tangente_hiperbolica is

component complemento_2 is
        port ( complem_in               :           in
        signed (input_length+output_length-1 downto 0);
                        complem_out     :           out     signed
(input_length+output_length-1 downto 0)
);
```

```
end component;

component coeficientes is
        port (complem_2          :          in          signed
(input_length+output_length-1 downto 0);
                    coef_A          :          out      signed
(input_length+output_length-1 downto 0);
                    coef_B          :          out      signed
(input_length+output_length-1 downto 0)
);
end component;

component calculo_reta is
        port ( sigma          :          in          signed
(input_length+output_length-1 downto 0);
                    complem_2     :          in          signed
(input_length+output_length-1 downto 0);
                    coef_A          :          in          signed
(input_length+output_length-1 downto 0);
                    coef_B          :          in          signed
(input_length+output_length-1 downto 0);
                    resultado       :          out      signed
(output_length-1 downto 0)
);
end component;

-------------------------------------------------------------------------------
signal sig_complem_2                    :                    signed
(input_length+output_length-1 downto 0);
signal sig_coef_A, sig_coef_B : signed   (input_length+output_length-1   downto
0);
-------------------------------------------------------------------------------

begin
```

```
bloco_complemento_2: complemento_2 PORT MAP (sigma, sig_complem_2);
bloco_coeficientes : coeficientes  PORT MAP (sig_complem_2, sig_coef_A,
sig_coef_B);
bloco_calculo_reta : calculo_reta  PORT MAP (sigma, sig_complem_2,
sig_coef_A, sig_coef_B, y);

end hardware;

---------------------------------
--bloco do neurônio
---------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.neuron_types.all;

entity neuronio is
        port (    x0, x1, x2, x3                                    :
        in              signed (input_length -1 downto 0);
                        w0, w1, w2, w3
        :               in              signed  (output_length-1  downto  0);

                        bias
                        :               in              signed
(input_length+output_length-1 downto 0);
                        y
                        :               out     signed (output_length-1
downto 0)
);
end neuronio;

architecture hardware of neuronio is
```

```
component operacoes_lineares is
        port    (x0, x1, x2, x3                                 :
        in              signed (input_length -1 downto 0);
                        w0, w1, w2, w3                          :
                in              signed (input_length-1 downto 0);
                        bias
                        :               in              signed
(input_length+output_length-1 downto 0);
                        sigma
                :               out     signed  (input_length+output_length-1
downto 0)
);
end component;

component tangente_hiperbolica is
        port ( sigma                            :               in
        signed (input_length+output_length-1 downto 0);
                        y                                       :
                out     signed (output_length-1 downto 0)
);
end component;

-------------------------------------------------------------------------------------
signal sig_sigma :      signed (input_length+output_length-1 downto 0);
-------------------------------------------------------------------------------------

begin

bloco_op_lin:   operacoes_lineares      PORT                            MAP
(x0,x1,x2,x3,w0,w1,w2,w3,bias, sig_sigma);
bloco_tansig:   tangente_hiperbolica    PORT MAP (sig_sigma, y);

end hardware;
```

```
----------------------------------------
--bloco camada oculta
----------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.neuron_types.all;

entity camada_oculta is
        port (  x0, x1, x2, x3                          :               in
                signed (input_length-1 downto 0);
        -- dados de entrada
                                w0,w1,w2,w3
        :               in              signed (input_length-1 downto 0);
                                w4,w5,w6,w7
        :               in              signed (input_length-1 downto 0);
                                w8,w9,w10,w11                   :
                in              signed (input_length-1 downto 0);
                                w12,w13,w14,w15                 :
                in              signed (input_length-1 downto 0);
                                b0,b1,b2,b3
        :               in              signed  (input_length+output_length-1
downto 0);
                                y0, y1, y2, y3                  :
                out     signed (output_length-1 downto 0)
);
end camada_oculta;

architecture hardware of camada_oculta is

component neuronio is
        port (  x0, x1, x2, x3                          :
        in              signed (input_length -1 downto 0);
```

```
                                w0, w1, w2, w3
        :               in              signed  (input_length-1  downto  0);

                                bias
                        :               in              signed
(input_length+output_length-1 downto 0);
                                y
                                :               out     signed (output_length-1
downto 0)
);
end component;

-------------------------------------------------------------------------------------------
begin

neuron0: neuronio PORT MAP (x0,x1,x2,x3,w0 ,w1 ,w2 ,w3 ,b0,y0);
neuron1: neuronio PORT MAP (x0,x1,x2,x3,w4 ,w5 ,w6 ,w7 ,b1,y1);
neuron2: neuronio PORT MAP (x0,x1,x2,x3,w8 ,w9 ,w10,w11,b2,y2);
neuron3: neuronio PORT MAP (x0,x1,x2,x3,w12,w13,w14,w15,b3,y3);
end hardware;

-------------------------------------
--bloco camada de saída
-------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.neuron_types.all;

entity camada_saida is
        port (  x0, x1, x2, x3                          :               in
                signed (input_length-1 downto 0);
        -- dados de entrada
```

```
                                w0,w1,w2,w3
        :               in              signed (input_length-1 downto 0);
                                w4,w5,w6,w7
        :               in              signed (input_length-1 downto 0);
                                w8,w9,w10,w11                           :
                in              signed (input_length-1 downto 0);

                                b0,b1,b2
                :               in              signed
(input_length+output_length-1 downto 0);
                                y4, y5, y6
        :               out     signed (output_length-1 downto 0)
);
end camada_saida;

architecture hardware of camada_saida is

component neuronio is
        port (  x0, x1, x2, x3                                  :
        in              signed (input_length -1 downto 0);
                                w0, w1, w2, w3
        :               in              signed  (output_length-1  downto  0);

                                bias
                        :               in              signed
(input_length+output_length-1 downto 0);
                                y
                                :               out     signed (output_length-1
downto 0)
);
end component;

-------------------------------------------------------------------------------------------
begin
```

```
neuron4: neuronio PORT MAP (x0,x1,x2,x3,w0 ,w1 ,w2 ,w3 ,b0,y4);
neuron5: neuronio PORT MAP (x0,x1,x2,x3,w4 ,w5 ,w6 ,w7 ,b1,y5);
neuron6: neuronio PORT MAP (x0,x1,x2,x3,w8 ,w9 ,w10,w11,b2,y6);


end hardware;



---------------------------
--bloco rede mlp
---------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.neuron_types.all;

entity rede_mlp is
        port (  x0, x1, x2, x3                          :               in
                signed (input_length-1 downto 0);
        -- dados de entrada
                                y1, y2, y3
        :               out     signed (input_length-1 downto 0)
);
end rede_mlp;

architecture hardware of rede_mlp is

component camada_oculta is
        port (  x0, x1, x2, x3                          :               in
                signed (input_length-1 downto 0);
        -- dados de entrada
```

```
                                w0,w1,w2,w3
        :               in              signed (input_length-1 downto 0);
                                w4,w5,w6,w7
        :               in              signed (input_length-1 downto 0);
                                w8,w9,w10,w11                   :
                in              signed (input_length-1 downto 0);
                                w12,w13,w14,w15                 :
                in              signed (input_length-1 downto 0);
                                b0,b1,b2,b3
        :               in              signed  (input_length+output_length-1
downto 0);
                                y0, y1, y2, y3                  :
                out     signed (output_length-1 downto 0)
);
end component;

component camada_saida is
        port (  x0, x1, x2, x3                          :               in
                signed (input_length-1 downto 0);
        -- dados de entrada
                                w0,w1,w2,w3
        :               in              signed (input_length-1 downto 0);
                                w4,w5,w6,w7
        :               in              signed (input_length-1 downto 0);
                                w8,w9,w10,w11                   :
                in              signed (input_length-1 downto 0);

                                b0,b1,b2
                :               in              signed
(input_length+output_length-1 downto 0);
                                y4, y5, y6
        :               out     signed (output_length-1 downto 0)
);
end component;
```

```
-----------------------------------------------------------------------------------
signal sig_y0, sig_y1, sig_y2, sig_y3 : signed (output_length-1 downto 0);

-----------------------------------------------------------------------------------
begin

HiddenLayer :  camada_oculta  PORT  MAP  (x0,x1,x2,x3,
w_0,w_1,w_2,w_3,w_4,w_5,w_6,w_7,w_8,w_9,w_10,w_11,w_12,w_13,w_14,w
_15,b_0,b_1,b_2,b_3, sig_y0, sig_y1, sig_y2, sig_y3);
OutputLayer :  camada_saida  PORT MAP (sig_y0, sig_y1, sig_y2, sig_y3, w_16,
w_17, w_18, w_19, w_20, w_21, w_22, w_23, w_24, w_25, w_26, w_27, b_4,
b_5, b_6, y1, y2, y3);

end hardware;



-----------------------------
--bloco de saída
-----------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.neuron_types.all;

entity decoder is
        port ( in_dec0, in_dec1, in_dec2 : in  signed (input_length-1 downto 0);
                        dec0, dec1, dec2                        :       out
std_logic
);
end decoder;

architecture hardware of decoder is
signal aux      :       std_logic_vector (2 downto 0);
```

```
begin

aux<= "001" WHEN in_dec2 > in_dec0 AND in_dec2 > in_dec1 ELSE
                    "010" WHEN in_dec1 > in_dec0 AND in_dec1 > in_dec2 ELSE
                    "100" WHEN in_dec0 > in_dec1 AND in_dec0 > in_dec1 ELSE
                    "000";
dec0<=aux(2);
dec1<=aux(1);
dec2<=aux(0);

end hardware;
```