



Pós-Graduação em Ciência da Computação

**“Desenvolvimento de uma Arquitetura Paralela
para Redes Neurais Artificiais MLP baseada em
FPGAs”**

Por

Antonyus Pyetro do Amaral Ferreira

Dissertação de Mestrado



Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, MARÇO/2011



UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ANTONYUS PYETRO DO AMARAL FERREIRA

“Desenvolvimento de uma Arquitetura Paralela para Redes Neurais
Artificiais MLP baseada em FPGAs ”

*ESTE TRABALHO FOI APRESENTADO À PÓS-GRADUAÇÃO EM
CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICA DA
UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO REQUISITO
PARCIAL PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIA DA
COMPUTAÇÃO.*

ORIENTADOR(A): Edna Natividade da Silva Barros

RECIFE, MARÇO/2011

**Catalogação na fonte
Bibliotecária Jane Souto Maior, CRB4-571**

Ferreira, Antonyus Pyetro do Amaral

**Desenvolvimento de uma arquitetura paralela para
redes neurais artificiais MLP baseada em FPGAS /
Antonyus Pyetro do Amaral Ferreira - Recife: O Autor,
2011.**

xviii, 106 folhas: il., fig., tab.

Orientador: Edna Natividade da Silva Barros.

**Dissertação (mestrado) - Universidade Federal de
Pernambuco. CIn, Ciência da computação, 2011.**

Inclui bibliografia, anexo e apêndice.

**1. Inteligência Artificial. 2. Redes neurais. 3. Arquitetura de
computador. 4. FPGA. I. Barros, Edna Natividade de
(orientadora). II. Título.**

006.3

CDD (22. ed.)

MEI2011 – 074

Dissertação de Mestrado apresentada por **Antonyus Pyetro do Amaral Ferreira** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título “**Desenvolvimento de uma Arquitetura Paralela para Redes Neurais Artificiais MLP baseada em FPGAs**”, orientada pela **Profa. Edna Natividade da Silva Barros** aprovada pela Banca Examinadora formada pelos professores:



Prof. Manoel Eusébio de Lima
Centro de Informática / UFPE



Prof. José Antonio Gómes de Lima
Departamento de Informática / UFPB



Profa. Edna Natividade da Silva Barros
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 15 de março de 2011.



Prof. Nelson Souto Rosa
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

Resumo

Este trabalho apresenta a definição de uma arquitetura, baseada em FPGA, para implementação de Redes Neurais (RNAs) MLP. A arquitetura proposta foi projetada observando-se critérios limitantes como grande quantidade de entradas, redução do consumo de área, utilização de pinos, recursos de interconexão e compromisso entre área/desempenho. Um importante resultado é a utilização de $\log_2 m$ adicionadores para uma RNA com m entradas. Uma RNA cuja topologia é 256:10:10 atingiu um speed-up de 36x, comparado com uma implementação convencional em C rodado em um PC. Uma ferramenta de geração automática do código da RNA em linguagem HDL também foi desenvolvida.

Abstract

This work presents an architecture for a FPGA based implementation of a Multilayer Perceptron Artificial Neural Network (ANN). The proposed architecture aims to support the implementation of large ANNs in FPGA concerning area reduction, interconnection resources and area/performance trade-off. The proposed architecture uses $\log_2 m$ adders for an ANN with m inputs. An ANN whose topology is 256-10-10 could reach a speed-up of 36 times compared to a conventional software implementation in C language running in a PC. A software tool to generate the HDL level code of the ANN has been developed as well. Results and comparisons are also presented.

Sumário

RESUMO	II
ABSTRACT.....	III
ÍNDICE DE ILUSTRAÇÕES.....	VI
ÍNDICE DE TABELAS	VIII
1. INTRODUÇÃO	1
1.1 ESTRUTURA DA DISSERTAÇÃO.....	5
2. REDES NEURAIS ARTIFICIAIS MLP	7
2.1 NEURÔNIO ARTIFICIAL	8
2.2 APRENDIZADO DAS RNAs.....	11
2.3 O ALGORITMO BACK-PROPAGATION	15
2.4 SUPORTE AO PROJETO DE RNAs MLP	15
3. FPGA E SUA UTILIZAÇÃO EM SISTEMAS DE ALTO DESEMPENHO (ROCHA, 2010).....	19
3.1 FPGA.....	19
3.2 COMPUTAÇÃO DE ALTO DESEMPENHO (HPC).....	23
3.3 UTILIZAÇÃO DE FPGAS EM SISTEMAS HPC.....	26
4. ESTADO DA ARTE	31
4.1 IMPLEMENTAÇÕES DE RNAs EM FPGA	31
4.1.1 <i>FPGA implementation of a face detector using neural networks (Lee, et al., 2006)</i>	31
4.1.2 <i>FPGA implementation of Feed-Forward Neural Networks for smart devices development (Oniga, et al., 2009)</i>	33
4.1.3 <i>Field Programmable Gate Array (FPGA) based neural network implementation of Motion Control and fault diagnosis of induction motor drive (Soares, et al., 2006)</i>	35
4.1.4 <i>Análise Comparativa</i>	38
4.2 GERAÇÃO AUTOMÁTICA DE RNAs EM LINGUAGEM HDL	40
4.2.1 <i>Automatic synthesizable VHLD code generation from neural networks models using Matlab (Gugala, et al., 2009)</i>	40
4.2.2 <i>VANNGen- a Flexible CAD Tool for Hardware Implementation of Artificial Neural Networks (Braga, et al., 2005)</i>	42
4.2.3 <i>Análise Comparativa</i>	44
5. SOLUÇÃO PROPOSTA	47
5.1 COMPUTAÇÃO DO ESTADO DE ATIVAÇÃO	48
5.2 APROXIMAÇÃO DA FUNÇÃO DE ATIVAÇÃO	58
5.3 DESEMPENHO ESPERADO	70
6. IMPLEMENTAÇÃO	73
6.1 PONTO FLUTUANTE VS PONTO FIXO	73
6.2 MÓDULOS DE HARDWARE	76

6.2.1	<i>Mux_bias</i>	77
6.2.2	<i>Shift Register</i>	78
6.2.3	<i>Interface_Camadas</i>	79
6.2.4	<i>Weight_SR</i>	82
6.2.5	<i>Lookup_table</i>	83
6.3	GERAÇÃO AUTOMÁTICA DA RNA EM LINGUAGEM HDL.....	84
7.	ANÁLISE DE RESULTADOS	89
7.1	ÁREA.....	93
7.2	DESEMPENHO.....	94
7.3	PRECISÃO.....	95
8.	CONCLUSÕES E TRABALHOS FUTUROS.....	99
9.	APÊNDICE.....	101
9.1	APROXIMAÇÕES DA SIGMÓIDE	101
10.	REFERÊNCIAS	103
	ASSINATURAS.....	106

Índice de Ilustrações

Figura 1.1 Arquitetura SOC de uma máquina fotográfica digital	3
Figura 2.1 Modelo de um neurônio artificial	9
Figura 2.2 Exemplo de problema não linearmente separável	12
Figura 2.3 Resolvendo problemas não linearmente separáveis com redes MLP	13
Figura 2.4 Topologia de uma RNA MLP	14
Figura 2.5 Fase do projeto das RNAs.....	16
Figura 3.1 Estrutura do FPGA.	20
Figura 3.2 Estrutura do CLB.....	21
Figura 3.3 Estrutura de uma Switch Matrix.	22
Figura 3.4 Gap Tecnológico.....	24
Figura 4.1 exemplo de um sistema de detecção de faces.....	32
Figura 4.2 Diagrama de blocos do multiplicador-acumulador.....	33
Figura 4.3 Esquema do neurônio.....	34
Figura 4.4 Esquema de uma camada com 7 neurônios	35
Figura 4.5 sistema de detecção de falhas em motores.....	36
Figura 4.6 arquitetura do sistema de detecção de falhas com 7 RNAs	37
Figura 4.7 arquitetura do neurônio	38
Figura 4.8 estrutura da rede gerada.....	41
Figura 4.9 fluxo de dados do neurônio artificial.....	41
Figura 4.10 arquitetura do neurônio	42
Figura 4.11 sistema do VANNGen	43
Figura 5.1 fluxo de projeto da RNA com o gerador automático.....	48
Figura 5.2 Estrutura de interconexão das camadas	50
Figura 5.3 Exemplo do pipeline até a primeira soma em uma camada com 4 neurônios e 4 entradas	52
Figura 5.4 caminho de dados 4 neuronios 4 entradas	53
Figura 5.5 caminho de dados para camada com 4 neurônios e 5 entradas	54
Figura 5.6 caminho de dados para camada com 21 entradas	56
Figura 5.7 evolução de m comparado com $\log_2(m)+1$	58

Figura 5.8 Aproximação linear por partes otimizada	60
Figura 5.9 Aproximação linear otimizada (esquerda).....	61
Figura 5.10 Aproximação linear otimizada vs sigmoide.....	61
Figura 5.11 Aproximação PLAN (esquerda)	63
Figura 5.12 PLAN vs sigmoide.....	63
Figura 5.13 Aproximação de 2 ^a ordem (esquerda)	64
Figura 5.14 Aproximação de 2 ^a ordem vs sigmoide.....	65
Figura 5.15 arquitetura da lookup table com complexidade de acesso O(1)	68
Figura 5.16 Arquitetura modular da RNA completa.....	68
Figura 5.17 Taxa de leitura de dados de entrada de uma camada com 4 neuronios	69
Figura 6.1 Representação Ponto Flutuante precisão simples.....	73
Figura 6.2 Formato representação ponto fixo	74
Figura 6.3 Blocos dos componentes aritméticos de ponto flutuante	76
Figura 6.4 Mux bias	77
Figura 6.5 Máquina de estados do módulo mux bias.....	78
Figura 6.6 Shift-register	79
Figura 6.7 Interface Camadas.....	80
Figura 6.8 FSMs do módulo interface_camadas.....	81
Figura 6.9 Weight-SR	82
Figura 6.10 Lookup table	83
Figura 6.11 Algoritmo da geração do top level da RNA	85
Figura 6.12 Algoritmo de geração das camadas	88
Figura 7.1 Gráfico da função transcendental y.....	90
Figura 7.2 Curva ROC para a rede Íris	91
Figura 7.3 Curva ROC para a rede Semeion.....	92
Figura 7.4 Gráfico do erro das saídas da rede que aproxima a senóide	93
Figura 7.5 Comparativo entre as curvas ROC das rede Iris em SW e em HW(direita)	97
Figura 7.6 Comparativo entre as curvas ROC das rede Semeion em SW e em HW(direita)...	97
Figura 7.7 Gráfico da função senóide comparados com a implementação em SW(acima) e em HW	98

Índice de Tabelas

Tabela 3.1 Supercomputadores reconfiguráveis baseados em FPGA	29
Tabela 4.1 Quadro Comparativo dos trabalhos de implementação	40
Tabela 4.2 Quadro comparativo dos trabalhos relacionados de geração automática.....	44
Tabela 5.1 Aproximação PLAN	62
Tabela 5.2 Quadro comparativo das aproximações.....	65
Tabela 5.3 exemplo de aplicação de acesso a lookup table.....	67
Tabela 6.1 Comparativo ponto flutuante vs ponto fixo.....	74
Tabela 7.1 Características das três redes implementadas.....	90
Tabela 7.2 Utilização dos Recursos de Hardware	94
Tabela 7.3 Comparativo de desempenho HW - SW	95
Tabela 7.4 Erros das redes implementadas.....	96

1. Introdução

Computadores de propósito geral podem processar milhões de operações de ponto flutuante por segundo, no entanto, não são capazes de distinguir formas visuais ou categorizar objetos de tipos semelhantes.

Sistemas de computação sequencial têm bastante sucesso em resolver problemas matemáticos e científicos; criar, manipular e manter bancos de dados; têm sucesso em comunicações eletrônicas; no processamento de texto e gráfico; mas eles são incapazes de interpretar o mundo. Essa dificuldade típica em sistemas baseados na arquitetura sequencial de Von Neumann impulsionou gerações de pesquisadores a focarem no desenvolvimento de novos sistemas de processamento como as Redes Neurais Artificiais (RNAs) que solucionam problemas, como os acima citados, do mesmo jeito que o cérebro humano faz.

Este órgão biológico tem várias características altamente desejáveis em qualquer sistema de processamento digital: é robusto e apresenta tolerância a falhas, neurônios morrem todos os dias sem afetar sua funcionalidade global; é flexível, uma vez que se ajusta a novos ambientes através da aprendizagem; pode gerenciar informações difusas (inconsistências ou ruído); é altamente paralelo e, consequentemente, eficiente (eficiente em tempo de processamento); é pequeno, compacto e consume pouca energia.

As RNAs emulam as redes neurais biológicas que não requerem programação, mas generalizam e aprendem com a experiência. RNAs atuais são compostas por um conjunto de elementos de processamento muito simples que emulam os neurônios biológicos e suas conexões. Estes componentes não executam instruções, respondem em paralelo as entradas apresentadas.

É, por conseguinte, um sistema tolerante a falhas e a certo nível de ruído capaz de aprender e modificar os valores associados das conexões dos nós para ajustar o valor de suas saídas. O conhecimento e poder das RNAs não reside em instruções, mas na topologia (posições e conexões dos elementos de processamento), nos

valores das conexões (pesos) e nas funções que definem os elementos e o mecanismo de aprendizagem.

As RNAs oferecem uma alternativa a computação clássica de problemas do mundo real que usa o conhecimento natural (que pode ser incerto, impreciso, inconsistente e incompleto) e para qual o desenvolvimento de um programa de computador, que cobre todas as possibilidades e eventualidades, é impensável ou pelo menos muito trabalhoso e caro. Existem vários exemplos de aplicações de sucesso que usam RNAs tais como: processamento de imagem e voz, reconhecimento de padrões, interfaces adaptativas para sistemas homem/máquina, predição de séries temporais, controle não linear e otimização, filtragem de sinais, visão de robôs e etc (Rabuñal, et al., 2006).

Em muitos desses sistemas que utilizam RNAs o volume de dados a ser processado é enorme e/ou o tempo de resposta necessário é pequeno e crítico. Nesses casos, o não cumprimento dos requisitos não funcionais inviabiliza o uso da computação sequencial clássica.

Esse cenário fica ainda mais crítico e real quando aplicações embarcadas (portáteis ou não) necessitam implementar sistemas com RNAs, aí a capacidade de processamento necessária para suprir a taxa de computação demandada deve elevar o custo do sistema e seu consumo de potência (crítico para aplicações embarcadas). Ou seja, apenas o processamento da RNA dentro de uma gama de funcionalidades de um sistema demanda um aumento global na capacidade de processamento e, por conseguinte, uma mudança no projeto e um aumento de custo e potência.

O uso de hardware dedicado, neste caso, pode diminuir o custo final e o esforço do projeto (custo NRE), semelhante como é feito nas aplicações de processamento de vídeo embarcadas. Numa câmera fotográfica, por exemplo, a CPU não precisa ser capaz de processar os dados de entrada (análogicos), nem realizar compressão/descompressão de imagem e vídeo. Veja na Figura 1.1 a arquitetura SOC (System-On-a-Chip) de uma máquina fotográfica digital, com os coprocessadores de imagens (JPEG) e de vídeo.

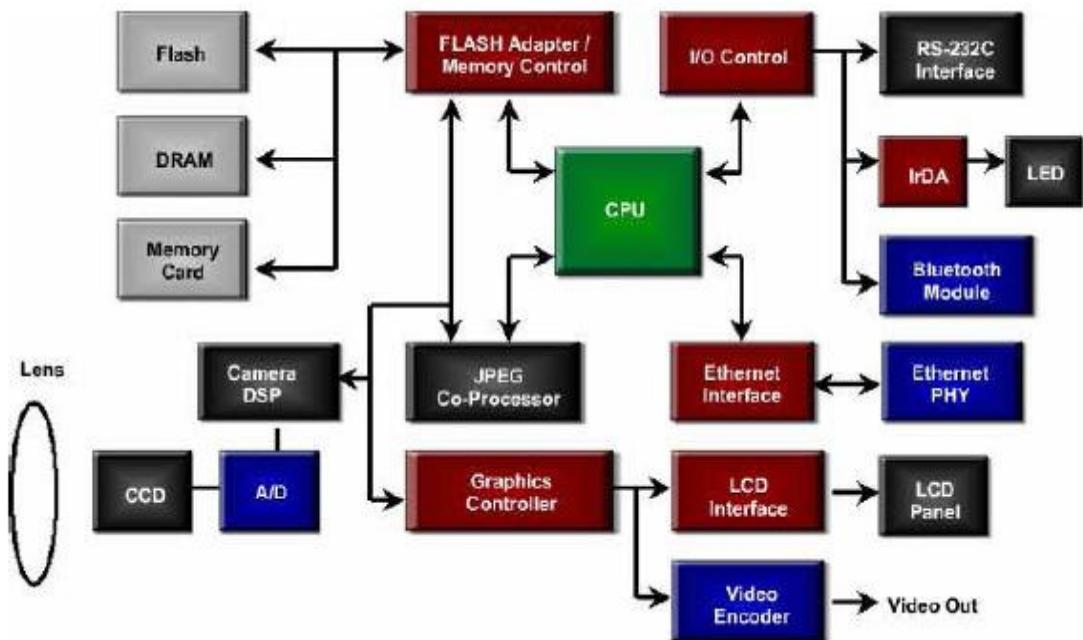


Figura 1.1 Arquitetura SOC de uma máquina fotográfica digital

De forma similar, um módulo de hardware dedicado que implementa uma RNA pode ser acrescido numa plataforma SOC sem grande esforço. Esse incremento de funcionalidade pode ser o diferencial, visto que traz para ele toda a vasta gama de aplicações em que se podem utilizar redes neurais.

Proposições de implementações de RNAs mais rápidas podem ser encontradas na literatura e vão desde implementações em hardware digital (Omondi, et al., 2006), (Girau, 2006), (Canas, et al., 2006), (Girones, et al., 2006), passando por GPUs (Graphics Processin Units), até sistemas analógicos. O que motiva esse grande número de implementações das RNAs é sua capacidade de ter paralelismo intrínseco de dados e de controle.

Na literatura os trabalhos de RNAs em implementados em dispositivos lógicos reconfiguráveis como os FPGAs (Field Programmable Gate Array) se dividem em implementações diretas da RNA para uma aplicação específica, implementações genéricas e até geração automática de código em uma linguagem de descrição de hardware (HDL).

Dos trabalhos de implementações diretas de RNAs em FPGA destacam-se os trabalhos de (Lee, et al., 2006) que desenvolve um detector de faces usando redes

neurais e integra o sistema em um FPGA. Fica evidente, neste trabalho, a aplicação de RNAs em aplicações que demandam alta capacidade de processamento e tempo crítico de resposta.

Em (Oniga, et al., 2009) é proposto o desenvolvimento de redes neurais em FPGAs para uso em dispositivos inteligentes. Eles usam os componentes disponíveis na biblioteca da fabricante de FPGAs Xilinx para construir os blocos aritméticos que realizam as computações da RNA.

Por último, em (Soares, et al., 2006), integra-se uma técnica de controle inteligente para controle de direção de alto desempenho. Segundo os autores, normalmente os dispositivos mais usados para desenvolver estratégias de controle em hardware são os DSPs (digital signal processors). Entretanto, sistemas de controle inteligentes baseados em redes neurais (utilizando FPGAs) permitiram um processamento paralelo, mais econômico que as implementações em ASICs, além de maior adaptabilidade e tolerância a falhas.

Esses trabalhos de implementação direta de RNAs em FPGA dão exemplos das inúmeras aplicações em que se podem obter ganhos ao se utilizar RNAs implementadas em uma arquitetura voltada para o processamento de alto desempenho e baixo tempo de resposta (latência).

Do grupo de trabalhos de geração automática de código em uma linguagem HDL, para implementações de RNAs em FPGA, o primeiro é de (Gugala, et al., 2009), que a partir da ferramenta CAD (Computer-Aided Design) *Matlab* desenvolveu um gerador de RNAs baseado na linguagem de *Matlab script*. Esta abordagem tem a desvantagem de apenas suportar apenas o Matlab como ferramenta CAD, além de seu gerador não ter sido desenvolvido numa linguagem de propósito geral.

Um outro trabalho de geração automática de (Braga, et al., 2005) apresenta o VANNGEN, uma ferramenta escrita em Java, que gera automaticamente RNAs em FPGAs.

A análise mais criteriosa desses trabalhos sugere que o desempenho das implementações está aquém do que se pode atingir com RNAs em FPGA, visto que

elas não exploram todo o paralelismo que se pode obter. O mesmo se pode dizer dos trabalhos de geração automática, cuja arquitetura da RNA em hardware ou é pouco eficiente, ou consome muita lógica.

Desta forma, propõe-se neste trabalho uma arquitetura de hardware digital visando o aumento da capacidade de processamento, compromisso área/desempenho, consumo de pinos de E/S e recursos de interconexão. Inicialmente é aplicada para sistemas que possuam FPGAs (Field-Programmable Gate Arrays) no intuito de conciliar a característica das RNAs de terem processamento paralelo e distribuído com a dos FPGAs de serem arquiteturas de processo paralelo. A arquitetura proposta explora o paralelismo temporal e espacial através de caminhos de dados implementados como pipelines em FPGA.

Comparando a arquitetura proposta com a dos trabalhos encontrados na literatura obtêm-se os seguintes diferenciais:

- Frequência máxima de operação do circuito bem maior.
- Possibilidade de escolha entre redução de área e aumento de desempenho.
- Pouco uso de recursos de interconexão do FPGA e baixo fan-out.
- Possibilidade de implementar RNAs sem limitações de número de entradas.
- Utilização de $\log_2 m$ somadores (para uma camada de neurônios com m entradas).
- Fácil modificação da função de ativação do neurônio.
- Baixo uso de memória e lógica.

1.1 Estrutura da dissertação

Os capítulos 2 e 3 apresentam as bases conceituais das RNAs e FPGAs respectivamente. No capítulo 4 descreve-se o estado da arte dividido em dois tipos: trabalhos de implementação de RNAs em FPGAs e trabalhos de geração automática de RNAs em FPGAs. Já no capítulo 5 a arquitetura proposta neste trabalho é apresentada em detalhes. O capítulo 6 demonstra a arquitetura com a apresentação

de exemplos de RNAs em FPGA. No 7 são analisados e comparados os resultados em termos de área, desempenho e precisão. Por fim o capítulo 8 apresenta as conclusões finais e propostas para futuros trabalhos.

2. Redes Neurais Artificiais MLP

O cérebro humano é composto por bilhões de células interconectadas numa rede gigantesca. Algumas tarefas simples que se faz no dia-a-dia, usando uma parte dos trilhões de conexões do cérebro, ainda são alvo de pesquisas que tentam reproduzir por meio de computador os resultados que se consegue sem mesmo se saber como. As redes neurais artificiais (RNAs) são modelos computacionais que se espelham no arranjo e na arquitetura dos cérebros dos animais. Inspirado no modelo biológico, foi definido para as RNAs um modelo de neurônio artificial e as conexões que simulam as sinapses. Esse ramo da computação teve seu primeiro apogeu com a criação das redes Perceptron (Braga , et al., 2007). Após vários desafios de identidade, as redes neurais artificiais, hoje, têm sido reconhecidas como um importante segmento de pesquisa, produzindo, muitas vezes, resultados muito mais satisfatórios do que a computação algorítmica tradicional.

O processamento do cérebro é inherentemente paralelo e distribuído, as informações são guardadas nas conexões de neurônios e se aprende através de exemplos e repetição. Tudo isso realizado por unidades simples, porém numerosas que isoladamente não fazem diferença para o todo. Mas é bem verdade que se cria e também se usa conhecimento prévio para adaptar-se a situações nunca antes passadas. Usando o paralelo biológico, nas RNAs um grande número de neurônios trabalha de forma conexa para processar a informação e fornecer um resultado.

As RNAs tem sido usadas em inúmeras áreas do conhecimento, como processamento de sinais, análise de imagens médicas, sistemas de diagnóstico e previsões de séries temporais. Um problema tipicamente solucionável com RNAs deve ser descrito como um problema de reconhecimento de padrões ou de aproximação de uma função. Para os casos de reconhecimento de padrões, uma RNA deve classificar um padrão de entrada dentre os de saída, mesmo sem nunca ter visto o referido padrão.

As RNAs apresentam as seguintes características desejáveis (Braga , et al., 2007):

- a. Aprendem através de exemplos – vários exemplos são mostrados à rede e na medida em que se ajustam os pesos a rede vai aprendendo.
- b. Adaptabilidade – alguns modelos de RNAs podem aprender continuamente, sem que se precise uma nova rodada de treinamento.
- c. Capacidade de generalização – a rede após treinamento deve responder bem a exemplos nunca antes vistos por ela.
- d. Tolerância a falhas – se retirados poucos neurônios a rede ainda assim responde da mesma forma (conhecimento descentralizado). E mesmo que os dados de treinamento contenham um certo nível de ruído, a rede ainda responde corretamente.
- e. Implementação rápida – as únicas etapas de desenvolvimento são treinamento, validação e teste. Elas são repetidas até que uma configuração candidata responda satisfatoriamente ao conjunto de testes.

2.1 Neurônio artificial

O neurônio artificial é a unidade da arquitetura das redes neurais artificiais. Na estrutura do neurônio observa-se:

- a. Um conjunto de entradas que recebem os sinais de entrada do neurônio;
- b. Um conjunto de sinapses cujas intensidades são representadas por pesos associados a cada uma delas;
- c. Uma função de ativação que relaciona as entradas e suas sinapses ao limiar da função a fim de definir se o neurônio será ativado ou não.

A Figura 2.1 mostra o modelo de um neurônio artificial, extraída de (Omondi, et al., 2006), cada w_i representa os pesos associados com as entradas x_i e Φ a função de

ativação ou limiar.

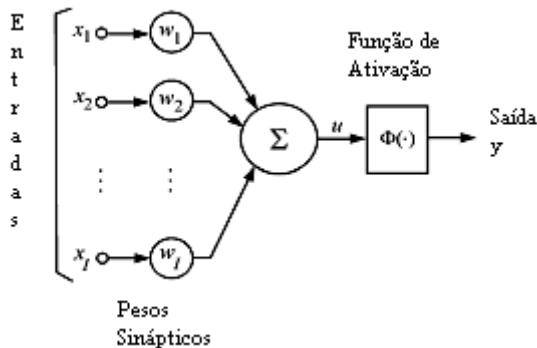


Figura 2.1 Modelo de um neurônio artificial

O resultado da sinapse, também chamado estado de ativação da rede, total de entrada é dado pelo produto interno do vetor de entrada pelo vetor de pesos. A saída da rede, y , é obtida pela aplicação de $\Phi(u)$. Cada neurônio computa a soma de produtos como segue na fórmula:

$$c = \sum_{j=1}^m w_j \cdot x_j$$

Assim, a saída do neurônio, y , corresponde à aplicação da soma de produtos na função de ativação (onde b corresponde ao **bias** ou limiar que é um peso associado a cada neurônio):

$$y = \Phi(u) = \Phi(c - b)$$

O vetor de pesos tem $m+1$ elementos correspondentes às m entradas do neurônio além do bias (b), que pode ser visto como mais um peso, cujo valor da entrada (x_{m+1}) correspondente tem valor fixo em -1 , resultando em:

$$u = \sum_{j=1}^{m+1} w_j \cdot x_j$$

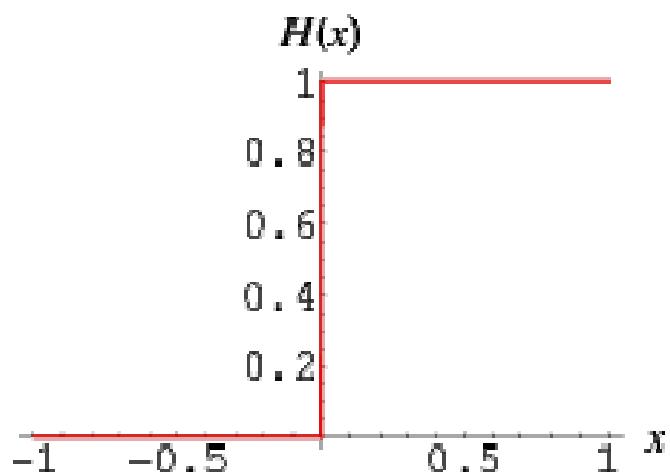
Assim, nas implementações de RNAs que seguem um modelo similar de neurônio precisa-se computar, paralelamente, um número de produtos internos que cresce exponencialmente com a quantidade de conexões na rede.

Depois de computada a sinapse total (u), esse resultado é aplicado na função de limiar que deve apresentar (para o algoritmo *backpropagation* apresentado na seção 2.3) algumas características particulares:

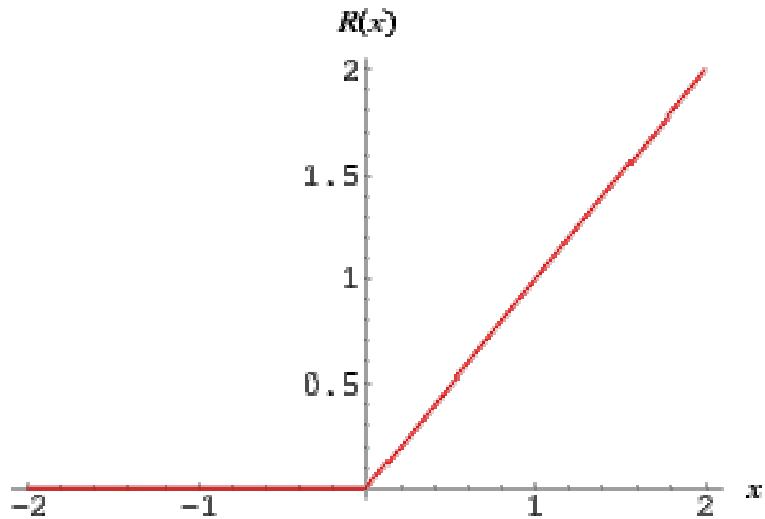
- a. Não linear;
- b. Diferenciável, contínua e, geralmente, não decrescente;
- c. Sigmoidal.

Algumas funções de limiar usadas são:

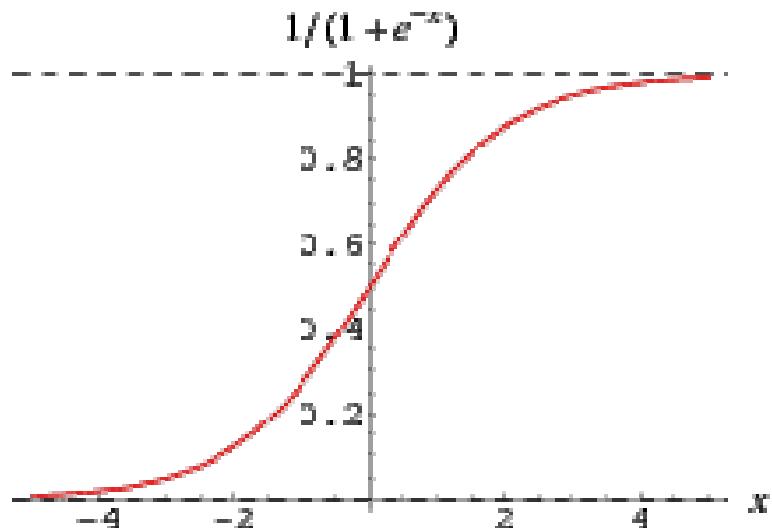
- a. Função degrau unitário: $\Phi(u) = 1$ se $u > 0$, $\Phi(u) = 0$, caso contrário



b. Função rampa unitária: $\phi(u) = \max\{0.0, \min\{1.0, u + 0.5\}\}$



c. Função sigmóide logística: $\phi(u) = a / \{ 1 + \exp(-bu) \}$, onde a e b representam, respectivamente, a amplitude e a inclinação da função.



2.2 Aprendizado das RNAs

Por aprendizado entende-se a capacidade de aprender a partir da interação com o ambiente e melhorar seu desempenho no decorrer do tempo. Os principais paradigmas de aprendizado são:

- a. **Aprendizado supervisionado** – nesse tipo de aprendizado se tem um prévio conhecimento sobre o ambiente, na forma de um conjunto de pares (perguntas, respostas) sobre os quais se deve aprender. Baseado nesse paradigma de aprendizado, a RNA pode ser treinada, usando uma etapa de treinamento anterior à operação (offline) ou pode-se aprender quando a rede estiver operando (online).
- b. **Aprendizado por reforço** – visualiza-se nesse paradigma a figura de um crítico (sinal de realimentação externo) que, ao contrário do aprendizado supervisionado, não detém as respostas a priori, mas avalia a decisão da rede como “boa” ou “ruim”. A rede usa essa informação para maximizar a satisfação do crítico.
- c. **Aprendizado não supervisionado** – neste aprendizado, não se possui informações prévias nem o papel do crítico. Uma rede neural, que segue o aprendizado não supervisionado, classifica automaticamente os dados, extraindo estatisticamente suas características mais relevantes.

O paradigma de aprendizado mais amplamente utilizado pelas RNAs (e mais especificamente pelas RNAs MLP) é o aprendizado supervisionado.

Em RNAs o procedimento usual na solução de problemas passa inicialmente por uma fase de aprendizagem, em que um conjunto de exemplos é apresentado à rede e a partir daí são geradas respostas para o problema segundo sua capacidade de generalização.

A generalização está associada à capacidade de a rede aprender através de um conjunto reduzido de exemplos e posteriormente dar respostas coerentes para dados não conhecidos.

Sabe-se que as redes de apenas uma camada (apenas a camada de saída) resolvem apenas problemas linearmente separáveis. Veja no exemplo da Figura 2.2 que em um problema que não é linearmente separável não se pode separar as elipses dos retângulos traçando uma única reta.

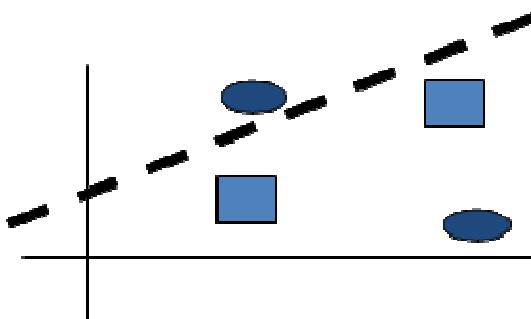


Figura 2.2 Exemplo de problema não linearmente separável

O que cada camada da RNA faz é traçar retas no espaço do conjunto de entradas. Assim, a solução de problemas não linearmente separáveis passa pelo uso de redes com uma ou mais camadas intermediárias. Uma rede com uma camada intermediária pode aproximar qualquer função contínua. Veja na Figura 2.3 que com uma RNA que tenha duas camadas intermediárias é possível solucionar o problema de separar as elipses dos retângulos.

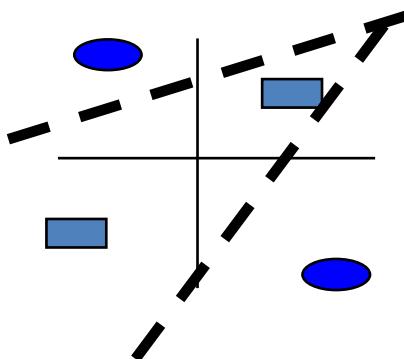


Figura 2.3 Resolvendo problemas não linearmente separáveis com

A precisão obtida da implementação da função objetivo depende do número de neurônios utilizados nas camadas intermediárias. Esse número é em geral definido empiricamente, e depende fortemente da distribuição dos padrões de treinamento e de validação da rede. A Figura 2.4 mostra os elementos da topologia de uma RNA com uma camada intermediária. A camada de entrada não realiza nenhum processamento, apenas repassa os dados de entrada para a primeira camada escondida das redes MLP.

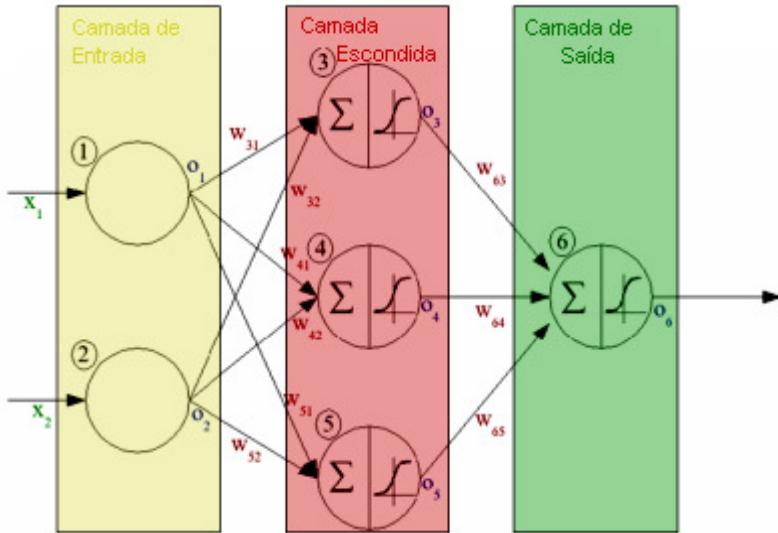


Figura 2.4 Topologia de uma RNA MLP

O número adequado de neurônios na camada intermediária depende de vários fatores, como:

- Número de exemplos de treinamento – pequenas bases de dados de treinamento trazem problemas na fase de aprendizado.
- Complexidade da função a ser aprendida – uma rede com poucos neurônios na camada intermediária em geral não consegue bom desempenho para problemas complexos de se resolver.

Deve-se ter cuidado para não utilizar nem unidades demais, o que pode levar a rede a memorizar os padrões de treinamento e detalhes do ruído, em vez de extrair as características gerais garantindo a generalização (o chamado *overfitting*), nem um número muito pequeno, o que pode, por sua vez, aumentar o tempo para se encontrar a representação ótima, além de esta rede poder ser pouco complexa para aproximar a função desejada (o *underfitting*).

2.3 O algoritmo Back-Propagation

O algoritmo de aprendizado mais conhecido para treinamento das RNAs é o back-propagation. O back-propagation é um algoritmo supervisionado que utiliza pares (entrada, saída desejada) para ajustar os pesos da rede. O treinamento ocorre em duas fases, sendo que em cada fase a rede é percorrida em um sentido (direto e reverso). Estas duas fases são chamadas de *forward* (sentido direto) e *backward* (sentido reverso).

Na fase forward a entrada é utilizada para computar, ao longo das camadas, as saídas produzidas pelos neurônios da camada de saída. Essas saídas são comparadas às saídas desejadas e o erro para cada neurônio da camada de saída é calculado.

Na fase de backward, em cada neurônio é feito um ajuste do peso de forma a reduzir o erro. Isso é feito assumindo uma taxa de aprendizagem previamente definida que pode ser única para todas as camadas, variável com o decorrer do tempo ou crescente à medida que a camada se afasta da saída. Os erros dos neurônios das camadas mais internas correspondem a uma ponderação do erro das camadas anteriores na fase de backward.

O algoritmo do back-propagation procura minimizar o erro obtido pela rede ajustando os pesos para que eles correspondam a pontos de mínimo na superfície de erro. Este ajuste é feito através de um método de gradiente descendente. O back-propagation é um bom método para minimização local do erro.

Para superfícies de erro simples este método certamente encontra a solução com um erro mínimo. Já para superfícies mais complexas, o algoritmo pode levar a convergir para mínimos locais.

2.4 Suporte ao projeto de RNAs MLP

O projeto de RNAs Multilayer Perceptron (MLP) normalmente é assistido por uma

ferramenta de software que auxilia nas diversas fases de desenvolvimento. A definição dos parâmetros da rede em si define o projeto da RNA. Nessa fase são determinados: a topologia da rede, os pesos dos neurônios, as funções de pré-processamento e pós-processamento de dados, bem como a função de ativação.

As ferramentas CAD mais usadas no projeto de RNAs são: Matlab (Neural Networks Toolbox), Scilab¹, Tanagra² e Weka³. Um projetista de sistemas de computação inteligente se vale do seu conhecimento do problema e de RNAs para desenvolver a rede que pode ser validada com as ferramentas CAD de IA. Depois dessa fase a rede pode ser implementada em software usando as linguagens de programação de propósito geral (C/C++, java, C#, etc). Nesta segunda fase as atividades se concentram no desenvolvedor de software que implementa a solução final. Esse fluxo de projeto pode ser visualizado na Figura 2.5.

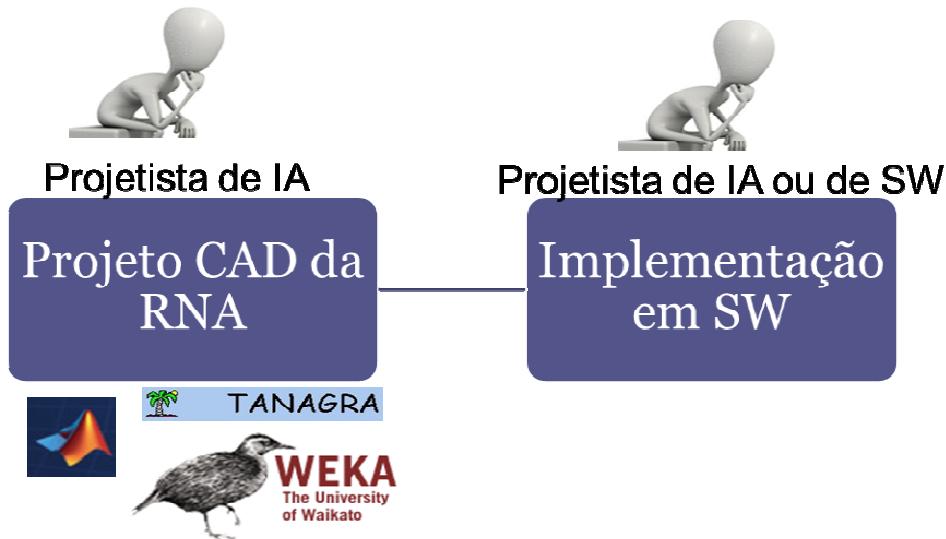


Figura 2.5 Fase do projeto das RNAs

Para ajudar na fase de implementação das RNAs em software, existem algumas bibliotecas como a FANN (Fast Artificial Neural Network Library) (FANN, 2011). Estas bibliotecas ajudam a diminuir o tempo de projeto para as implementações em

¹ <http://www.scilab.org/>

² <http://eric.univ-lyon2.fr/~ricco/tanagra/en/tanagra.html>

³ <http://www.cs.waikato.ac.nz/ml/weka/>

software, visando apenas processadores de propósito geral.

3. Dispositivos lógicos reconfiguráveis – FPGAs (Rocha, 2010)

3.1 FPGA

Recentemente, a computação reconfigurável vem surgindo como uma possível opção para preencher gaps entre implementações de sistemas em hardware e software, proporcionando soluções de melhor desempenho que soluções em software e garantindo uma maior flexibilidade que implementações em hardware. A partir dos dispositivos lógicos reconfiguráveis, incluindo os *FPGAs*, esta tecnologia tem conseguido excelentes resultados em diversas aplicações (Gokhale, et al., 2005).

O *FPGA* foi criado pela empresa Xilinx, e teve o seu lançamento no ano de 1985 como um dispositivo passível de ser configurado por *software*, ou seja, ele poderia ser programado de acordo com as aplicações dos usuários.

Estes dispositivos são formados, em geral, por um *array* bi-dimensional de circuitos computacionais cuja funcionalidade pode ser determinada por um *stream* programável de *bits*. Esses circuitos computacionais, conhecidos como blocos lógicos, podem ser conectados através de matrizes de conexão, que também são programáveis. Assim, circuitos digitais grandes ou pequenos podem ser facilmente mapeados em um único dispositivo reconfigurável através da programação lógica das funções do circuito em seus blocos lógicos e das matrizes de interconexão.

Devido a esta flexibilidade no desenvolvimento de sistemas digitais, na simplicidade de realizar alterações lógicas, desempenho, muitas vezes iguais ao do produto final, redução do consumo de potência e proteção da propriedade intelectual, os *FPGAs* têm sido amplamente utilizados no processamento de informações digitais, principalmente na prototipação rápida de circuitos digitais. A Figura 3.1 apresenta a estrutura interna de um FPGA convencional.

Um *FPGA* é constituído basicamente de um vetor bidimensional de blocos lógicos (*CLB, configuration logical blocks*), com trilhas verticais e horizontais e matrizes de interconexão (*Switch Matrix*), localizadas entre as linhas e colunas dos *CLB's*. No perímetro do dispositivo estão localizados blocos de entrada e saída (*IOB, input output block*) usados para conectar o chip a outros componentes do sistema.

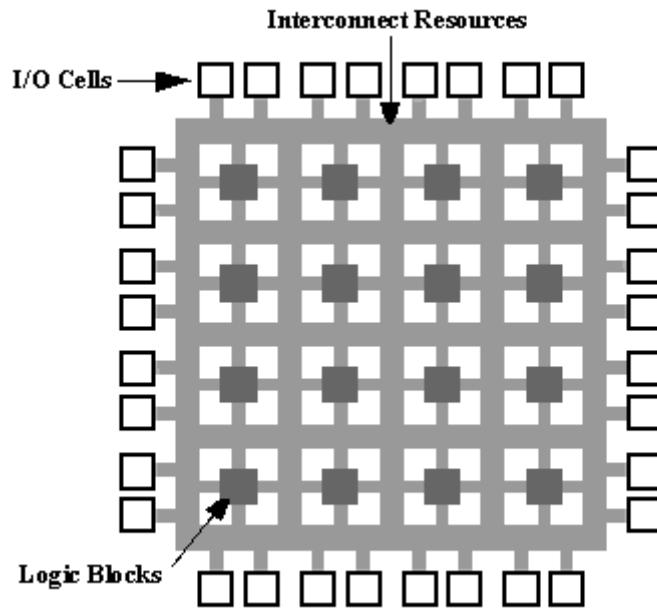


Figura 3.1 Estrutura do FPGA.

Os *CLB's* são, em geral, formados por um conjunto de *flip-flops* (2 a 4), e blocos lógicos de 4 ou mais variáveis, que podem implementar diferentes lógicas combinacionais. Os circuitos combinacionais apresentam as saídas únicas e exclusivas em função das variáveis de entrada. A Figura 3.2 representa a estrutura interna de um *CLB*.

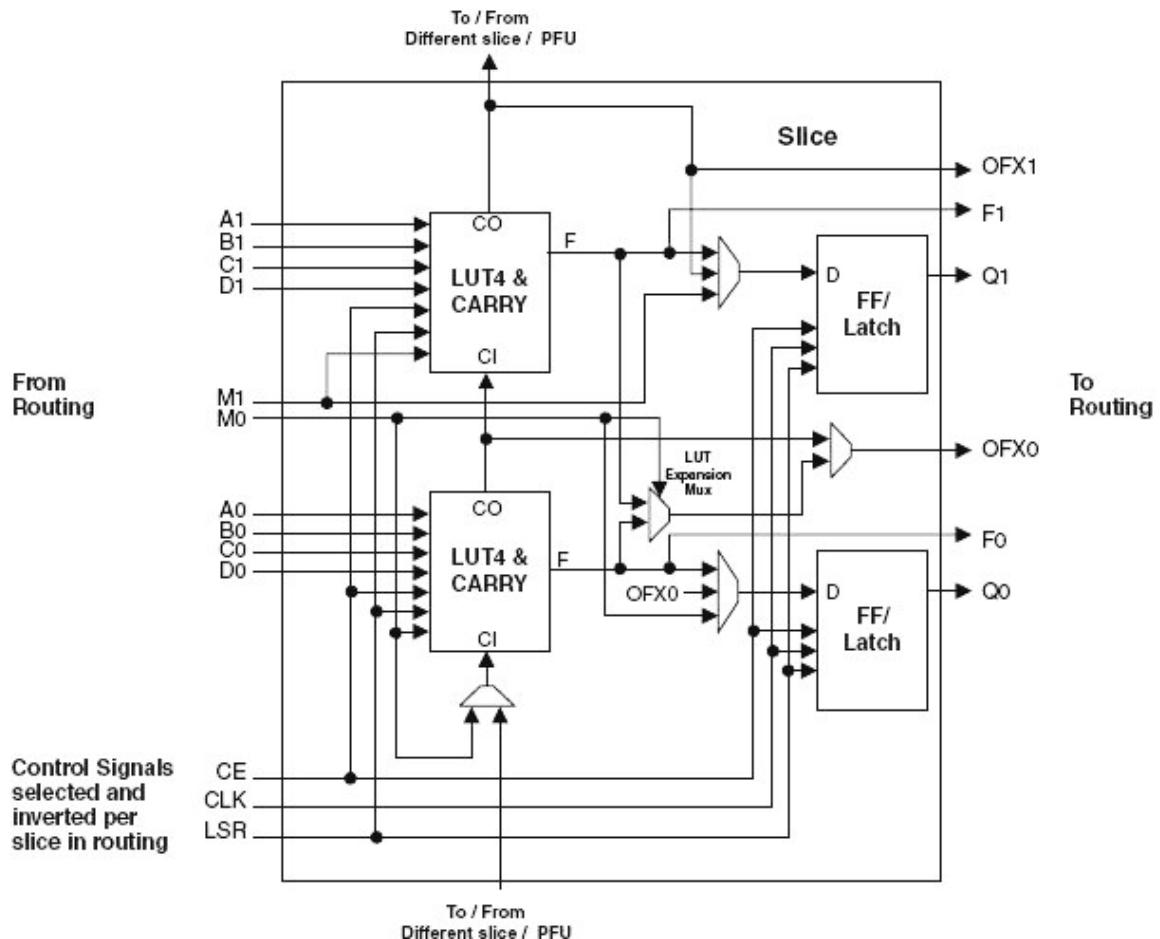


Figura 3.2 Estrutura do CLB.

No interior de cada bloco lógico do *FPGA*, existem vários modos possíveis para implementação de funções lógicas, em função do tipo de bloco lógico disponível para sua implementação (portas AND, OR, Mux, Look-up-tables). Um dos mais utilizados por fabricantes de *FPGA* são as *LUTs* (*Look-Up Tables*). Esse tipo de bloco lógico contém células de armazenamento que são utilizadas para implementar pequenas funções lógicas e onde cada célula é capaz de armazenar um único valor lógico: zero ou um. Cada *LUT* é essencialmente uma memória pré-programada que fornece uma saída dado um conjunto de variáveis de entrada. A *LUT* representa uma tabela verdade da função lógica que foi programada. As *LUTs* possuem geralmente quatro ou cinco entradas, o que permite endereçar de 16 a 32 células. Assim, quando um circuito lógico é implementado em um *FPGA*, os blocos lógicos são programados para realizar as funções necessárias, e os canais de roteamento são estruturados de

forma a realizar a interconexão necessária entre estes blocos lógicos.

As células de armazenamento dos *LUTs* de um *FPGA* são voláteis, o que implica na perda do conteúdo armazenado, no caso de falta de suprimento de energia elétrica. Dessa forma, o *FPGA* deve ser programado toda vez que for energizado. Geralmente, utiliza-se uma memória *FLASH EEPROM (Electrically Erasable Programmable Read Only Memory)*, cuja função é carregar automaticamente as células de armazenamento, toda vez que o *FPGA* for energizado.

As matrizes de chaveamento (*Switch Matrix*) apresentadas na Figura 3.3 são matrizes que permitem as interconexões entre os *CLB's*. Em cada matriz $n \times n$, poderá haver $m \times n$ pontos de interconexão (onde m é menor ou igual a n), os quais podem ser utilizados na confecção de ligações horizontais e verticais durante a criação do circuito. A partir de uma determinada programação das ligações dos *CLB's* a funcionalidade desejada de um circuito digital pode ser implementado.

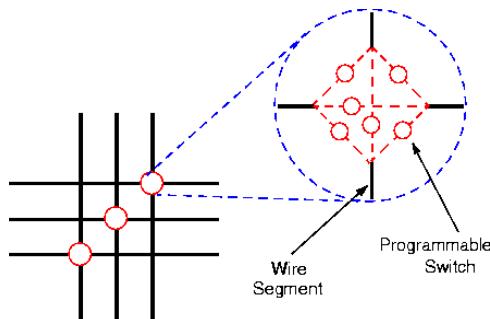


Figura 3.3 Estrutura de uma Switch Matrix.

Os blocos de I/O (*IOB's*), como dito acima, são os componentes do *FPGA* responsáveis pelo interfaceamento das saídas provenientes das combinações de *CLB's* com o mundo externo. Através dos *IOB's* o *FPGA* pode trocar informações com um ambiente externo. Estes blocos são basicamente compostos por *buffers*, que funcionarão como um pino bidirecional (entrada e saída) do *FPGA*.

3.2 Computação de Alto Desempenho (HPC)

Desde sua invenção quase quatro décadas atrás, o microprocessador teve grandes avanços no aumento do seu desempenho. O desempenho dos microprocessadores vem duplicando a cada 18 meses, característica que ficou conhecida como Lei de Moore (Altera, 2007). Como resultado, os processadores saíram de uma invenção que era quase um “brinquedo” dos pesquisadores para dispositivos extremamente velozes utilizados largamente em todo o mundo.

Cada vez que processadores mais eficientes eram lançados no mercado, novas aplicações, mais complexas, passaram a ser implementadas e demandadas, fazendo com que um próximo processador mais eficiente fosse sempre esperado com grande expectativa pelos desenvolvedores de software. Em quase toda a história dos processadores, aplicações sempre aumentavam suas complexidades para se equiparem com o desempenho dos processadores, permitindo, assim, que os processadores estivessem sempre à frente da demanda. Porém, aplicações de alto desempenho computacional estão demandando uma capacidade de processamento que um único processador não pode suportar de forma adequada, criando assim um *gap* tecnológico entre a demanda e o desempenho, como é mostrado na Figura 3.4.

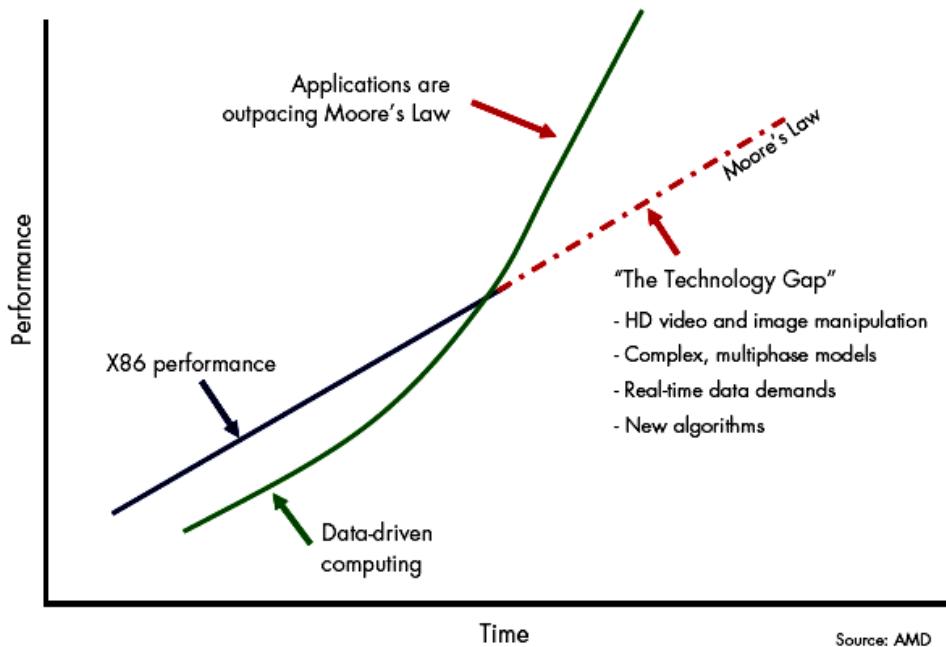


Figura 3.4 Gap Tecnológico.

O termo *HPC* foi originalmente usado para descrever os primeiros supercomputadores que surgiram para suprir a demanda por um maior poder de processamento. Porém, com o passar dos anos o termo *HPC* evoluiu para englobar um conceito maior. Agora o termo *HPC* inclui também sistemas com uma capacidade de computação maior, sistemas com uma grande taxa de transferência de dados, e sistemas com a habilidade de suportar computação distribuída.

As arquiteturas de sistemas *HPC* também evoluíram com o passar dos anos. Originalmente os sistemas *HPC* eram supercomputadores que utilizavam arquiteturas baseadas no paradigma *Massively Parallel Processing (MPP)* (Potter, 1985) e *Symmetric Multiprocessing (SMP)* (Hung, et al., 2005) (Huerta, et al., 2008). Tais sistemas requeriam processadores específicos, caminhos de dados proprietários, gerando assim sistemas excessivamente caros.

Atualmente, essas arquiteturas vêm perdendo popularidade e os sistemas que vêm sendo adotados são os chamados *clusters computing*, que são sistemas bem mais baratos e que possuem uma capacidade de processamento tão boa quanto os

mainframes e supercomputadores. Sistemas *cluster computing* estão entre os 10 sistemas mais rápidos do mundo, segundo o top500 (top11).

Sistemas baseados em *clusters* são formados por vários computadores, conectados em rede, trabalhando juntos para resolver uma determinada tarefa. Devido a sua interconexão, esses computadores podem ser visto como um supercomputador. Cada um dos computadores desses sistemas são processadores de propósito geral. A interconexão entre os vários computadores (nós) suporta grande largura de banda e baixa latência, como a *Infiniband*⁴ e *MyriNet*⁵. O sistema operacional de tais sistemas é geralmente o *Linux* devido à grande variedade de software *open-source* disponíveis, diminuindo o custo dos mesmos.

Devido às vantagens associadas ao baixo preço pelo uso de máquinas (*PCs*) comerciais, pela utilização de sistemas operacionais *open-source*, e pela facilidade de criação de ambientes de desenvolvimento de software, entre outras características, é esperado que a utilização de *clusters*, principalmente deste tipo, continue crescendo. É esperado que não apenas sistemas do tipo *clusters* continuem crescendo, mas todos os sistemas *HPC*, já que o número de aplicações que necessitam de tais sistemas vêm aumentando ao longo dos anos.

Aplicações *HPC* estão em todo lugar vão desde o modelamento de elementos finitos, bastante utilizado na indústria automobilística, na área de testes, evitando a destruição de milhares de automóveis, passando por dinâmica de fluidos, processamento de imagem, aplicações em sísmica, modelagem de moléculas, aplicações financeiras, etc. Todas essas aplicações têm características em comum, todas apresentam problemas de alta complexidade, possuem um grande conjunto de dados e levam muito tempo para serem computadas.

⁴ www.infinibandta.org/

⁵ www.myri.com/myrinet/overview/

3.3 Utilização de FPGAs em sistemas HPC

Aplicações que necessitam de um alto poder de computação e de uma grande quantidade de dados vêm ganhando grande enfoque nos últimos anos com o rápido aumento de desempenho dos computadores de propósito geral e com o surgimento de sistemas de computação paralela. Apesar da frequência nominal das *CPU*s de propósito geral ser bem elevada, a real utilização de todo esse poder computacional é bem reduzida quando comparada com o desempenho de pico desejado nestas aplicações. A principal razão dessa ineficiência é a grande quantidade de transistores nessas *CPU*s, que são utilizados na implementação de sua lógica de controle na provisão de um fluxo de dados flexível. Este tipo de arquitetura não é adequada para implementação de vários problemas científicos ou de engenharia que possuem seus domínios computacionais em regiões geométricas 2D e 3D, tornando seu tempo de resolução impraticável devido ao crescente montante de dados a serem processados ou a necessidade de inúmeras simulações nessas regiões complexas.

Problemas em *HPC* podem ser representados também por diferentes tipos de estruturas computacionais e precisões numéricas. Em computação científica, por exemplo, as operações podem ser realizadas com números no formato ponto flutuante (com precisão dupla ou simples), adotando padrões como o IEEE 754. O desempenho neste caso pode ser medido em unidades de operações de ponto flutuante por segundo, como Megaflops ou Gigaflops.

Aplicações que requerem sistemas *HPC* geralmente necessitam de grandes quantidades de dados e de grande poder computacional. Essas aplicações geralmente possuem uma relação de operações, em ponto flutuante, por acesso à memória, relativamente baixa requerendo usualmente um espaço de memória considerável para armazenar dados intermediários. Este comportamento tende a gerar endereçamentos indiretos irregulares, resultando no baixo desempenho no uso de estruturas de cache em computadores de propósito geral, devido à complexa arquitetura do seu sistema de hierarquia de memória.

Existe, portanto, uma grande diferença entre o desempenho teórico de pico e a real velocidade de execução de um programa em uma *CPU* de propósito geral.

Implementações puramente em *software*, como por exemplo, um *cluster* de PC com alto grau de paralelismo, ou uma implementação com otimizações de utilização de memória ou disco, ou uma implementação com reordenação de instruções, buscam acelerar a execução de aplicações *HPC*. Entretanto, poucas são as aplicações, como multiplicações de matrizes densas ou a Transformada Rápida de Fourier, que conseguem alcançar de 80 a 90 por cento do desempenho de uma *CPU*. A maior parte das aplicações alcançam índices em torno de 20 a 30 por cento, com algumas aplicações realísticas chegando a 10 ou 5 por cento de desempenho (Underwood, et al., 2004).

Outra abordagem para computação de alto desempenho em aplicações específicas é a utilização de sistemas customizados. Neste caso, cada dispositivo, é construído para a execução de algoritmos ou aplicações específicas, obtendo assim um melhor desempenho, e menor requisito de energia, quando comparado com *CPU*s convencionais. Esses dispositivos são chamados de circuitos integrados de aplicações específicas (*Application-Specific Integrated Circuit – ASIC*). Do ponto de vista tecnológico, o desenvolvimento de um *ASIC* não é um problema, porém, na prática, tal desenvolvimento encontra várias barreiras, como por exemplo, o custo NRE (Non-Recurrent Engineering) bastante elevado. Um exemplo de sucesso de um sistema que utiliza um *ASIC* é o GRAPE (GRA11), projeto gerenciado por um grupo de cientistas da computação e astrofísicos na Universidade de Tokyo (Makino, et al., 1998). O projeto do GRAPE teve como objetivo a construção de uma infraestrutura que combinasse a alta velocidade de rede, sistemas computacionais de alto desempenho e servidores de armazenamento de grande capacidade. Foi aplicado no campo de pesquisas científicas, especialmente aplicações de dados distribuídos, computação de alto desempenho para simulações numéricas de grande escala e banco de dados distribuídos.

Quando a demanda por desempenho nas aplicações começou a aumentar, uma das alternativas desenvolvida foi a utilização de coprocessadores junto aos processadores para a realização de certas tarefas específicas. A utilização de tais coprocessadores começou com alguns processamentos especializados de E/S, como o de modems e controladores Ethernet, estendendo-se para o uso de aceleradores gráficos e sistema de criptografia (*encryption engines*). No auxilio à computação científica e processamento de sinais logo sugiram também os coprocessadores aritméticos, como as *Floating-point Units (FPUs)*, para lidar com operações de multiplicação e divisão em ponto flutuante. Unidades especiais para lidar com processamento digital de sinais, *Digital Signal Processors (DSPs)* também foram lançados como coprocessadores, desenvolvidos para lidar com algoritmos matemáticos mais complexos.

Observa-se, no entanto, que soluções como aquelas apresentadas no sistema GRAPE, ou o uso de coprocessadores como aqueles descritos anteriormente, não são resposta para o *gap* computacional ora existente, já que elas apenas resolvem apenas parte do problema.

Uma opção para este novo paradigma computacional seria uso de coprocessadores ou plataformas baseados em componentes de hardware, provendo três habilidades fundamentais, quais sejam: o coprocessador ou a plataforma deve prover uma aceleração de *hardware* específica para algumas funcionalidades da aplicação, independente de qual aplicação seja; o coprocessador ou a plataforma precisa oferecer a possibilidade de uso de *pipeline* e estruturas paralelas, para que eles possam continuar oferecendo um bom desempenho mesmo se as funcionalidades das aplicações aumentarem; e o coprocessador e/ou a plataforma precisa prover uma grande largura de banda e uma baixa latência nas interfaces que se ligam com o processador principal e com a memória.

Um dispositivo que é uma alternativa para que não se tenha que projetar um coprocessador ou uma plataforma para cada uma das varias aplicações possíveis, é o *FPGA*. Este dispositivo pode ser configurado de acordo com as necessidades da

aplicação do usuário. Assim, para cada aplicação nova, necessita-se de um projeto para mapear o problema na arquitetura do FPGA. Atualmente os *FPGAs* oferecem um desempenho excelente em aplicações intrinsecamente paralelas (paralelismo de dados e de controle), que incluem construtores do tipo *loops*. Eles podem suportar estruturas de *pipeline* com profundidades variáveis e provêm um alto potencial em computação paralela, permitindo que funções altamente complexas sejam executadas em até um ciclo de relógio. Podem ser utilizados para agilizar o tempo de prototipação do sistema em hardware, antes que se inicie o projeto de um *ASIC* que tem alto NRE. Caso o *FPGA* suporte reconfiguração dinâmica, ele pode ainda prover um coprocessamento altamente customizado (customização *on-the-fly*) para uma grande variedade de aplicações em um único *chip* (Berthelot, et al., 2006).

Aplicações de *HPC* começaram a utilizar dispositivos *FPGAs* a partir da década de 90. A Tabela 3.1 lista alguns institutos/companhias que projetaram e desenvolveram supercomputadores de alto desempenho baseados em *FPGAs* para aplicações específicas, com o objetivo de substituir os computadores de propósito geral.

Tabela 3.1 Supercomputadores reconfiguráveis baseados em FPGA

Systems	Manufacturer	Year	Number of FPGAs	Applications	Hardware Costs
BEE-2 [20]	UC Berkeley	2005	5 per Blade 200 per Rack	Radio astronomy	\$ 500K
Starbridge [21]	NASA	2005	11 per system	Aeronautics & astronautics	\$ 150K
DN8000K10 [22]	DINI Group	2005	16 per board	Digital circuit emulation	-

De fato, uma maneira interessante de se tirar proveito de arquiteturas concorrentes é utilizar-se do que há de melhor em cada uma delas, desenvolvendo plataformas híbridas. Por exemplo, adotando um modelo híbrido, associando-os a *CPUs* de propósito geral na plataforma PC. Essa arquitetura é interessante, pois ela suprime a ineficiência que os *FPGA* possuem na realização de tarefas, como operações de ramificação, condição e *loops* com tamanhos variáveis, etc., visto que os processadores de propósito geral já apresentam estruturas de previsão de desvios e

execução especulativa. Por outro lado, ela permite que operações aritméticas complexas possam ser resolvidas com alto desempenho em coprocessadores baseados em *FPGAs*. Assim pode-se construir através desse modelo híbrido *CPUs-FPGAs* aplicações altamente customizadas que se beneficiam com o melhor das duas tecnologias.

4. Estado da arte

Muitos pesquisadores têm desenvolvido implementações em hardware de RNAs usando diversas técnicas sejam elas digitais, analógicas e até óticas. Embora as redes analógicas apresentem a desvantagem de serem imprecisas e de baixa flexibilidade, por outro lado permitem um projeto de maior velocidade e de baixo custo. O principal problema das implementações digitais é o uso de grande quantidade de multiplicadores e a não linearidade da função de ativação do neurônio.

Neste capítulo são apresentados trabalhos da literatura divididos em trabalhos de implementação direta de RNAs em FPGA e em trabalhos de geração automática de RNAs em FPGA.

4.1 Implementações de RNAs em FPGA

4.1.1 FPGA implementation of a face detector using neural networks (Lee, et al., 2006)

Nesse trabalho, Yongsoon Lee e Seok-Bum Ko implementam um detector de faces usando rede neural MLP em FPGA. O esquema do sistema de detecção de faces é mostrado na Figura 4.1. Detalhes de cada componente não são dados no trabalho.

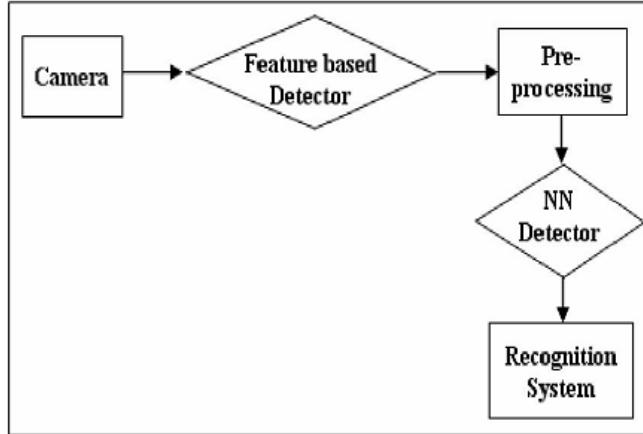


Figura 4.1 exemplo de um sistema de detecção de faces

Os autores utilizaram aritmética de ponto flutuante para representar os dados motivados pela característica de range dinâmico e de redução de bits da unidade aritmética (podendo assim, também, reduzir o consumo de memória utilizada). Unidades de multiplicador-acumulador (MAC) também acrescem uma maior precisão na computação do estado de ativação da rede. Na arquitetura proposta pelos autores, cada MAC é responsável pela computação do estado de ativação de cada camada.

A implementação da função de ativação foi feita usando a aproximação pelo polinômio de Taylor de 4^a ordem. A equação abaixo mostra o cálculo da aproximação. Observe que é necessário computar três produtos de x , a divisão de x^3 pelo fator 6 (uma operação de shift e uma divisão por 3), a divisão de x^4 pelo fator 24 (três shifts e uma divisão por 3), as 5 adições, além da divisão de k por $k + 1$. Logo essa aproximação consome grande quantidade de recursos para ser realizada.

$$\frac{1}{1 - e^{-SUM}} = \frac{1}{1 + 1/k} = \frac{k}{k+1},$$

, where $k = (1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24})$.

Uma rede de três camadas (25:6:2) leva 1,7ms para classificar um frame, rodando à frequência de 38MHz. O desempenho conseguido corresponde a um ganho de 38

vezes comparado a um Pentium 4 de 1.4GHz que computou um frame em 50ms. O ganho de desempenho desse sistema poderia ser usado para melhorar a resolução da imagem e assim aprimorar a taxa de acerto do sistema.

O que se evidencia nesse trabalho é a necessidade do uso de RNAs em aplicações de tempo real e por essa característica, essas aplicações necessitam de um desempenho superior do que se conseguiria com software.

4.1.2 FPGA implementation of Feed-Forward Neural Networks for smart devices development (Oniga, et al., 2009)

Neste trabalho os autores propõem o desenvolvimento de redes neurais em FPGA para uso em dispositivos inteligentes. Eles usam os componentes disponíveis na biblioteca de componentes da fabricante de FPGAs Xilinx para construir os blocos aritméticos que realizam as computações da RNA. A computação do estado de ativação é feita pelos módulos a seguir:

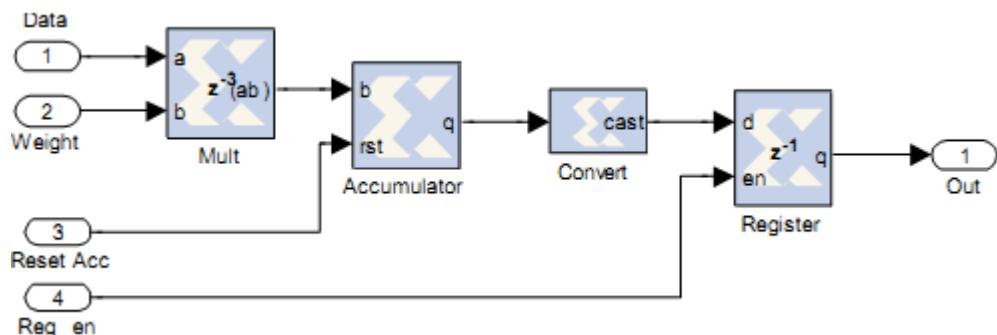


Figura 4.2 Diagrama de blocos do multiplicador-acumulador

Esses módulos integrados formam o multiplicador-acumulador (MAC) em que as entradas e pesos do neurônio vão sendo processadas. Na arquitetura proposta pelos autores, cada neurônio pode conter 1 ou mais MACs. Caso apenas 1 MAC seja usado, a soma de produtos das entradas pelos pesos é computada serialmente (parcela a parcela). No caso de vários MACs cada neurônio pode paralelizar a computação da soma de produtos com a penalidade do aumento da área utilizada. Todos os

componentes são módulos que implementam operações da aritmética de inteiros.

A lógica de controle agregada ao caminho de dados do neurônio proposto é mostrada na Figura 4.3:

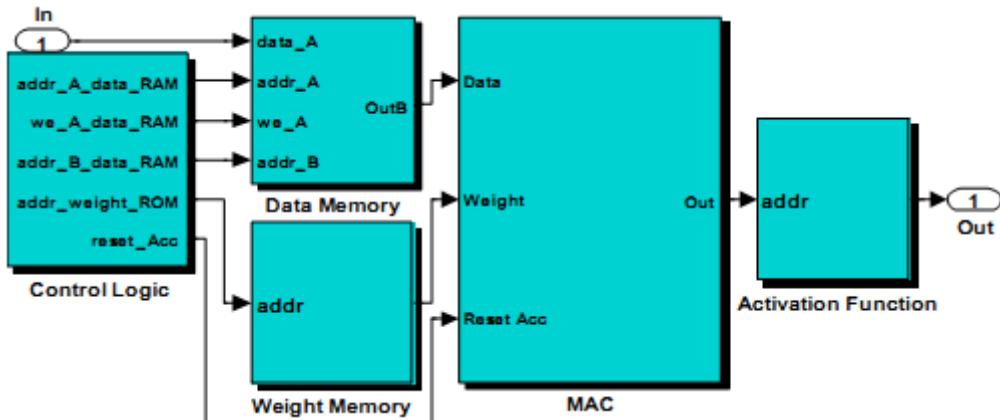


Figura 4.3 Esquema do neurônio

A lógica de controle seta os parâmetros para acessar as memórias tanto de entrada quanto dos pesos. O bloco do MAC computa o estado de ativação e o bloco da função de ativação computa a saída. A partir da definição do neurônio a rede pode ser montada. A Figura 4.4 ilustra um exemplo de uma camada da rede com 7 neurônios. As camadas são agrupadas para formarem uma rede. O esquema mostrado na Figura 4.4 indica que a camada atual não pode começar a processar um segundo grupo de entradas enquanto uma camada posterior não terminar o processamento da saída da camada atual para o primeiro grupo de entradas. Por esta razão, nenhum mecanismo de armazenamento é utilizado na saída da camada. Assim, a arquitetura proposta não permite que as camadas trabalhem concorrentemente (paralelismo de camadas) visto que as saídas da camada precisariam ser armazenadas para que não fossem sobreescritas.

A frequência máxima atingida por uma rede de topologia 7:7:7 foi 135MHz. Os autores não fornecem informações sobre a área total da rede de exemplo quando prototipada em FPGA.

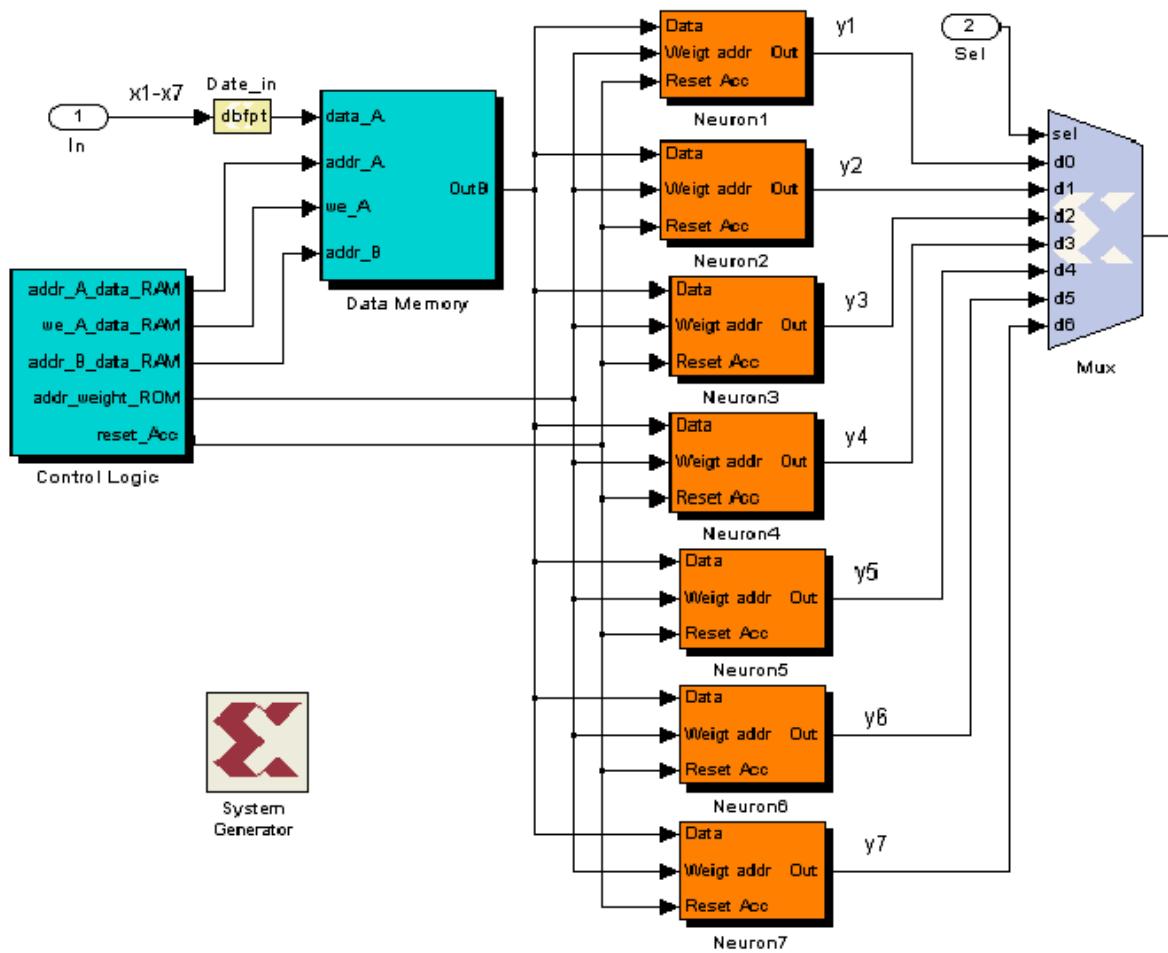


Figura 4.4 Esquema de uma camada com 7 neurônios

4.1.3 Field Programmable Gate Array (FPGA) based neural network implementation of Motion Control and fault diagnosis of induction motor drive (Soares, et al., 2006)

Este trabalho intenciona integrar uma técnica de controle inteligente com um controle de direção de alto desempenho. Normalmente os dispositivos mais usados para desenvolver estratégias de controle em hardware são os DSPs (digital signal processors). Entretanto, sistemas de controle inteligentes baseados em redes neurais permitem um processamento paralelo, implementações econômicas em ASICs, adaptabilidade e tolerância a falhas.

Tradicionalmente, RNAs são implementadas por um único ou por vários DSPs. E mesmo com as melhorias introduzidas nos mais modernos DSPs a computação continua sendo sequencial.

Neste cenário, a construção de ASICs de uma RNA seria muito custoso devido ao caro e longo processo de desenvolvimento do chip. Por outro lado, a implementação de uma RNA em FPGAs traz a vantagem da flexibilidade e da diminuição do custo do projeto, sem perda expressiva de desempenho.

O diagrama de blocos do sistema construído para o sistema de controle tem a forma da Figura 4.5. Nesta figura, o bloco de detecção de falhas tem a finalidade de verificar as voltagens que podem ser provocadas por falhas nas chaves do circuito do motor. A rede é treinada utilizando padrões normais e anormais de voltagens.

Os autores decidiram treinar 7 redes neurais, uma para cada cenário, como mostrado na Figura 4.6.

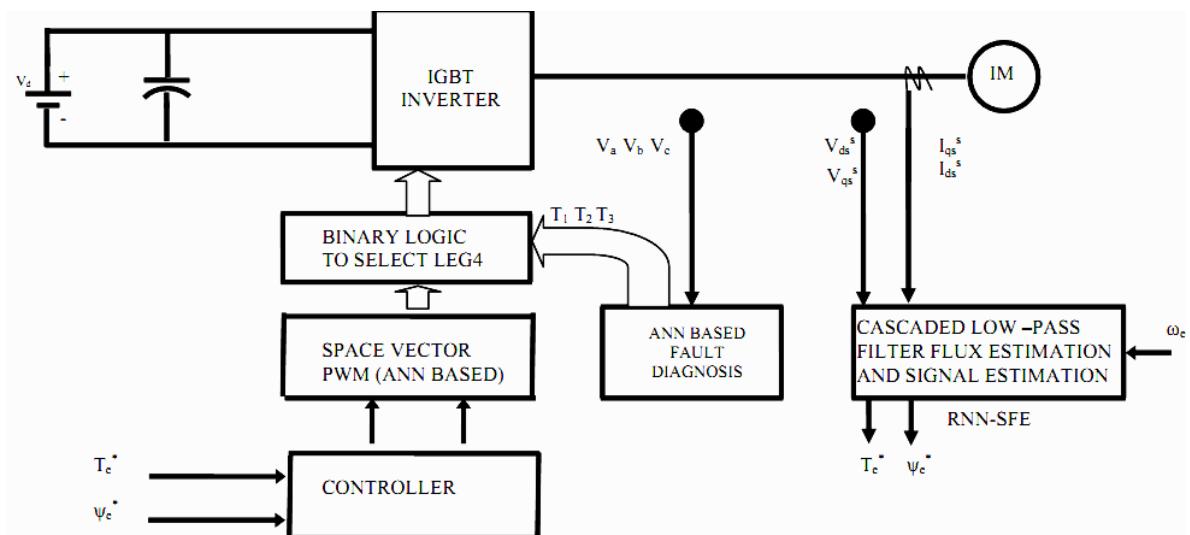


Figura 4.5 sistema de detecção de falhas em motores

Um código de saída é gerado correspondente a concatenação das saídas das redes arredondadas para 0 ou 1. Os padrões de entrada são 7 vetores de 40 características cada (as redes neurais tem 40 entradas cada).

A função de ativação utilizada foi a sigmoide aproximada por 7 segmentos de retas a

fim de reduzir a complexidade da implementação. A arquitetura do neurônio é composta por 1 somador, 1 multiplicador, 1 acumulador e 2 MUX como mostrado na Figura 4.7.

Os mesmos blocos que computam a soma de produtos realizam a computação da sigmoide. Isso faz com que o neurônio seja sequencial e mais lento. O tempo de resposta para uma entrada foi relatado pelos autores como sendo de $2\mu s$ e foi utilizada aritmética de ponto flutuante (precisão simples).

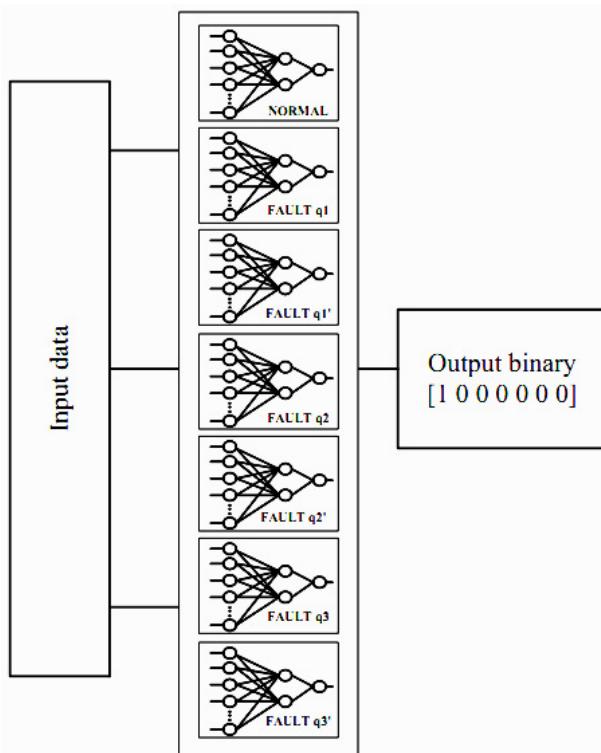


Figura 4.6 arquitetura do sistema de detecção de falhas com 7 RNAs

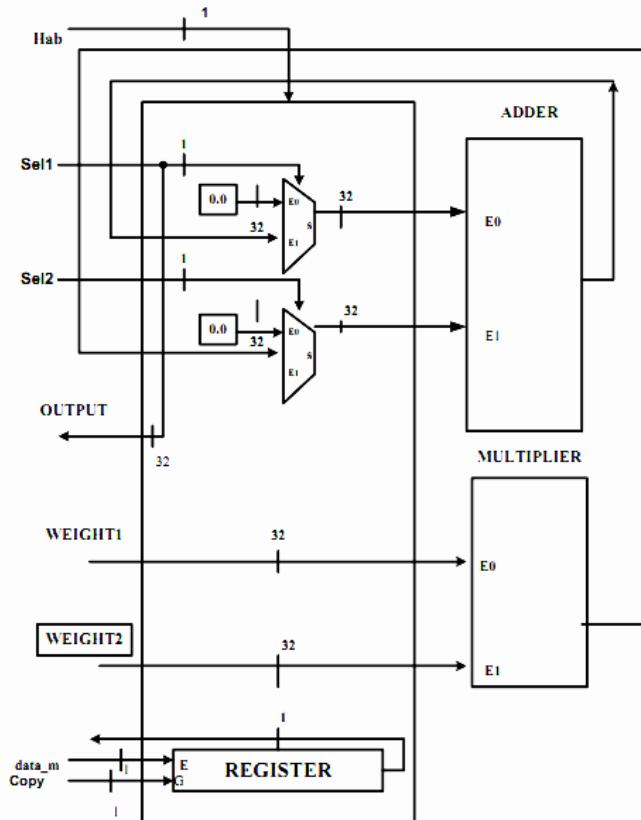


Figura 4.7 arquitetura do neurônio

4.1.4 Análise Comparativa

A Tabela 4.1 mostra as características analisadas dos três trabalhos de implementação de RNAs em FPGAs citados neste capítulo, são elas:

1. Taxa de alimentação das entradas – as entradas são necessárias todas de uma vez ou sequencialmente? Caso todas as entradas forem necessárias de uma vez mais recursos de armazenamento e de interconexão são necessários.
2. Paralelismo de camadas – as camadas processam em paralelo? Caso haja, há um aumento do desempenho da rede.
3. Paralelismo de neurônios – os neurônios de uma camada são processados em paralelo? É importante porque aumenta a taxa de computação da rede como um todo.
4. Compromisso Área-desempenho – está previsto o trade-off área/desempenho?

Caso exista permite que se escolha, convenientemente, diferentes instâncias da mesma rede otimizando ou área ou desempenho.

5. Sensível à grande quantidade de entradas – existe alguma limitação para RNAs com muitas entradas? Este fator pode determinar se uma rede com grande quantidade de entradas pode ou não ser realizada em hardware.
6. Sensível aos recursos de interconexão – a arquitetura foi pensada para minimizar os recursos de interconexão entre as camadas? Na medida em que o número de neurônios cresce a frequência máxima pode decrescer pelo crescente uso de barramentos e vias do FPGA.
7. Uso de área – qual o nível de consumo de área (alto, médio, baixo)?
8. Fmax – qual a frequência (em MHz) atingida?
9. Ponto fixo ou flutuante – qual representação foi utilizada? Ponto fixo demanda um projeto mais customizado, enquanto ponto flutuante pode ser amplamente utilizado sem preocupação do intervalo de representação das variáveis.
10. Fácil modificação da função de ativação – é fácil trocar a função de ativação por outra função? Ex.: sigmoide por tangente hiperbólica. Permite que diversas funções de ativação sejam utilizadas da mesma forma que no projeto de RNAs em software.
11. Função de ativação com precisão customizável – é possível utilizar diferentes precisões da função de ativação? Este requisito proporciona uma maior customização do uso de recursos de hardware. Ou seja, caso se precise de uma maior precisão, mais recursos são utilizados, caso contrário, os recursos podem ser destinados à melhoria de desempenho.

Tabela 4.1 Quadro Comparativo dos trabalhos de implementação

	1	2	3	4	5	6	7	8	9	10	11
Lee06	-	-	-	-	-	-	-	38	Flutuante 32bits	Não	Não
Oniga09	Serial	Não	Sim	Sim	Não	Não	Alto	135	Inteiros	Não	Não
Soares06	Serial	Não	Sim	Não	Não	Não	Baixo	-	Flutuante 32 bits	Não	Não

Todos esses requisitos foram observados na proposta da arquitetura de RNAs em FPGAs detalhada no capítulo 5.

4.2 Geração automática de RNAs em linguagem HDL

4.2.1 Automatic synthesizable VHDL code generation from neural networks models using Matlab (Gugala, et al., 2009)

Este trabalho propõe um mecanismo para geração automática de código VHDL sintetizável para RNAs desenvolvidas na ferramenta Matlab.

Os autores enfatizam a necessidade de se ter uma ferramenta para geração do código em uma linguagem HDL visto que seria necessária a descrição manual caso uma implementação em hardware fosse necessária. Essa facilidade permite um grande ganho, em relação a implementação manual, em produtividade, diminuindo o tempo de projeto.

O algoritmo proposto pelos autores para gerar o código VHDL é dividido em duas fases:

- Preparação das estruturas de dados
- Geração dos arquivos na linguagem VHDL

O algoritmo foi implementado na linguagem Matlab script e parametrizado em funções. A função de mais alto nível é a `Generate_VHDL_Model(net, encoding,exponentRes, fractionRes)`. Esta função captura no objeto `net` do ambiente do Matlab todas informações da RNA previamente treinada na ferramenta. Os demais argumentos dizem respeito a codificação dos dados de entrada da rede. *Encoding* indica se serão usados números de ponto flutuante ou de ponto fixo; e os últimos dois argumentos dizem respeito a parte fracionária e a parte inteira do número.

A arquitetura gerada automaticamente segue a estrutura canônica da rede neural MLP como mostra a Figura 4.8.

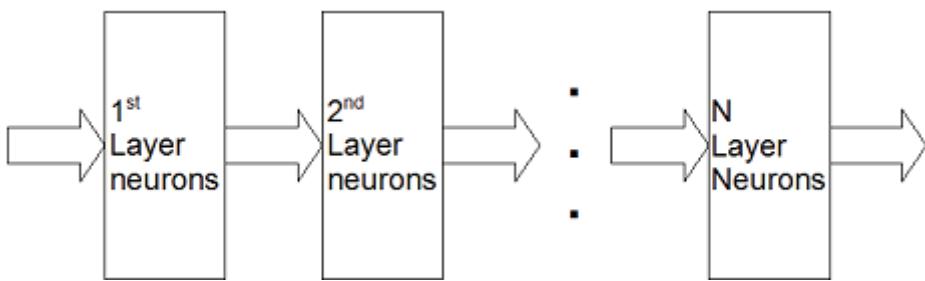


Figura 4.8 estrutura da rede gerada

Os autores informam que a implementação das operações aritméticas é realizada por pacotes (da linguagem VHDL) disponibilizados na biblioteca IEEE (IEEE). Nenhum detalhe da arquitetura interna do neurônio artificial é dado além do fluxo das operações como mostra a Figura 4.9. A função de ativação é implementada como uma look-up table.

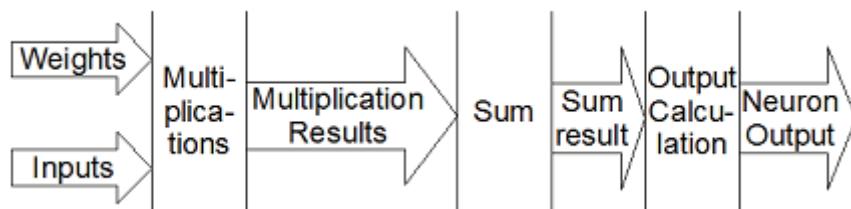


Figura 4.9 fluxo de dados do neurônio artificial

Um único teste feito foi com uma rede **perceptron** (sem camadas intermediárias)

com topologia 5:1. Essa rede corresponde a apenas 1 neurônio com 5 entradas. Esse projeto ocupou 38% do FPGA utilizado (Xilinx Spartan 3E). Nenhuma informação da frequência máxima atingida pelo circuito ou sobre o *speed-up* em relação à implementação em software foi fornecida.

4.2.2 VANNGen- a Flexible CAD Tool for Hardware Implementation of Artificial Neural Networks (Braga, et al., 2005)

VANNGen é um acrônimo para VHDL ANNs Generator. Este gerador se propõe a automatizar a implementação de RNAs MLP (com diferentes configurações) em FPGAs.

A arquitetura proposta pelos autores utiliza multiplicadores baseados em *look-up tables*. Esta técnica pode ser usada quando um dos operandos é constante, neste caso o operando constante é um peso associado a uma entrada em um dado neurônio. Isso reduz o uso de lógica, embora aumente significativamente o uso de memória (um recurso não tão abundante em FPGAs), além de não se valer dos blocos de DSPs disponíveis nos FPGAs. Nessa arquitetura cada neurônio contém o número de multiplicadores igual ao número de entradas. Isso inviabiliza uma implementação de redes com grande número de entradas.

Na arquitetura do neurônio proposta (Figura 4.10), pode-se ver as *look-up tables* que implementam a multiplicação e o adicionador, ambos de ponto fixo.

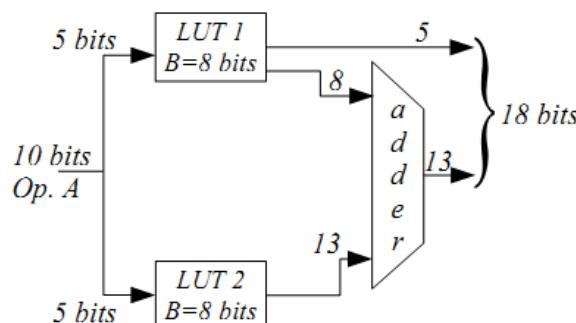


Figura 4.10 arquitetura do neurônio

Uma máquina de estados com 5 estados controla o caminho de dados do neurônio. São eles: start-state, multiplication-state, sum-state, stabilization-state e response-state. A arquitetura do gerador (implementado na linguagem Java) é mostrada na Figura 4.11, a partir dos *templates* e das escolhas do usuário o núcleo do gerador produz o código fonte da RNA em VHDL.

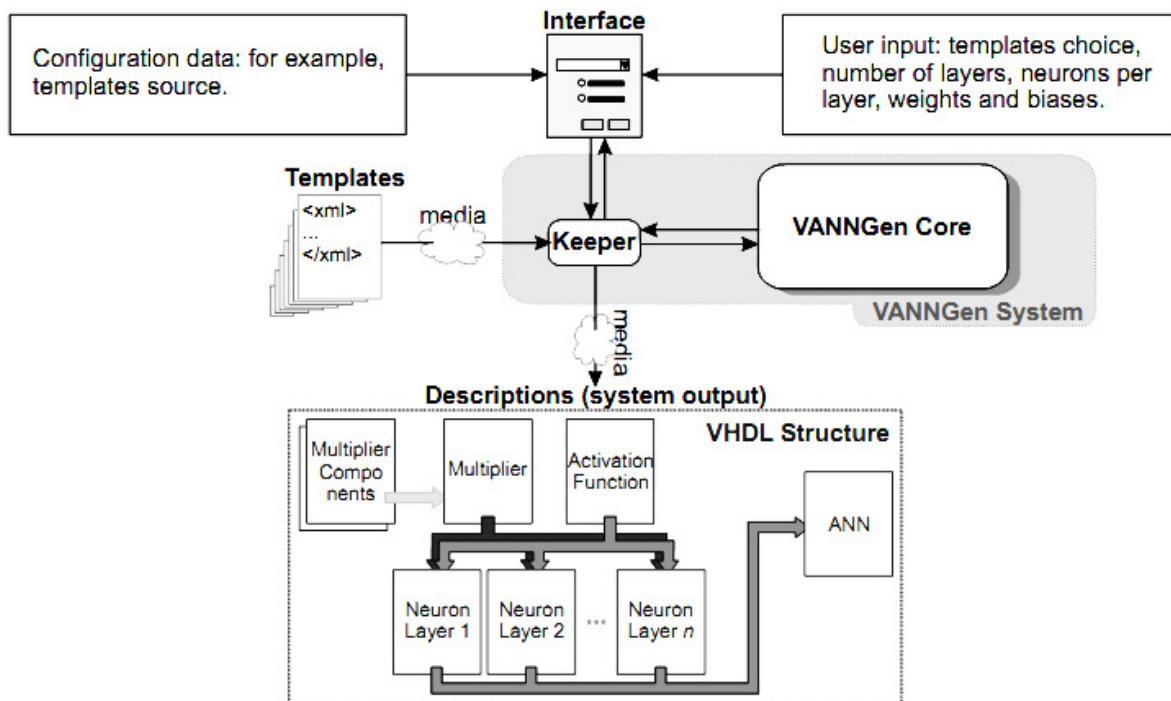


Figura 4.11 sistema do VANNGen

O sistema foi testado utilizando uma rede, com topologia 1:2:1, que aproxima a função:

$$y = \frac{1}{e^{0.03x}} * \sin(0.003x)$$

Duas configurações foram implementadas, a primeira utilizando números de ponto fixo de 16 bits e a segunda com 15 bits. A função de ativação foi implementada através de uma *look-up table* com 2^{16} e com 2^{15} posições. Obviamente que com essa quantidade de memória utilizada por cada *look-up table* e cada neurônio necessitando (pela arquitetura proposta) de uma *look-up table*, não se pode instanciar muitos neurônios. Fato ainda piorado pelo uso de memória também na implementação dos multiplicadores.

A frequência de operação do circuito da RNA ficou em torno de 100MHz e os autores não compararam o tempo de execução com nenhuma outra plataforma de referência.

4.2.3 Análise Comparativa

A Tabela 4.2 mostra as características analisadas dos dois trabalhos de geração automática de RNAs em linguagem HDL citados neste capítulo, quais sejam:

1. Linguagem do gerador. Esse ponto é importante porque a linguagem do gerador pode não ser de propósito geral ou amarrada a alguma ferramenta CAD de RNAs.
2. Linguagem gerada (HDL).
3. Integração com alguma ferramenta CAD. A implementação dessa integração facilita o fluxo de projeto das RNAs em FPGA.
4. Migração para outras ferramentas CAD. Para o usuário do gerador de RNAs é importante que o gerador não esteja atrelado a apenas uma ferramenta CAD de RNAs e sim àquela que melhor convier ao usuário.
5. Eficiência da arquitetura gerada.
6. Consumo de área da arquitetura gerada.
7. Fmax – frequência (em MHz) máxima atingida.
8. Ponto fixo ou flutuante.

Tabela 4.2 Quadro comparativo dos trabalhos relacionados de geração automática

	1	2	3	4	5	6	7	8
Gugala09	Matlab script	Vhdl	matlab	Não	-	-	-	ambos
Braga05	Java	Vhdl	nenhuma	Não	Baixa	Alto	100	fixo

Nestes trabalhos de geração automática, pôde ser notado que existe pouca ou

inexistente integração com as ferramentas CAD de RNAs, ou quando ela existe não é flexível o suficiente para abranger outras ferramentas. Outro problema é que o foco dos trabalhos ficou restrito à geração automática, enquanto que as arquiteturas das RNAs se mostraram baixo desempenho.

A seguir, no capítulo 5, a arquitetura que se propõe a permitir a geração automática de uma arquitetura eficiente, que utiliza aritmética de ponto flutuante (vide seção 6.1) é proposta e apresentada em detalhes. O gerador implementado foi escrito na linguagem C/C++ para garantir portabilidade e permitir sua integração com qualquer ferramenta CAD de RNAs.

5. Solução proposta

Neste trabalho está sendo proposta uma arquitetura paralela para implementação eficiente de RNAs em FPGAs, bem como uma técnica para automatização do fluxo de projeto de desenvolvimento das RNAs nestes dispositivos. O fluxo de projeto proposto está ilustrado na Figura 5.1. Inicialmente o projetista de IA realiza o projeto da arquitetura da RNA utilizando as ferramentas CAD para projetos de RNA (vide seção 2.4).

Para extrair os parâmetros da representação da RNA desenvolvida na ferramenta CAD foi desenvolvido um *script* que converte esse formato interno em um formato intermediário que contém os pesos das camadas e a topologia da rede. A utilização deste *script* e do formato intermediário permite que o gerador automático seja utilizado em diferentes ferramentas CAD para projeto RNAs. Para tanto apenas precisa ser modificado o *script* na linguagem da ferramenta CAD (e.g Matlab script).

A partir das informações contidas nos arquivos que compõem o formato intermediário, o usuário pode gerar uma descrição em SystemVerilog da RNA projetada na ferramenta CAD. O gerador da RNA foi desenvolvido na linguagem C/C++ (ao invés de uma linguagem de *script* da ferramenta CAD) a fim de garantir uma maior portabilidade da aplicação.

O gerador automático de RNAs é responsável pela instanciação dos módulos de hardware previamente implementados (que formam uma biblioteca de módulos parametrizáveis) e pela definição dos parâmetros destes. Mais detalhes sobre o gerador serão dados na seção 6.3.

O código é, então, gerado na linguagem Systemverilog implementada no nível RTL (Register Transfer Level) e pode ser prototipado ou simulado. A disponibilidade do gerador de RNAs proposto neste trabalho reduz o tempo de projeto da Rede Neural em um projeto de hardware digital.

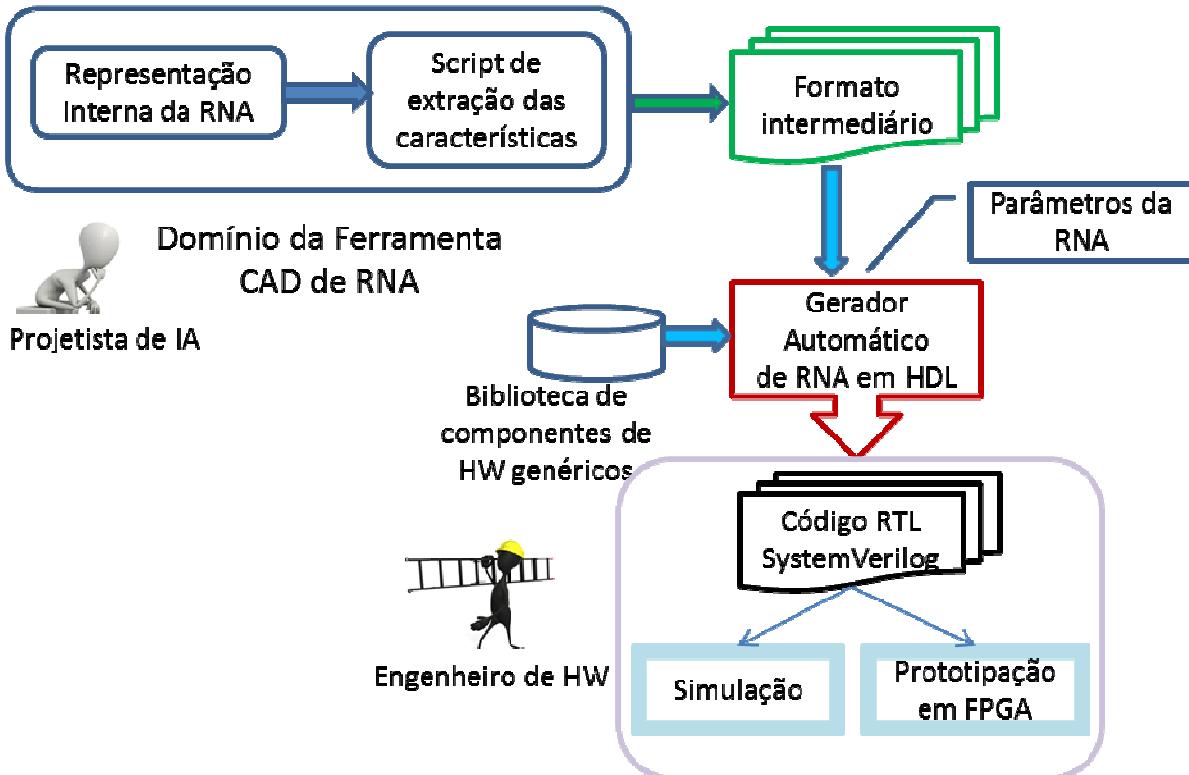


Figura 5.1 fluxo de projeto da RNA com o gerador automático

5.1 Computação do Estado de ativação

Como descrito na seção 2.1, cada neurônio artificial computa a soma de produtos:

$$u = \sum_{j=1}^{m+1} w_j \cdot x_j$$

Assim, uma dada **camada de neurônios** computa o produto matriz-vetor:

$$U = W_{n,m+1} \cdot X_{m+1,1}$$

Em que a matriz de pesos W , com dimensão n (neurônios) e $m+1$ (m entradas mais bias), é multiplicada pelo vetor de entradas X .

No algoritmo convencional de multiplicação de matrizes, cada linha da matriz W é multiplicada pelo vetor X . Isto significa que todo vetor de entrada precisaria estar completamente armazenado para realizar os n produtos linha da matriz W pela

coluna do vetor X.

$$U = \begin{pmatrix} w_{1,1} & \cdots & w_{1,m+1} \\ \vdots & \ddots & \vdots \\ w_{n,1} & \cdots & w_{n,m+1} \end{pmatrix} * \begin{pmatrix} x_1 \\ \vdots \\ x_m \\ -1 \end{pmatrix} = \begin{pmatrix} w_{1,1}x_1 + w_{1,m}x_m - w_{1,m+1} \\ \vdots \\ w_{n,1}x_1 + w_{n,m}x_m - w_{n,m+1} \end{pmatrix}$$

A implementação desta operação em hardware possui complexidade de interconexões, de memória (vias de ligação e registradores) e de pinos que é, no pior caso, $O(m)$. Esse fato pode inviabilizar a implementação de redes neurais com um número grande de entradas.

Neste trabalho, utilizamos a abordagem já utilizada por (Souza, et al., 2009) que propõe a computação do produto de matrizes usando a estratégia coluna-linha de forma a melhorar o reuso dos dados de entrada.

No escopo deste trabalho a estratégia coluna-linha se resume ao produto da coluna de W por um ponto de X. Dessa forma, primeiro computamos os produtos $w_{1,1}x_1, \dots, w_{n,1}x_1$ que corresponde à primeira coluna da matriz resultado U, e sucessivamente as demais colunas serão computadas.

Nesta abordagem, os valores de entrada (vetor X) são necessários apenas uma vez (não precisando ser armazenados); à medida que o primeiro dado é lido já se pode iniciar o processamento; e a complexidade dos recursos de interconexão e de pinos do FPGA é então da ordem $O(1)$.

Esses resultados são bastante interessantes porque a mesma dinâmica pode ser aplicada tanto para a 1^a camada intermediária quanto para a 2^a entrada. No caso da 1^a camada intermediária, essa abordagem reduz a taxa de dados a serem lidas pelo fator do número de neurônios na referida camada. Assim, caso a 1^a camada intermediária tenha 2 neurônios a taxa de entradas dos dados no FPGA precisa ser 1/2 da frequência de operação da RNA em FPGA. Isto acontece porque a entrada precisa ficar estável para que seja multiplicada por cada peso associado a ela em cada neurônio. Nesse exemplo, a entrada deverá ficar estável por 2 ciclos do clock de operação do circuito, não representando, portanto, a taxa de leitura de dados da memória externa um gargalo. Essa operação se assemelha mais a um "*stream*

processor" onde os dados podem ser produzidos aos poucos e processados prontamente.

A Figura 5.2 mostra como fica simplificada a estrutura de interconexão das camadas com a escolha da estratégia baseada no produto coluna-linha.

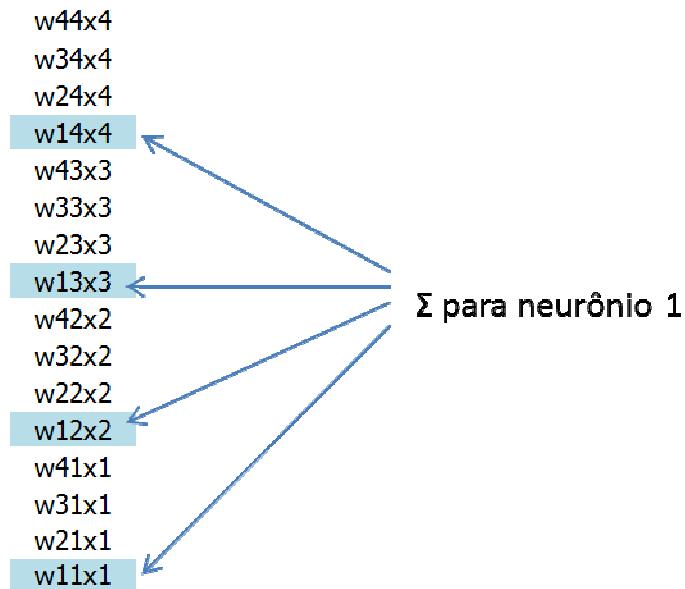


Figura 5.2 Estrutura de interconexão das camadas

Para computar a matriz U ainda falta somar os termos correspondentes de cada coluna. O primeiro termo da primeira coluna precisa ser somado com o primeiro termo da segunda coluna e assim sucessivamente.

$$U = \begin{pmatrix} w_{1,1}x_1 & + & w_{1,m}x_m & -w_{1,m+1} \\ \vdots & \ddots & \vdots & \vdots \\ w_{n,1}x_1 & + & w_{n,m}x_m & -w_{n,m+1} \end{pmatrix}$$

Num exemplo de uma camada com 4 neurônios e 4 entradas a primeira operação é a multiplicação. Assim, cada entrada é apresentada e pode ser multiplicada pelos pesos correspondentes, resultando no fluxo de dados abaixo. As linhas sombreadas correspondem aos dados que precisam ser somados para o neurônio 1. O mesmo acontece para os outros neurônios. Observa-se que o bias (vide seção 2.1) ainda não foi adicionado.



O caminho de dados proposto visa agrupar os dados correspondentes a cada neurônio somando os pares correspondentes. Ou seja $w_{11}x_1$ é somado primeiro a $w_{12}x_2$, assim como $w_{13}x_3$ é somado a $w_{14}x_4$. Após isso, a próxima operação é $w_{11}x_1+w_{12}x_2$ ser somado a $w_{13}x_3+w_{14}x_4$. Esta operação é feita para os demais neurônios da camada.

Veja na Figura 5.3 o fluxo dos dados que sai do multiplicador (que calcula os produtos $w_{ij} \cdot x_j$), entram num shift-register e ao mesmo tempo entram no somador. O alinhamento dos dados correspondentes é feito pelo shift-register. Na parte 'a' da Figura 5.3 os pesos associados a entrada x_1 já estão no shift-register. O próximo passo na parte 'b' é a saída da parcela $w_{12}x_2$ que entra no shift-register e no somador. Se passaram 4 ciclos de clock até que tem-se a configuração da parte 'c' da Figura 5.3. Neste instante soma-se o par $w_{41}x_1$ com $w_{42}x_2$ e no shift-register existem dados que não serão somados mais a ninguém. Nesse momento até que o shift-register se encha novamente temos bolhas no pipeline (como mostrado na parte 'd' da Figura 5.3).

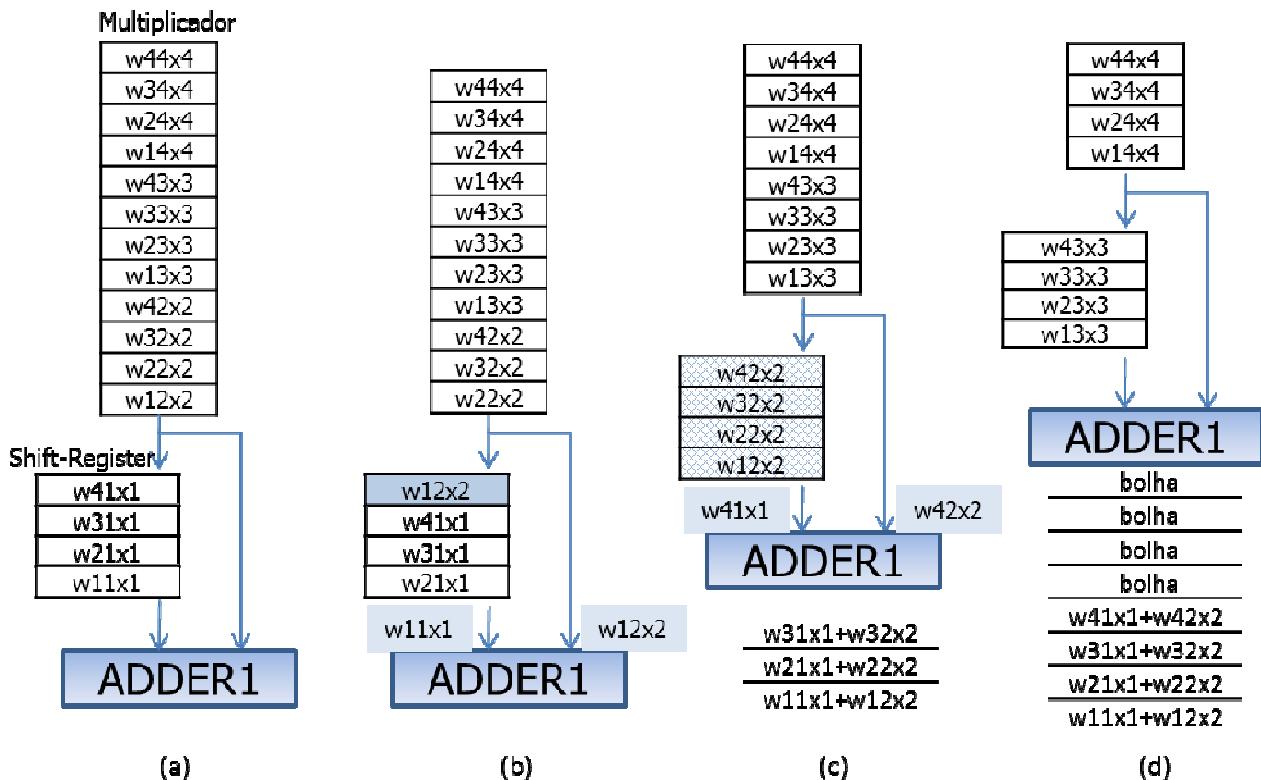


Figura 5.3 Exemplo do pipeline até a primeira soma em uma camada com 4 neurônios e 4 entradas

No próximo ciclo de clock a partir da parte 'd' da Figura 5.3 as parcelas $w13x3$ e $w14x4$ entram no somador e o processo se reinicia.

O pipeline da figura Figura 5.3 mostra o fluxo de dados até o primeiro somador. Veja na Figura 5.4 o caminho de dados completo para uma camada com 4 neurônios e 4 entradas.

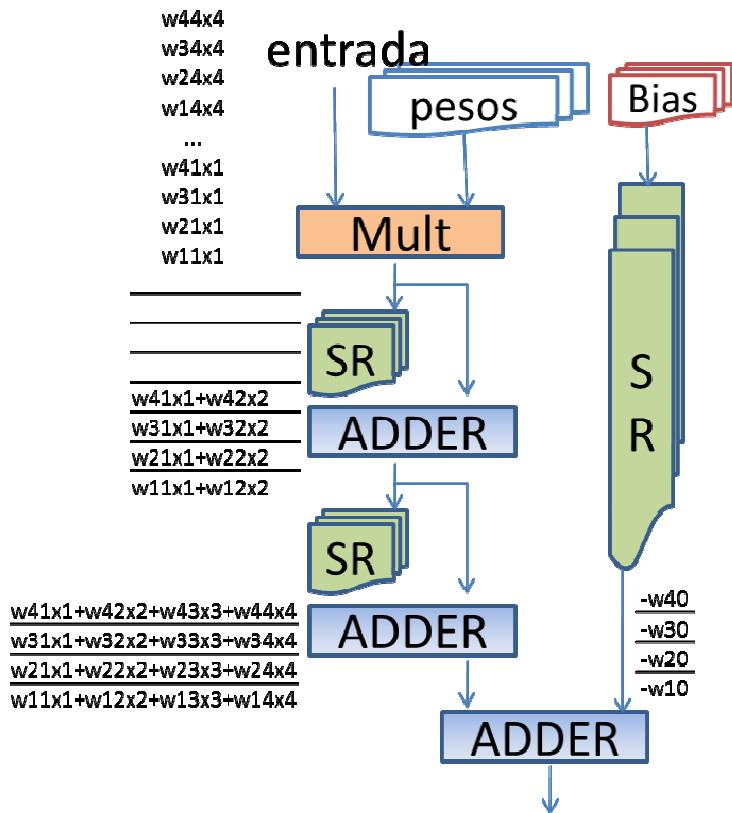


Figura 5.4 caminho de dados 4 neurônios 4 entradas

Nota-se que com duas somas seria possível completar o cálculo da soma de produtos, porém ainda existe o *bias* que precisa ser adicionado. Para o caso em que o número de entradas é uma potência de 2, o *bias* (armazenado como $-bias$, no módulo homônimo) deve ser adicionado ao final (vide Figura 5.4).

Exemplo 2: considere agora uma camada com 4 neurônios e 5 entradas. O número de entradas vai indicar quantos somadores existirão no caminho de dados. A Figura 5.5 mostra o caminho de dados para o exemplo 2. O destaque é para o *bias* que entra como uma entrada -1 a ser multiplicada pelo peso correspondente.

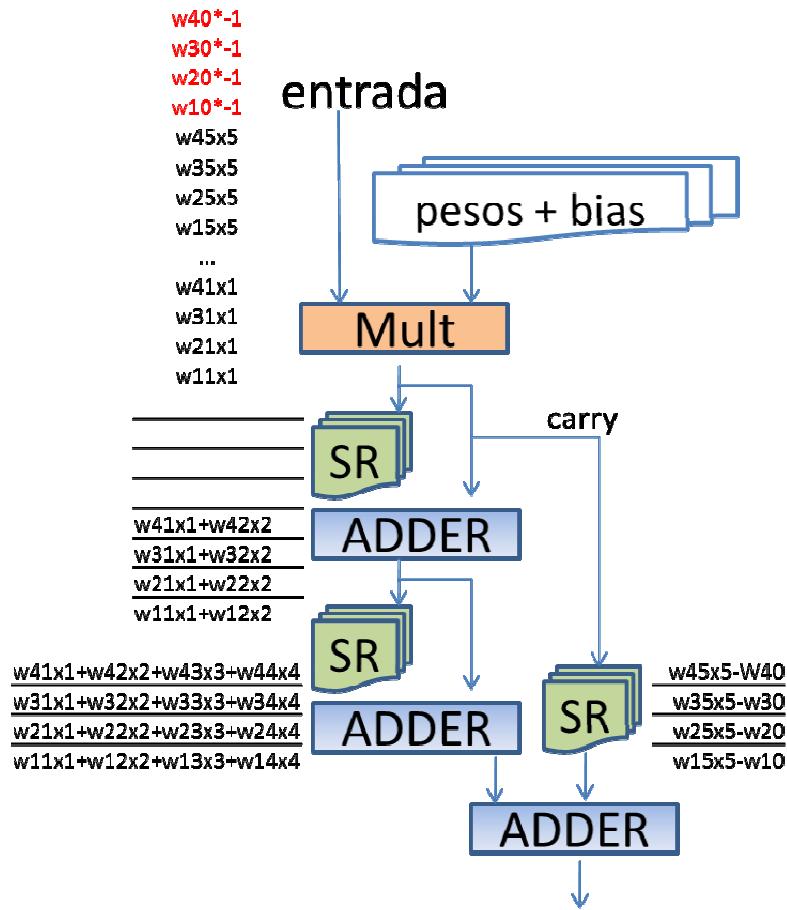


Figura 5.5 caminho de dados para camada com 4 neurônios e 5 entradas

Faz-se isso porque a quantidade de entradas dessa camada é um número ímpar e a adição de 1 torna-o par. Dessa forma pode-se formar grupos 2 a 2 e trabalha-se como se a camada tivesse 6 entradas. Após o primeiro somador tem-se 3 grupos (três parcelas a serem somadas) o que impede um novo agrupamento 2 a 2 para todas elas. Resolve-se esse impasse levando o último grupo para o próximo somador onde haja um total de grupos ímpar (o que aqui ocorre com 1 grupo, ou seja no final). A essa parcela que não teve par e foi passada adiante para encontrar um par deu-se o nome de carry.

Várias conclusões podem ser tiradas desses exemplos:

1. A operação segue agrupando os grupos 2 a 2, i.e. dividindo o número de grupos por 2 a cada somador.
 2. O número de somadores é, portanto $\log_2(\text{entradas}) + 1$

3. As bolhas no pipeline são as potências de 2 multiplicadas pelo número de neuronios. No exemplo 1 o primeiro shift register tem de retardar o dado em 4 ciclos = 1*4(neuronios); no segundo SR 2*4, e assim sucessivamente.

A prova da conclusão 2 segue a noção de que se um número não é par ele pode ser adicionado de 1 ou subtraído de 1 para torná-lo par. É o que pode ser visto no código abaixo:

```

1   bias_inserted = carry = count = adders = 0;
2
3   while n ~= 1,
4
5       if(mod(n,2)==0) %can divide by 2
6           n = n/2;
7           adders = adders+1;
8       elseif (bias_inserted == 0)
9           n = (n+1)/2;
10          bias_inserted = 1;
11          adders = adders+1;
12      elseif (carry == 0 )
13          n = (n-1)/2;
14          carry=1;
15          count = count+1;
16          adders = adders+1;
17      elseif(carry == 1 )
18          n = (n+1)/2;
19          carry=0;
20          adders = adders+1;
21      end
22  end
23
24 if(bias_inserted == 0 || carry ==1)
25     adders = adders+1;
26 end

```

As iterações do algoritmo produzem sequências que podem ter elementos ímpares. As configurações dos caminhos de dados para as camadas com 10, 9, e 21 entradas são apresentadas a seguir:

- $10 \Rightarrow 5+b \Rightarrow 3-c_1 \Rightarrow 1+c_1$; no primeiro somador tem-se 10 parcelas, no segundo $10/2 = 5$, porém o bias ainda não foi adicionado e é inserido para tornar o número de parcelas par ($5+b = 6$). No terceiro somador tem-se $6/2 = 3$ parcelas o que demanda que se faça o carry (c_1). No quarto e último somador tem-se $3-c_1 = 2/2 = 1$. Logo, pode-se adicionar o carry c_1 fazendo

$1+c_1$.

- $9 \Rightarrow 9+b \Rightarrow 5 - c_1 \Rightarrow 2 \Rightarrow 1 + c_1$;
- $21 \Rightarrow 21+b \Rightarrow 11 - c_1 \Rightarrow 5 + c_1 \Rightarrow 3 - c_2 \Rightarrow 1 + c_2$;

A Figura 5.6 ilustra o caminho de dados para uma camada com 21 entradas. Observe que o carry c_1 é o último elemento do grupo com 11 ($11 - c_1 = 10$) e precisa entrar no somador 3 para ser somado com o último elemento do conjunto com 5 ($5 + c_1$) parcelas possibilitando a soma. Esta camada é interessante porque dois *carries* são gerados. O segundo tem uma condição diferente do primeiro porque não entra em um somador interno. Vários arranjos podem ocorrer e estão previstos na arquitetura.

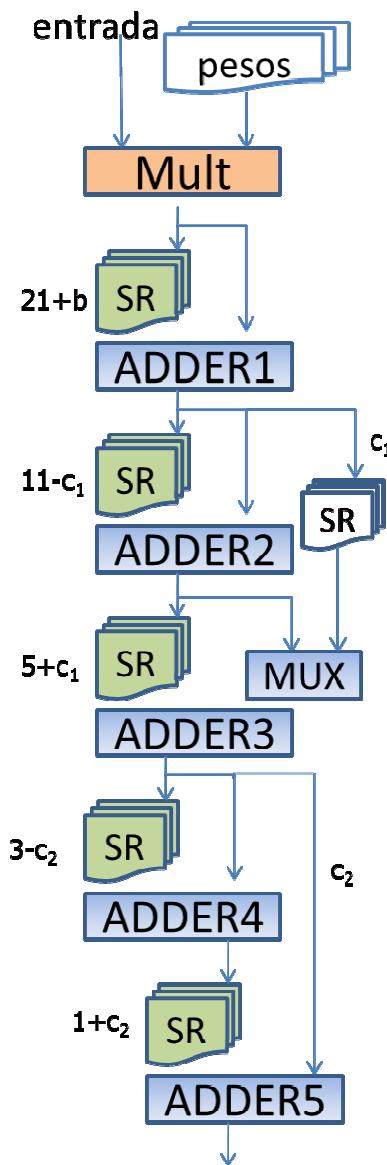


Figura 5.6 caminho de dados para camada com 21 entradas

- Caso m (o número de entradas) seja par
 - Caso o número de elementos ímpares na sequência é par
 - Adicione o bias no primeiro elemento ímpar
 - Ex.: $6 = 3+b, 2, 1$
 - Caso o número de elementos ímpares na sequência é ímpar
 - Adicione o bias no final (faça $1+b$)
 - Ex.: $8 = 4, 2, 1+b$
- Caso m seja ímpar
 - Adicione o bias como uma entrada e $m+1$ é par
 - Caso o número de elementos ímpares na sequência é ímpar
 - Adicione o ultimo carry no fim (faça $1+c$)
 - Ex.: $21 \Rightarrow 21+b, 11- c_1, 5+ c_1, 3- c_2, 1+ c_2$
 - Caso o número de elementos ímpares na sequência é par
 - Um carry dentro do próximo carry
 - Ex.: $13 \Rightarrow 13+b, 7- c_1, 3+ c_1, 2, 1$

Ao final conclui-se que o número de somadores necessários é igual a $\log_2(m) + 1$.

Na Figura 5.7 vemos a evolução do número de entradas (função linear) em comparação com o número de somadores necessários. Vê-se que o número de somadores necessários cresce muito mais lentamente que o número de entradas.

A arquitetura proposta neste trabalho também traz a vantagem de se ter implementações com diferentes trade-offs desempenho/área. Instanciando-se mais de um caminho de dados para uma dada camada de neurônios consegue-se aumentar o desempenho da computação. Na verdade a quantidade de possíveis caminhos de dados corresponde aos divisores do número de neurônios da referida camada.

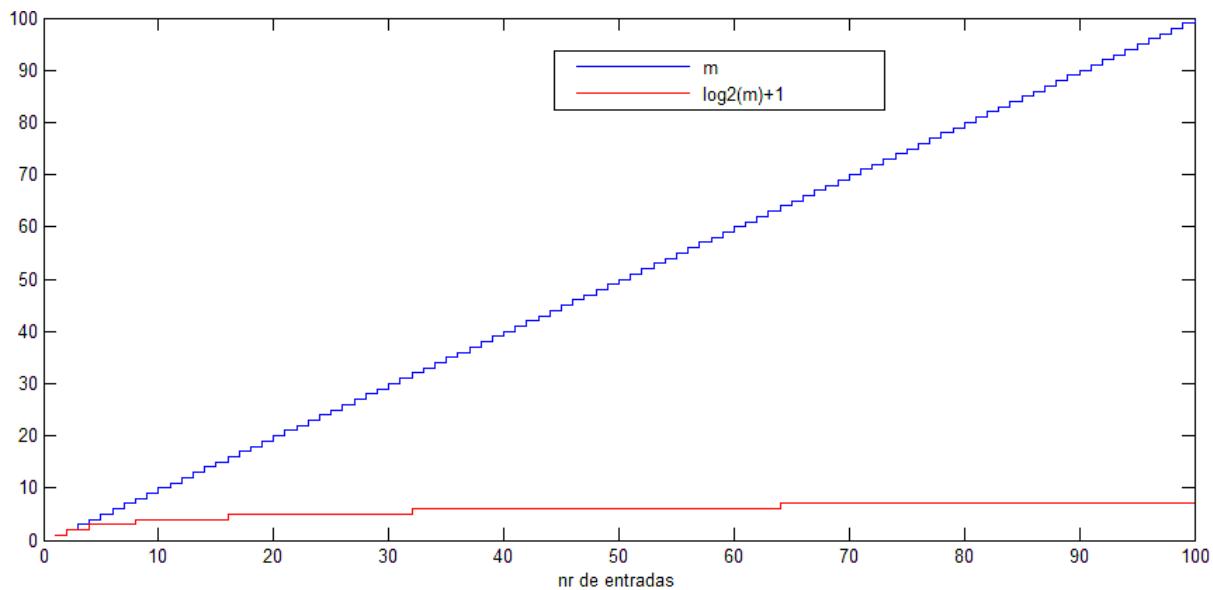


Figura 5.7 evolução de m comparado com $\log_2(m)+1$

Assim, uma camada com 4 neurônios pode ter 1, 2 e 4 caminhos de dados. Colocar 2 caminhos de dados, por exemplo, é o mesmo que fazer cada caminho ficar responsável por $n/2$ neurônios e assim por diante. Essa estratégia diminui os ciclos em que cada parcela da soma de produtos espera pelo seu correspondente no pipeline, aumentando a taxa de computação da camada. Na seção 5.3 dá-se mais detalhes a respeito do desempenho esperado e como a instanciação de mais de um caminho de dados pode melhorar o desempenho.

5.2 Aproximação da Função de Ativação

Como visto no capítulo 2, comumente se usa como função de ativação das redes neurais MLP a função sigmoide logística. Essa função é perfeitamente realizável, entretanto, uma implementação direta em hardware não é prática porque requer uma lógica excessiva.

Consequentemente, um grande número de aproximações, visando implementação em hardware, têm sido desenvolvidas. Desde a implementação direta em uma *Look up table* até outras categorias de aproximações como, por exemplo, lineares por

partes, segunda ordem por partes e os mapeamentos de entrada/saída combinacionais.

Uma abordagem mais intuitiva seria aproximar a função usando séries polinomiais como a série de Taylor que pode ser usada para representar qualquer função contínua arbitrária. Para se reduzir a ordem do polinômio pode-se valer do particionamento do domínio da função em subintervalos menores. Mas essa estratégia em hardware gera um maior número de lógica a ser implementada.

Olhando para a expansão da série de Taylor para a sigmoide vê-se que essa aproximação não é nem um pouco econômica.

$$\text{Sigmoid}_{\text{Taylor}} = \frac{1}{2} + \frac{1}{4} * x - \frac{1}{48} * x^3 + \frac{1}{480} * x^5 - \frac{17}{80640} * x^7 + \frac{31}{1451520} * x^9 \dots \quad (\text{eq. 5.1})$$

Logo foi necessário buscar outra aproximação que apresentasse melhor custo benefício entre área, tempo de processamento e dificuldade de implementação.

A primeira alternativa consistiu em uma aproximação linear por partes otimizada que define as funções (Basterretxea, et al., 2004):

$$\begin{aligned} y_1(x) &= 0 \\ y_2(x) &= \frac{1}{2} \left(1 + \frac{x}{2} \right) \\ y_3(x) &= 1 \end{aligned} \quad (\text{eq. 5.2})$$

O esquema é como segue na Figura 5.8:

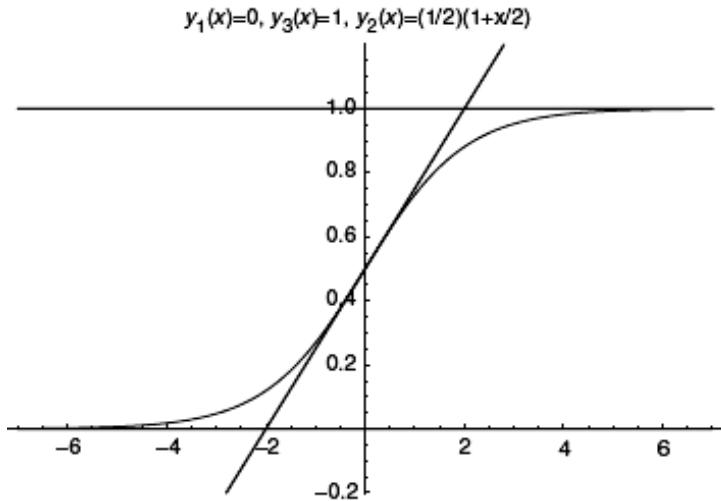


Figura 5.8 Aproximação linear por partes otimizada

A partir das funções iniciais o método computa a saída em q passos na forma da equação 5.3.

```

 $g(x) = y_1(x) = 0; h(x) = y_2(x) = 1/2(1 + x^2);$ 
for (i = 0; i = q; i++)
   $\{ g'(x) = \max [g(x), h(x)];$ 
     $h(x) = 1/2(g(x) + h(x) + \Delta);$  (eq. 5.3)
     $g(x) = g'(x);$ 
     $\Delta = \Delta/4; \}$ 
   $g(x) = \max [g(x), h(x)];$ 

```

Esse algoritmo define a parte negativa do domínio função. Havendo uma simetria, rebate-se a parte negativa para se obter a parte positiva.

O valor de Δ depende do valor de q. Os autores explicam que a melhor aproximação se dá para $q = 4$ e, por conseguinte $\Delta = 0.2638$. Esse método disponibiliza a saída em 4 passos. A implementação desse método em Matlab script encontra-se no apêndice.

Testando o método implementado no Matlab, constatou-se que o erro médio obtido

foi de 1.4539e-017; e o erro máximo igual a 0.0194, no intervalo -5 a 5. Na Figura 5.9 e na Figura 5.10 apresenta-se visualmente o comportamento da aproximação (esquerda) em comparação com a função sigmoide real (direita).

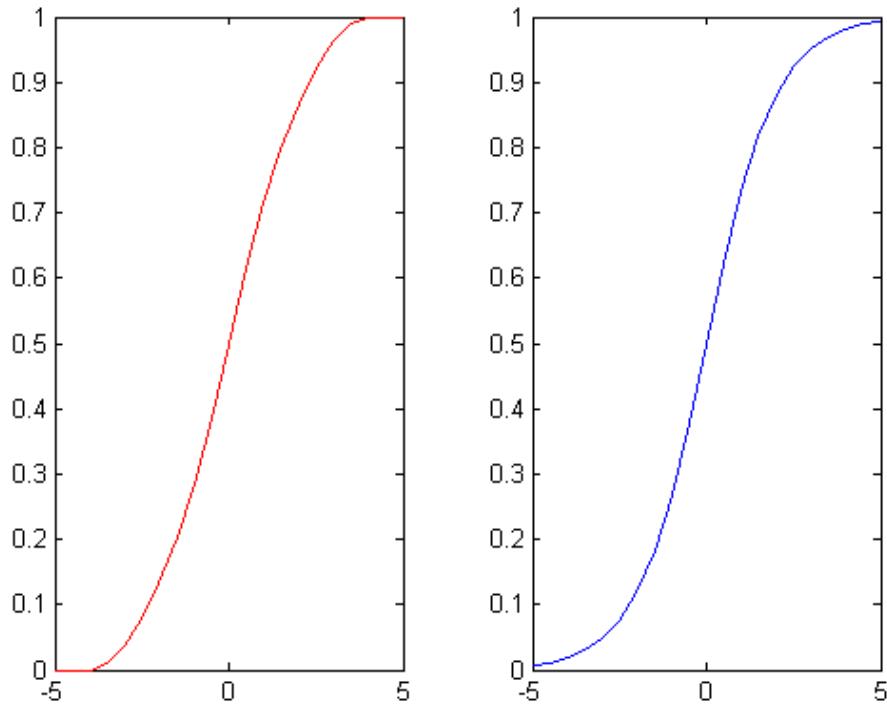


Figura 5.9 Aproximação linear otimizada (esquerda)

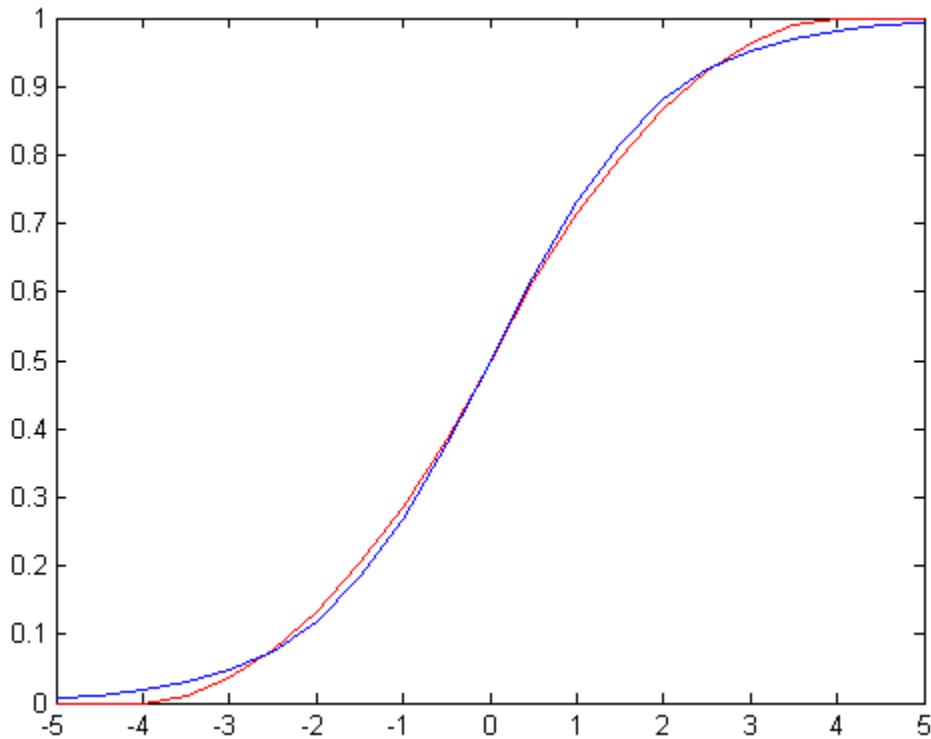


Figura 5.10 Aproximação linear otimizada vs sigmoide

Apesar da boa aproximação dada, nesse algoritmo são necessárias quatro iterações. Privilegiou-se a busca de uma forma mais direta da computação da sigmoide mesmo que o resultado seja menos preciso que o método atual.

O segundo método avaliado também pode ser classificado como linear por partes. A chamada PLAN Approximation foi proposta por Amin, Curtis e Hayes–Gill (Amin, et al., 1997) e é um método simples e direto de se implementar. Veja na Tabela 5.1 as retas que compõem a aproximação PLAN em cada intervalo.

Tabela 5.1 Aproximação PLAN

Operação	Condição
$Y = 1$	$ X \geq 5$
$Y = 0.03125 \cdot X + 0.84375$	$2.375 \leq X < 5$
$Y = 0.125 \cdot X + 0.625$	$1 \leq X < 2.375$
$Y = 0.25 \cdot X + 0.5$	$0 \leq X < 1$

A maior desvantagem também advém da simplicidade do método que apresenta uma aproximação não suave da sigmoide. Como visto no capítulo 2, a função de ativação precisa ser diferenciável em todos os pontos para que se possa ser aplicado um algoritmo de aprendizado baseado no gradiente descendente. Graficamente ficam bem evidentes os pontos de intersecção dos segmentos, como na Figura 5.11.

Embora este método apresente alguma imprecisão, o método PLAN é uma boa aproximação com erro médio 8.9214e-018 e erro máximo sendo 0.0189, no intervalo -5 a 5. A Figura 5.12 mostra que o método é melhor nas bordas do intervalo inspecionado.

A implementação para essa técnica também está disponível em Matlab script no apêndice.

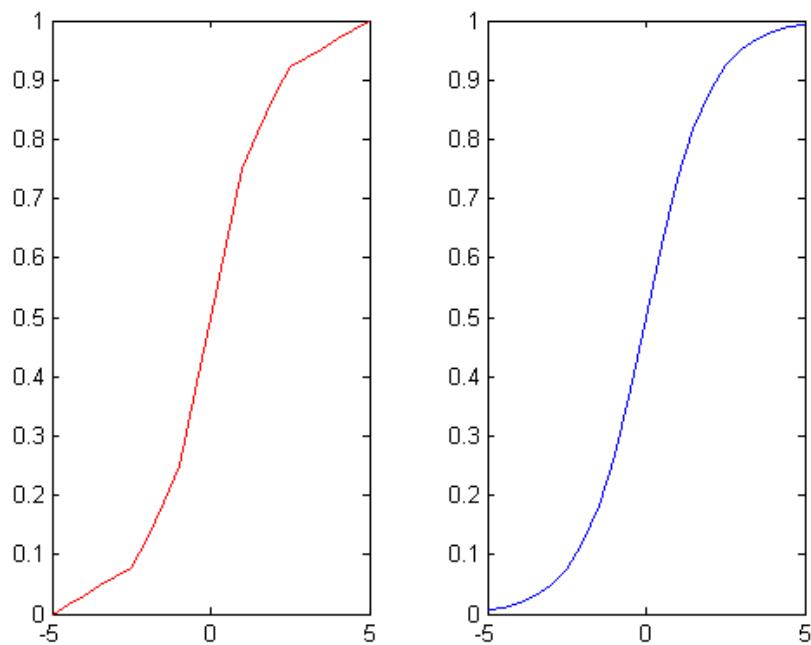


Figura 5.11 Aproximação PLAN (esquerda)

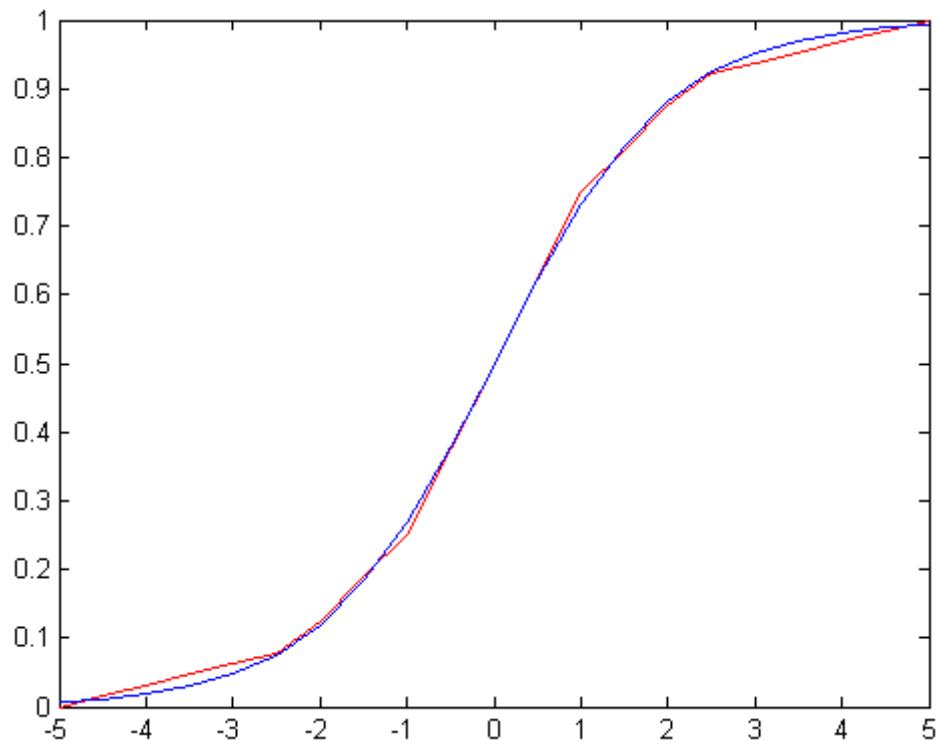


Figura 5.12 PLAN vs sigmoide

A terceira abordagem avaliada situa-se na classe das aproximações por partes de segunda ordem. Isso implica que a função tenha a forma de:

$$y(x) = c_0 + c_1 * x + c_2 * x^2. \quad (\text{eq. 5.4})$$

Uma desvantagem obvia desta abordagem é a necessidade de três multiplicações. Zhang, Vassiliadis e Delgado-Frias (Zhang, et al., 1996) propuseram uma versão que necessita de uma multiplicação (para calcular o quadrado) e o restante das multiplicações implementa-se com lógica combinacional. Ademais precisa-se de uma soma.

$$y = \begin{cases} 2^{-1} * (1 - |2^{-2} * x|)^2 & -4 < x < 0 \\ 1 - 2^{-1} * (1 - |2^{-2} * x|)^2 & 0 \leq x < 4 \end{cases} \quad (\text{eq. 5.5})$$

Esse método também apresenta a vantagem de fornecer uma aproximação suave (com uma descontinuidade em $x=0$). Essa característica propicia a uma possível extensão da arquitetura proposta para incluir a parte do aprendizado baseado num método de gradiente descendente. A implementação da técnica forneceu um erro médio de 8.5910e-018 e um erro máximo de 0.0215. Uma melhor visualização da suavidade (compatível com o 1º método analisado) pode ser vista na Figura 5.13.

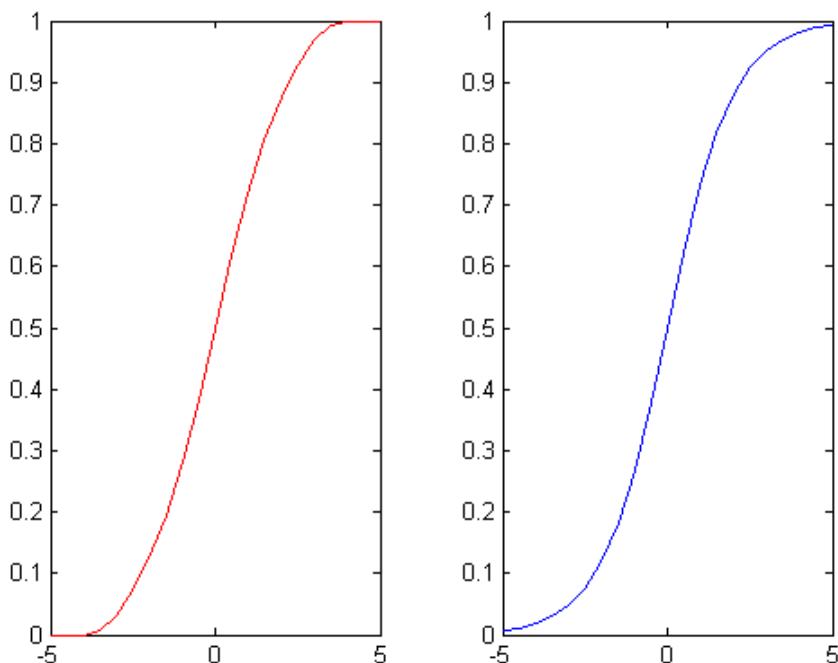


Figura 5.13 Aproximação de 2ª ordem (esquerda)

Sua acurácia é bem apresentada no comparativo com a função a ser aproximada como mostrada na Figura 5.14.

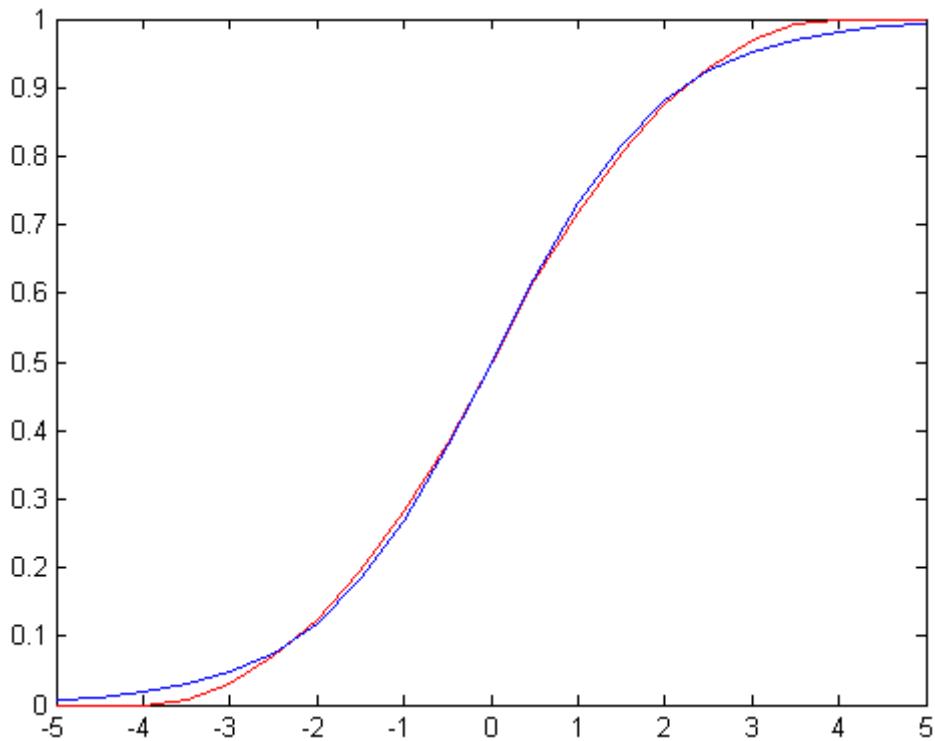


Figura 5.14 Aproximação de 2^a ordem vs sigmoida

O quadro comparativo da Tabela 5.2 ilustra melhor as características de cada abordagem. Os requisitos como erro médio, máximo, se a aproximação é suave (sem descontinuidades grandes) e por fim se é um método que apresenta bom desempenho.

Tabela 5.2 Quadro comparativo das aproximações

Método	erro médio	erro max	Suave	Rápido
1 ^a ordem por partes otimizado	1.4539e-017	0.0194	sim	não
1 ^a ordem por partes simples	8.9214e-018	0.0189	não	sim
2 ^a ordem por partes simples	8.5910e-018	0.0215	sim	sim

Analizando os dados da Tabela 5.2 constata-se que a terceira abordagem apresenta-se como a melhor escolha porque ela tem um erro máximo superior às outras, porém tem baixo erro médio, as características de ser suave, de fácil implementação e bom desempenho. Caso se necessite realizar a aprendizagem em FPGA, o método 3 seria o mais indicado.

No escopo deste trabalho não consta a aprendizagem online como um requisito. Sendo assim, o quarto método (além dos três summarizados na Tabela 5.2) de aproximação da função de ativação investigado foi o de se utilizar uma lookup table.

Este método é bastante intuitivo: são armazenados os valores da função alvo em uma tabela e através da escala de valores em x se indexa a tabela. O que demonstra que esse método é bem simples embora não pareça ser eficiente em área ocupada.

Outra característica interessante é que se pode instanciar qualquer outra função (e não apenas a sigmoide) como, por exemplo, a tangente hiperbólica. Além disso, a precisão da aproximação pode ser também ajustável com o tamanho da tabela (mais precisão = mais memória) oferecendo para o projetista uma precisão customizável em função do compromisso memória/precisão.

No entanto, esse método apresenta alguns problemas, conforme descrito a seguir.

1. Uma boa aproximação da função alvo só é conseguida com grande quantidade de valores armazenados e, por conseguinte, necessidade de grande quantidade de memória.
2. Encontrar o valor de x (neste trabalho é o resultado do estado de ativação) pode levar um tempo proporcional ao tamanho da *lookup table*. Mesmo com o uso de uma busca binária, levaria \log_2 (do tamanho da lookup table). Em uma implementação direta demandaria um número de comparadores (em paralelo) igual a $\log_2(\text{tamanho da lookup table})$.

O primeiro ponto não constitui um problema com a arquitetura detalhada na seção 5.1 visto que apenas uma *lookup table* é necessária para cada camada de neurônios. Logo, mais memória pode ser utilizada para garantir uma boa precisão para a função aproximada. Como apenas um elemento é necessário para a próxima camada, a

lookup table é acessada um elemento por vez, fato que simplifica bastante a implementação em hardware.

O segundo ponto pode ser resolvido com o desenvolvimento de uma forma mais eficiente de acessar a *lookup table* com complexidade O(1) ao invés de uma complexidade O(log2(tamanho)).

Técnica de busta na lookup table: Considere x como sendo a soma de produtos e y como sendo o resultado correspondente para a função de ativação, formando o par (x, y) . Represente x , cujo domínio é o intervalo fechado $[a, b]$ com $a < b$, com valores discretos de incremento $1/k$. Daí tem-se $(b - a).k + 1$ valores de x no intervalo $[a, b]$ e também a mesma quantidade de posições na *lookup table*. Nesses termos, a posição de um dado x é dada por $\lfloor (x - a) * k + 1 \rfloor$ e explicitando o x tem-se $\lfloor -a * k + 1 + x * k \rfloor$.

Exemplo: considere x no intervalo $[-1,1]$, com passo $1/k = 1/4$. A tabela tem $(1 + 1) * 4 + 1 = 9$ posições. O valor $x = 0$ está na posição $\lfloor (0 - (-1)) * 4 + 1 \rfloor = 5$ a posição de $x = 0,3$ é $\lfloor (0,3 - (-1)) * 4 + 1 \rfloor = \lfloor 6,2 \rfloor = 6$. Estes valores podem ser validados na Tabela 5.3.

Tabela 5.3 exemplo de aplicação de acesso a lookup table

1	2	3	4	5	6	7	8	9
-1	-0,75	-0,5	-0,25	0	0,25	0,5	0,75	1

Por razão de simplificação do hardware, se k for uma potência de 2 ($k = 2^Z$) o termo $x*k$ pode ser calculado com uma soma inteira de Z ao expoente de x (caso x seja um número de ponto flutuante). As outras parcelas são constantes.

Dessa simplificação resulta que se pode acessar a *lookup table* em tempo O(1) com apenas uma soma de um número em ponto flutuante por um valor constante. Adicionalmente, a operação de truncamento (arredondamento para baixo: piso) é necessária, visto que índices da tabela são valores inteiros.

O desenvolvimento de técnicas para o acesso eficiente à *lookup table* levou a escolha desta técnica como sendo a mais adequada para se usar na arquitetura proposta.

A Figura 5.15 mostra que a arquitetura proposta para implementar a *lookup table* utiliza apenas 1 somador de números de ponto flutuante e um módulo de arredondamento de ponto flutuante para números inteiros (função piso).

Para os valores de x que não estão no domínio especificado no projeto da *lookup table*, checa-se o índice de acesso a memória ROM. Caso o valor do índice calculado seja negativo, significa que x é menor que a e então o índice correto da tabela é gerado pelo módulo de acesso à ROM. Esta posição indica um valor constante da função de ativação para os valores de $x < a$ (para a função sigmóide este valor é 0). Caso o valor seja maior do que a quantidade de posições na memória ROM (maior índice na tabela), então o valor constante da função de ativação para os valores de $x > b$ deve ser apontado na ROM (na função sigmoide este valor é 1).

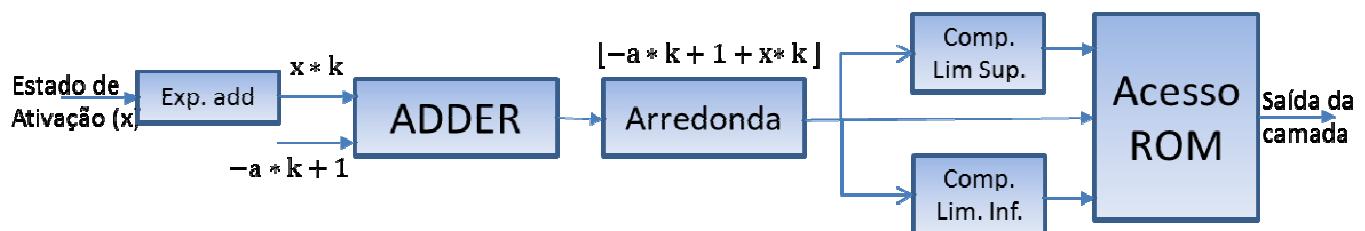


Figura 5.15 arquitetura da lookup table com complexidade de acesso O(1)

A arquitetura desenvolvida também foi implementada em pipeline, assim como a da computação do estado de ativação. A Figura 5.2, que mostra a estrutura da rede neural completa e a interconexão entre as camadas, pode ser refinada agora para detalhar melhor onde cada o módulo da computação da função de ativação se insere (Figura 5.16).

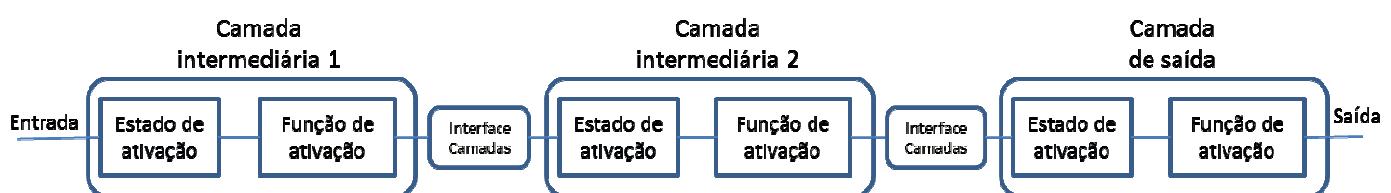


Figura 5.16 Arquitetura modular da RNA completa

Veja na Figura 5.16 que a computação do estado de ativação é seguida pelo processamento da resposta da camada cujo valor é dado pela função de ativação. As saídas de cada neurônio da mesma camada saem em sequência, um após o outro, a partir da subida de um sinal denominado **valid**. O sinal de *valid* indica que a entrada da rede está disponível. Este sinal é usado internamente na arquitetura para sincronizar a troca de dados entre as camadas e é por fim propagado até a saída, indicando que a resposta da rede está pronta.

Porém, uma camada subsequente pode não consumir os dados na mesma frequência, daí a necessidade do módulo de interface entre as camadas. A taxa de leitura das camadas é uma função do número de neurônios da camada, sendo 1 ciclo de clock para cada neurônio da camada. Uma camada com 4 neurônios consome dados na frequência de 1 entrada a cada 4 ciclos de clock. A Figura 5.17 mostra os dados que entram no caminho de dados do cálculo do estado de ativação da rede. Observe que cada entrada é necessária na taxa de 1 a cada **n** ciclos de clock (com **n** sendo o número de neurônios na camada).



Figura 5.17 Taxa de leitura de dados de entrada de uma camada com 4 neurônios

O módulo Interface Camadas (vide seção 6.2.3) armazena os resultados da camada anterior e os apresenta para a camada seguinte na frequência adequada. Adicionalmente, caso uma camada intermediária tenha um número de entradas

ímpar, este módulo é responsável por inserir a entrada -1 associada com o bias. Logo, qualquer quantidade de entradas pode ser implementada em qualquer das camadas.

A arquitetura da rede neural completa mostra a estrutura modular e regular que foi projetada para facilitar a implementação da rede a partir dos componentes desenvolvidos.

5.3 Desempenho esperado

Na computação de cada $u_i = \sum_{j=1}^{m+1} W_{ij} \cdot X_j$ com $i = 1, \dots, n$ inclui m somas de $m+1$ produtos independentes. Se for possível instanciar **m+1** multiplicadores e **m** somadores em algum dispositivo reconfigurável (após o pipeline estar cheio) o resultado da computação de U estará disponível em **n*m** ciclos de clock.

$$U = \begin{pmatrix} w_{1,1} & \cdots & w_{1,m+1} \\ \vdots & \ddots & \vdots \\ w_{n,1} & \cdots & w_{n,m+1} \end{pmatrix} * \begin{pmatrix} x_1 \\ \vdots \\ x_m \\ -1 \end{pmatrix}$$

Já na arquitetura proposta, a computação de U leva (após o pipeline estar cheio) **n * m + n** ciclos de clock, porém demanda apenas 1 multiplicador e **[log2(m) + 1]** somadores. Na verdade, quando m é ímpar a taxa de computação é de **n * (m + 1) + n**. Isto ocorre porque o bias conta como uma entrada para deixar o número de entradas par.

A explicação para essa taxa de computação reside no fato de que cada valor de entrada fica estável na entrada do multiplicador por n ciclos de clock (1 para cada peso associado àquela entrada em cada neurônio, vide Figura 5.4 e Figura 5.5), daí o fator $n*m$. Existem n neurônios, logo n respostas são esperadas uma após a outra. Daí chega-se na equação **n*m+n**. Assim que todas as entradas são mostradas ao multiplicador um novo conjunto de entradas pode ser apresentado, preservando-se a taxa de computação (mantida constante, após o preenchimento do pipeline).

Quando se utiliza mais de um caminho de dados, para uma camada, tanto a taxa de computação quanto a profundidade do pipeline são reduzidas. A taxa de computação varia dos $n*m+n$ ciclos de clock (1 caminho de dados para os n neurônios) até $2*m$ ciclos de clock (1 caminho de dados para cada neurônio). A menor taxa é $2*m$ porque uma camada com 1 entrada tem a mesma taxa que uma camada com 2 entradas. Isto é devido a adição do bias como uma entrada.

Para uma rede neural, como um todo, a taxa de computação total corresponde a menor taxa entre as suas camadas. Assim a camada mais lenta define a taxa total da rede. Outro detalhe é que a fórmula da taxa para a rede toda tem a forma de $n_k*m_k+n_s$. Ou seja, a parcela $n*m$ se refere a camada mais lenta k e o n final se refere ao número de neurônios da camada de saída da rede neural.

6. Implementação

6.1 Ponto Flutuante VS Ponto Fixo

O Instituto dos Engenheiros elétricos e eletrônicos (IEEE) padroniza as representações de ponto flutuante sob o padrão IEEE 754. A representação em ponto flutuante é similar à notação científica em que existe um número multiplicado por sua base elevada a um expoente.

O maior benefício desta representação é que ela provê vários graus de precisão baseados na escala dos números que se está usando. Por exemplo, falando em termos de nanômetros quando se relaciona a distância entre transistores em um circuito integrado.

O padrão IEEE 754 (vide Figura 6.1) define representações tanto para ponto flutuante de precisão simples (32bits) e de dupla precisão (64bits) quanto para suas versões estendidas. Um número representado em precisão simples tem: o bit de sinal, um campo de expoente, e a mantissa.

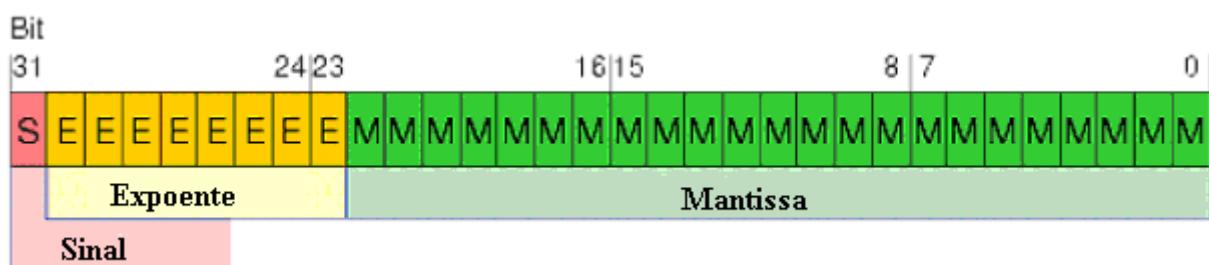


Figura 6.1 Representação Ponto Flutuante precisão simples

O expoente é um número de 8 bits resultando em um intervalo de -126 até 127. Na verdade, o expoente não está na representação típica de complemento a 2, ao invés disso, ele é enviesado, adicionando o valor 127 ao expoente desejado. Isto permite representar números com expoente negativos.

Neste padrão, a mantissa está na representação binária normalizada do número a

ser multiplicado pela base 2 elevado a potência definida no expoente.

Na notação de ponto fixo define-se um radix específico e há um número fixo de bits para representar os campos a esquerda e a direita do radix. Os bits da esquerda são chamados bits inteiros e os da direita, bits fracionários. Um formato na representação de ponto fixo pode ser visto na Figura 6.2.

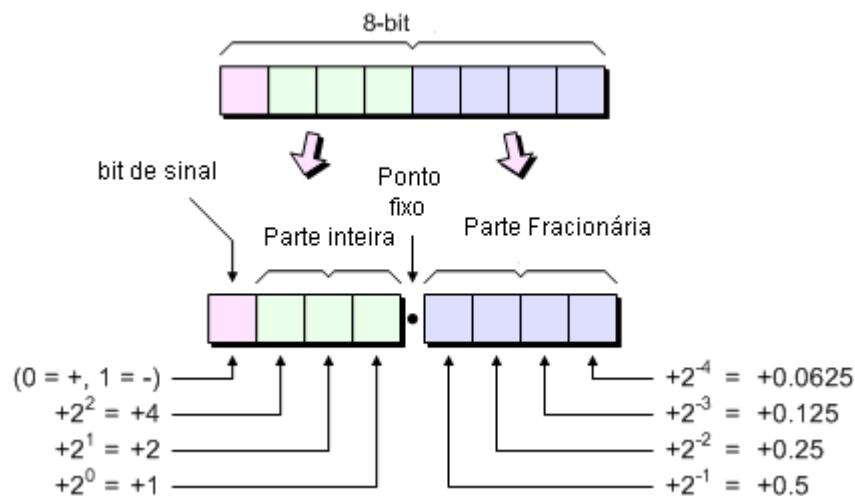


Figura 6.2 Formato representação ponto fixo

A maior vantagem em se usar ponto fixo para números reais reside no fato que números de ponto fixo aderem aos mesmos princípios da aritmética de números inteiros. Além do mais, migrar para essa representação a partir de uma arquitetura de inteiros não requer nenhuma lógica adicional.

A desvantagem do uso do ponto fixo é fortemente fundamentada no fato que os números só podem ser representados em um intervalo bastante limitado de valores. Logo, torna-se suscetível a ocorrência de overflows e underflows.

Tabela 6.1 Comparativo ponto flutuante vs ponto fixo

Ponto flutuante	Ponto fixo
Precisão	Custo do produto final
Range Dinâmico	Velocidade
Maior consumo de hw	Menor consumo de hw
Tempo de desenvolvimento	

Na implementação da arquitetura da RNA optou-se pelo uso do ponto flutuante. Nessa escolha buscou-se encontrar uma maior fidelidade do modelo das RNAs proposto em software e a precisão para se trabalhar com valores que estejam em ordem de grandeza não previamente especificados (necessário para ponto fixo). Outro ponto importante para a referida escolha foi que se levaria a uma arquitetura limitada a tarefa de classificação apenas. Visto que, o uso do ponto fixo ocasiona uma oscilação no processo de aprendizagem.

Levanto em conta a característica da arquitetura proposta, que como indicado na seção 5.1 possui consumo de somadores crescendo com o \log_2 do número de entradas, torna factível o uso de ponto flutuante mesmo com a maior utilização de recursos nas unidades de soma e multiplicação isoladamente. Na seção 7.1 o reflexo dessa escolha pode ser conferido em relatórios reais de síntese para um FPGA comercial.

A Altera dispõe de componentes de aritmética de ponto flutuante em sua biblioteca. Atualmente estão disponíveis componentes para precisão simples, precisão dupla e para precisão simples estendida. O uso de um componente disponibilizado pela fabricante do FPGA utilizado agregou maior confiabilidade ao fluxo de desenvolvimento da arquitetura visto que esses componentes já estão validados.

Usou-se os componentes: somador/subtrator (ALTFP_ADD_SUB), multiplicador (ALTFP_MULT) ambos de precisão simples padrão IEEE 754. A interface dos componentes é apresentada na Figura 6.3.

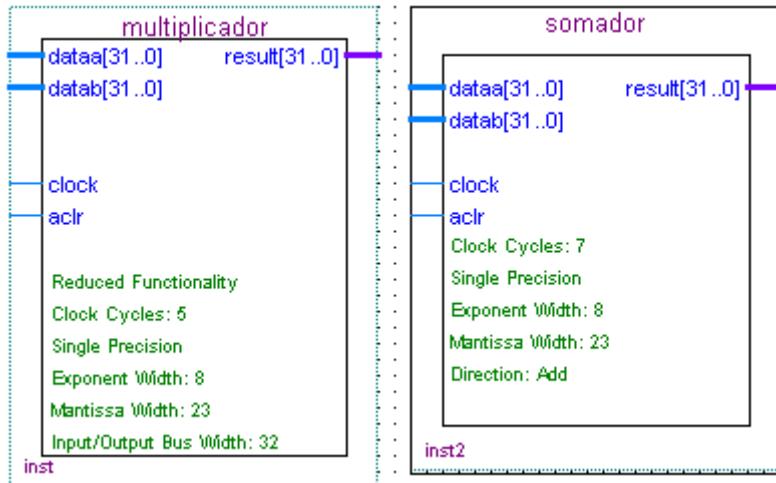


Figura 6.3 Blocos dos componentes aritméticos de ponto flutuante

A profundidade do pipeline do somador possui até 14 ciclos, enquanto a do multiplicador tem no máximo 11. A escolha do máximo tamanho de pipeline para os componentes se reflete no aumento da f_{max} de operação desses circuitos. Este fato se deve a quebra das operações em mais estágios de pipeline reduzindo o caminho crítico em cada estágio. Assim, nessa configuração a frequência de operação dos componentes é de 300mhz.

6.2 Módulos de hardware

Os módulos cuja arquitetura foi apresentada nas seções 5.1 e 5.2 foram implementados utilizando a linguagem de descrição de hardware SystemVerilog (subconjunto sintetizável). Quando necessárias, as máquinas de estados dos módulos foram implementadas utilizando o padrão Moore. Por este padrão toda lógica de troca de estado deve ser feita combinacionalmente e apenas as saídas tem transições sequenciais (no clock). Assim as saídas são em função do estado da máquina de estados. Este padrão de codificação pode produzir mais estados do que o padrão Mealy, porém o padrão Moore é mais indicado por atingir maiores frequências de operação em FPGAs (Sutherland, et al., 2006).

6.2.1 Mux_bias

Este módulo compõe o caminho de dado do processamento do estado de ativação para algumas camadas. Ele é necessário quando é preciso inserir o bias ou algum carry em algum somador (vide Figura 5.6).

Como todo MUX, ele chaveia entre a entrada principal `data_in_somador` e a alternativa `data_in_bias` (ou `datain_carry`). A palavra de seleção do mux é a entrada **valid** (o mesmo sinal que indica que o primeiro dado da entrada é valido).

Quando o pulso de `valid` é ativado o módulo já está chaveado para a entrada `data_in_somador`, neste momento ele chaveia para `data_in_bias` e começa a contar até atingir a quantidade de neurônios (quantidade de dados de bias). Após terminar a contagem novamente a entrada `data_in_somador` é comutada na saída.

Parâmetro: n (quantidade de dados a serem inseridos após o valid)

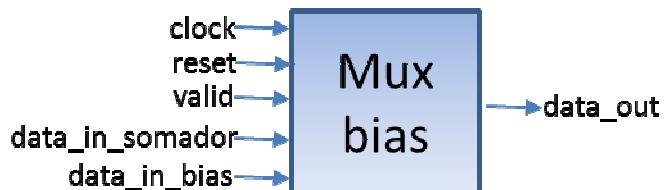


Figura 6.4 Mux bias

O parâmetro 'n' permite que a implementação do módulo seja genérica, assim, a partir de uma mesma descrição HDL de um componente pode-se instanciar módulos com características bem diversas. Exemplo: o módulo `mux_bias` pode ser instanciado para qualquer valor de `n` (que indica a quantidade de dados após a subida do `valid`). A instanciação do módulo na linguagem SystemVerilog, o mapeamento das portas do módulo e a definição do parâmetro `n` (através da diretiva `defparam`) é mostrada a seguir:

```
mux_bias mux_bias_i(.dataout(saida_mux_bias),
                     .datain_somador(saida_somador1),
```

```

.datain_bias(saida_weight_bias),
.clk(clock),
.reset(reset),
.valid(saida_sr_valid_mux_bias));

defparam mux_bias_i.n = 10;

```

Na Figura 6.5 é mostrada a máquina de estados da operação do módulo mux_bias. No estado `wait_valid` é aguardada a subida do sinal `valid` (que indica neste caso que o mux precisa comutar de `datain_somador` para `datain_bias`). Quando da subida do `valid` ocorre a transição para o estado `count`. O estado `count` é responsável por contar até o valor do parâmetro `n` (que é igual ao número de neurônios da camada atual) e a partir daí ocorre a transição para o estado original de `wait_valid`.

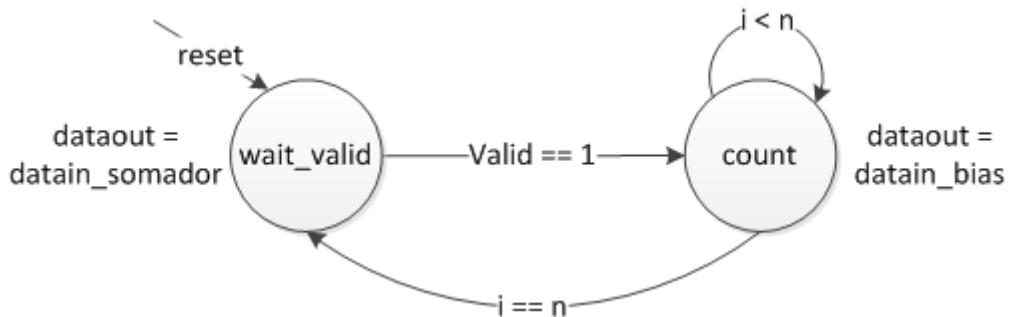


Figura 6.5 Máquina de estados do módulo mux bias.

6.2.2 Shift Register

O shift-Register ou registrador de deslocamento é um componente visualizado como uma sequência de registradores interconectados que servem para inserir atrasos numa linha de dados.

Na arquitetura proposta esse componente é utilizado tanto para atrasar os resultados parciais de saída de um somador para entrar no próximo, quanto para retardar o sinal de `valid`, em diversas situações.

Foi necessário implementá-lo de forma genérica e parametrizada visto que a entrada

a ser atrasada `data_in` pode ser instanciada com largura de bits diversas (para números no formato de ponto flutuante a entrada seria de 32 bits, para o bit de valid a entrada será de apenas 1 bit).

Parâmetros: profundidade (depth - quantos ciclos a entrada `data_in` levará para chegar a `data_out`); largura (width - quantidade de bits de `data_in`)



Figura 6.6 Shift-register

Internamente o módulo não possui nenhuma fsm (máquina de estados finita), visto que este módulo não precisa de controle e sim de caminho de dados. Em Systemverilog o trecho de código abaixo declara uma memória de largura dada pelo parâmetro `width` e tamanho dado pelo parâmetro `depth`.

```
logic[width-1:0] storage[1:depth-1];
```

As ferramentas de síntese inferem o comportamento do shift-register através do laço que atualiza as posições do vetor de armazenamento `storage`.

```
repeat(depth-2)begin  
    storage[i+1]<=storage[i];  
    i++;  
end
```

A saída do módulo é o valor armazenado pela posição `storage[depth-1]`, a última posição do vetor.

6.2.3 Interface_Camadas

Este componente é responsável pela ligação entre duas camadas de neurônios da

rede neural na arquitetura proposta (vide Figura 5.16).

A entrada *valid_in* é o sinal valid de saída da camada anterior e a entrada *data_in* a linha de dados na representação ponto flutuante (32 bits) correspondente.

As saídas da camada anterior são recebidas sequencialmente, armazenados e apresentados na saída *data_out* que é a entrada da camada seguinte. Um atraso no sinal de *valid* entre as duas camadas é necessário devido ao atraso da operação deste módulo.

Parâmetros: entradas (inputs) – igual à quantidade de neurônios da camada anterior; neurônios (neurons) – quantidade de neurônios da próxima camada.



Figura 6.7 Interface Camadas

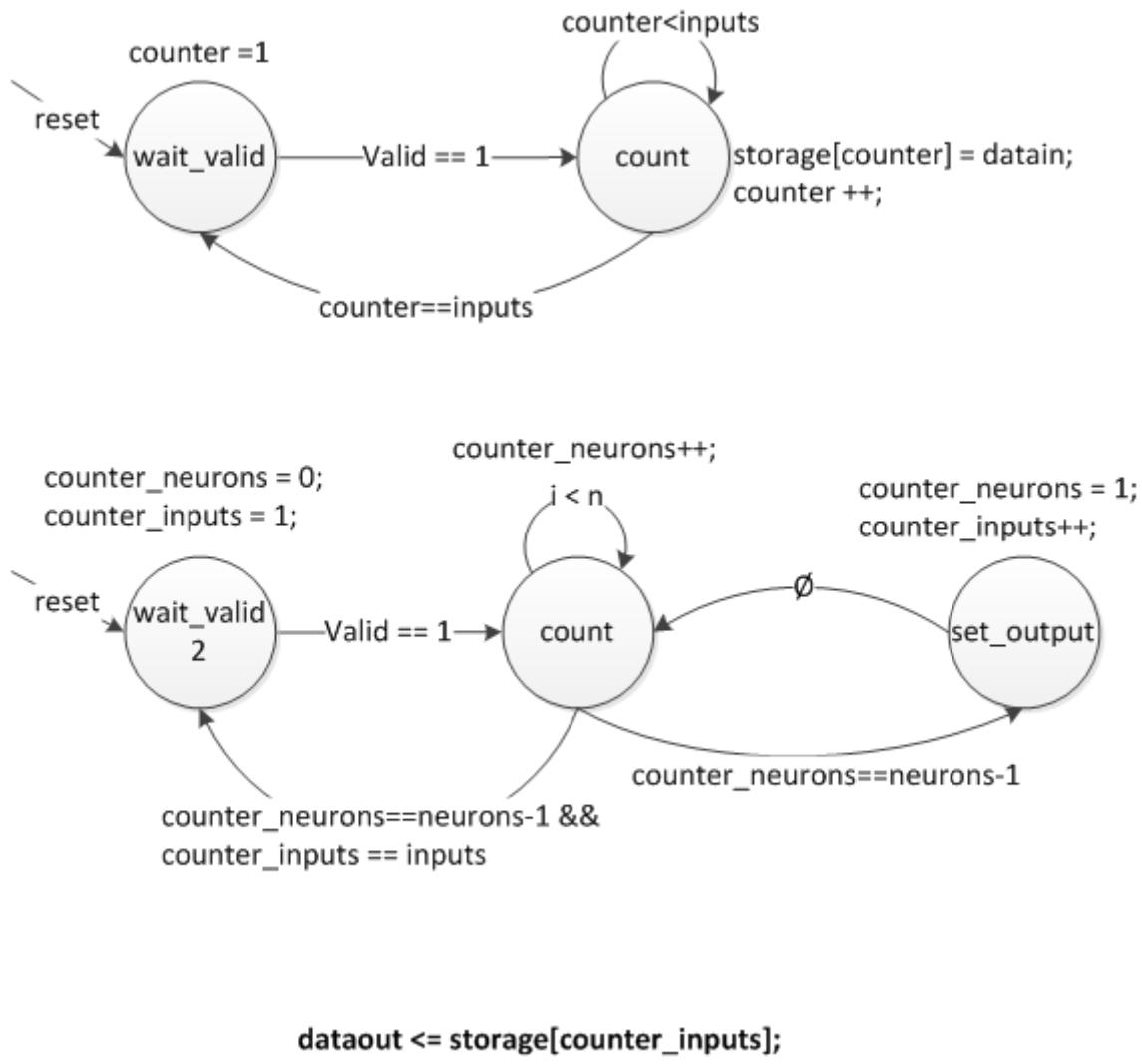


Figura 6.8 FSMs do módulo interface_camadas

Na Figura 6.8 é mostrada as máquinas de estados que implementam a funcionalidade do módulo. A primeira FSM tem o papel de ler os dados da camada anterior que vêm em `datain` na subida do sinal `valid` e armazená-los em `storage` (vetor de armazenamento interno cujo tamanho é igual ao parâmetro `inputs`).

Novamente no estado `wait_valid` é aguardada a subida do sinal `valid` e em `count` é incrementado o valor de `counter_neurons` (variável de controle de valor máximo igual ao parâmetro `neurons`). Assim, a máquina precisa aguardar uma quantidade de ciclos igual a `neurons` com um valor da saída da camada anterior estável para a entrada da próxima camada. Quando `counter_neurons` atinge o valor de `neurons` o valor de saída deve ser atualizado incrementando-se o índice `counter_inputs` no

estado *set_output*. Do estado *set_output* se retorna-se, no próximo ciclo, para o estado *count* e novo ciclo é reiniciado. Este procedimento continua até que todas as saídas da camada anterior sejam apresentadas à camada posterior. Quando isso acontece retorna-se ao estado *wait_valid* para se esperar um novo conjunto de saídas.

6.2.4 Weight_SR

Os pesos dos neurônios artificiais são armazenados dentro do FPGA através do módulo *Weight_SR*. Este módulo, a partir do pulso de entrada *valid*, fornece na saída, *data_out*, os dados armazenados de forma sequencial (coluna a coluna, da esquerda para a direita). Assim os dados de entrada da RNA podem ser multiplicados pelos pesos da matriz *W* apresentada na seção 5.1 (vide Figura 5.4).

Parâmetros: n – número de neurônios da camada; m – número de entradas da camada. *weights* – vetor dos pesos armazenados.

A matriz armazenada no módulo pode ter dimensão $n \times m$ (caso m, par) ou $n \times m+1$ (caso m, ímpar). Outra instância deste módulo é usada para armazenar apenas os valores do *bias* da camada para o caso de m ser par.



Figura 6.9 Weight-SR

A fsm do *weight_SR* é similar a do *mux_bias*. Existe um estado de espera do sinal *valid* (em que a saída é 0) e outro estado de contagem que incrementa o contador e coloca na saída, a cada ciclo, sequencialmente os valores armazenados em *weights*.

6.2.5 *Lookup_table*

Este módulo é a implementação da descrição arquitetural da Figura 5.15. A implementação em pipeline com 23 estágios concatena 1 somador, 1 arredondador (ponto flutuante para inteiro), 2 comparadores de ponto flutuante. A memória ROM não é explicitamente implementada, e sim implicitamente em SystemVerilog, declarando um array estático de elementos como em:

```
logic [31:0] logsig [0:1280+2] = '{/*valores da lookup table*/};
```

As ferramentas se síntese de hardware sintetizam este tipo de estrutura como uma memória interna no FPGA. Qualquer função de ativação com qualquer precisão variável pode ser implementada utilizando esse módulo. Basta apenas modificar a declaração do vetor acima com os valores da função pretendida.

Não existe nenhuma máquina des estados nesse módulo. A Figura 6.10 mostra os sinais de entrada clock, o valor do estado de ativação (x) e a saída da camada, y . O sinal y é o resultado a aplicação de x na função de ativação escolhida.

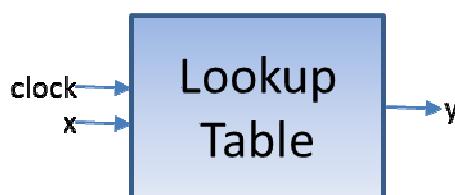


Figura 6.10 Lookup table

6.3 Geração automática da RNA em linguagem HDL

A geração de uma descrição em Systemverilog que implemente uma RNA incluiu o desenvolvimento de uma ferramenta de geração de código e a definição de um formato intermediário para facilitar a integração do gerador e das ferramentas CAD de RNAs. Neste trabalho foi implementada a geração de componentes de hardware em Systemverilog (arquivos ***.sv**), sendo o gerador desenvolvido na linguagem C/C++ e voltado para o ambiente Linux.

A técnica proposta para a geração da RNA em Systemverilog para a arquitetura proposta neste trabalho inclui o desenvolvimento de componentes parametrizáveis (mostrados na seção 6.2), cujos parâmetros são calculados dinamicamente em função da estrutura da RNA. Os componentes e a arquitetura desenvolvida neste trabalho foram pensados de forma a permitir sua geração automática.

A função principal do gerador de RNAs é gerar o módulo de mais alto nível (a rede neural em si). O nome da RNA pode ser especificado através de parâmetro, assim como um número que indica a quantidade de caminhos de dados na 1^a camada intermediária.

Esse parâmetro indica o grau de paralelismo requerido para a referida camada. Exemplo, se esse parâmetro for igual a 2, isto indica que haverá 2 caminhos de dados naquela camada e `nr_neurônios / 2` neurônios em cada caminho de dados.

Na Figura 6.11 pode-se ver o fluxo de geração da função principal que gera o top level da RNA. Onde pode-se identificar as fases principais:

- Gerar Cabeçalho – escreve a declaração do módulo da RNA e a lista de portas de entrada e saída.
- A leitura da topologia da RNA, exportada pelo *script* rodando na ferramenta CAD Matlab.
- While $i \leq nr_qtde_camadas$ – dentro desse laço ocorrem as chamadas da função auxiliar de geração das camadas. Caso i seja 1 então se trata da 1^a camada intermediária, caso i seja igual ao número de camadas então se trata

da camada de saída. É importante saber qual camada se está lidando porque diferentes conexões são feitas (por exemplo, na camada de saída não precisa instanciar o módulo de interface entre camadas; na 1ª intermediária as entradas são as entradas da rede enquanto nas demais as entradas são as saídas das outras camadas).

Na geração da 1ª camada intermediária ocorre também a instanciação dos vários caminhos de dados que podem compor esta camada. Na verdade um outro *loop* existe para esse propósito.

Na implementação do gerador da arquitetura proposta está limitado o uso de vários caminhos de dados apenas na 1ª camada intermediária. Porém, esse limitante foi apenas introduzido para diminuir a complexidade do gerador e nada tem haver com uma limitação da arquitetura.

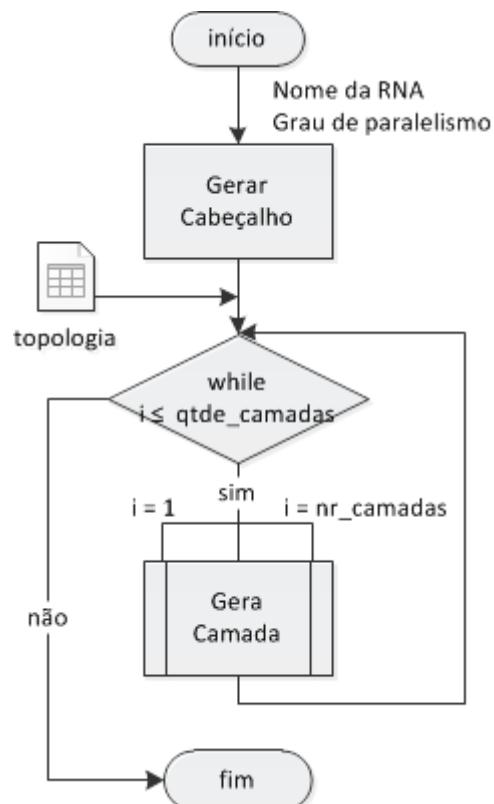


Figura 6.11 Algoritmo da geração do top level da RNA

A função auxiliar de geração das camadas, chamada pelo método principal, cria os módulos de cada camada separadamente a partir dos parâmetros fornecidos na topologia da rede quais sejam: como o número de neurônios e o número de entradas na camada.

A Figura 6.12 ilustra o processo de geração de código. As fases principais são:

- Inserir cabeçalho - Assim como na função principal, a declaração do módulo em Systemverilog exige a expressão das portas de entrada e de saída. Pela arquitetura proposta apenas um multiplicador precisa ser utilizado em cada caminho de dados. Assim, este pode ser gerado nesta fase. O primeiro somador difere dos outros pelas suas conexões de entrada, na medida em que apenas este se conecta com a saída do multiplicador.
- Os arquivos que contêm os pesos são lidos. Esses arquivos são gerados pelo *script* na ferramenta CAD de RNA e são listas dos pesos separados por camadas.
- While $n \neq 1$ – o laço principal testa se a variável de controle do laço n é diferente de 1. O valor de n inicial é igual ao número de entradas da camada (ou $n+1$ se n for ímpar).
 - Se n for divisível por 2 ($n \% 2 = 0$, operador mod de C/C++, resto da divisão inteira) então adiciona-se aos pares o módulo shift_register e um somador. N é atualizado com $n = n/2$;
 - Senão, se o bias não foi inserido ainda, então instancia-se o módulo weight_sr, é lido do arquivo os valores do bias, instancia-se um mux_bias e por fim um par somador e um módulo shift_register. N é atualizado com $n = (n+1)/2$;
 - Senão, se não há carry (de outro nível) para ser inserido, então agora $carry = 1$ e armazena-se o contexto de onde o carry vem. Adiciona-se um par somador e módulo shift_register. N é atualizado com $n = (n-1)/2$;
 - Senão, se há carry, então o carry precisa ser adicionado. Realiza-se os

cálculos dos atrasos para se instanciar o shift-register da linha de dados do carry, instancia-se um mux e um par somador e módulo shift_register. N é atualizado com $n = (n+1)/2$;

- Quando $n = 1$ tem-se o fim do laço, e então podem ocorrer três situações: *Bias* não inserido ainda; *Carry* precisa ser inserido; Nem *bias* nem *carry* pendentes. Nas três situações um novo par somador e shift_register é inserido. As demais operações realizadas em cada um desses casos são similares as de dentro do laço.
- Por fim, é instanciada a *lookup table* da camada e o *shift_register* que dá o atraso do sinal de *valid* recebido na entrada da camada.

O resultado da fase da geração são os arquivos Systemverilog (um para cada camada e outro para a rede neural) que podem ser diretamente prototipados em FPGA ou integrados a outros sistemas.

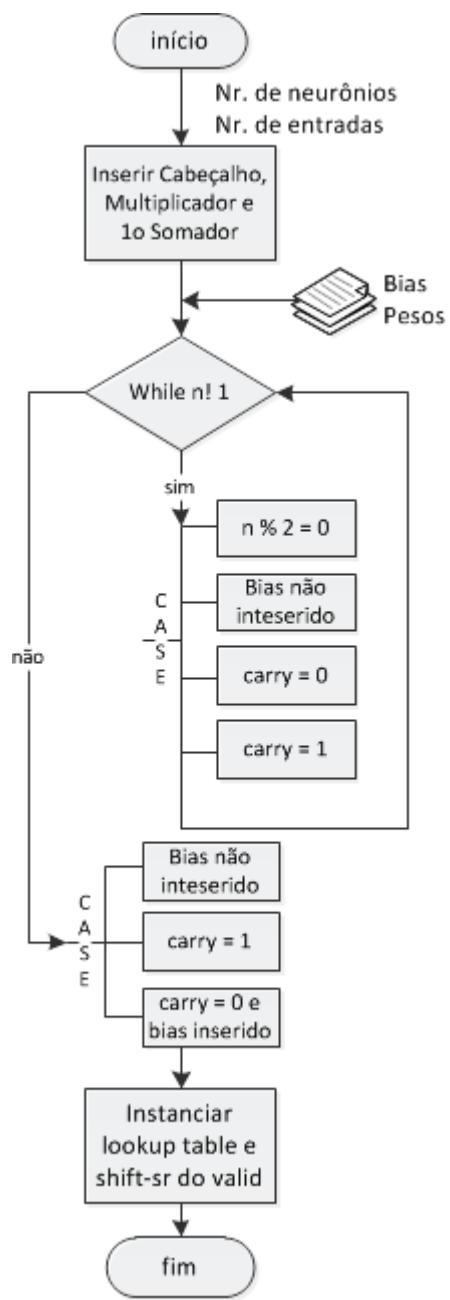


Figura 6.12 Algoritmo de geração das camadas

7. Análise de resultados

A fim de validar a arquitetura proposta neste trabalho, utilizou-se três problemas para os quais soluções baseadas em redes neurais foram implementadas. Esses problemas são representados como bases de dados com características pré-definidas extraídas do mundo real. Dois deles são bases de dados amplamente utilizadas na área de reconhecimento de padrões, classificação e aprendizagem de máquina (UCI). São as bases:

- IRIS – três tipos da flor íris (Íris Setosa, Íris Versicolour e Íris Virginica) representam as três classes do problema. Quatro características foram extraídas: Largura e comprimento em centímetros da pétala e da sépala (cálice) das flores analisadas. São ao todo 150 padrões na base, sendo 50 exemplos de cada classe. Todos os valores utilizados são reais. A rede neural desenvolvida deve fornecer, a partir dos quatro parâmetros de entrada, o tipo da flor.
- Semeion – 1593 dígitos (de 0 a 9), manuscritos por 80 pessoas, foram digitalizados em escala de cinza (256 tons de cinza), enquadrados numa matriz de 16x16 pontos e binarizados (utilizando uma técnica de limiar). Assim cada dígito é um vetor de 256 pixels indicando a presença ou ausência do pixel neste dígito. Esse problema apresenta a entrada dos 256 pixels e a saída da classificação deste entre as 10 classes (dígitos de 0 a 9).

O outro problema para o qual foi proposta uma rede neural foi um problema de aproximação da função transcendental descrita pela equação:

$$y = \frac{1}{e^{0.03x}} * \text{sen}(0.003x)$$

Para este problema um valor real de x (pertencente a um domínio limitado) é a entrada da rede e sua saída o valor correspondente para y . A Figura 7.1 mostra o gráfico de y em função de x .

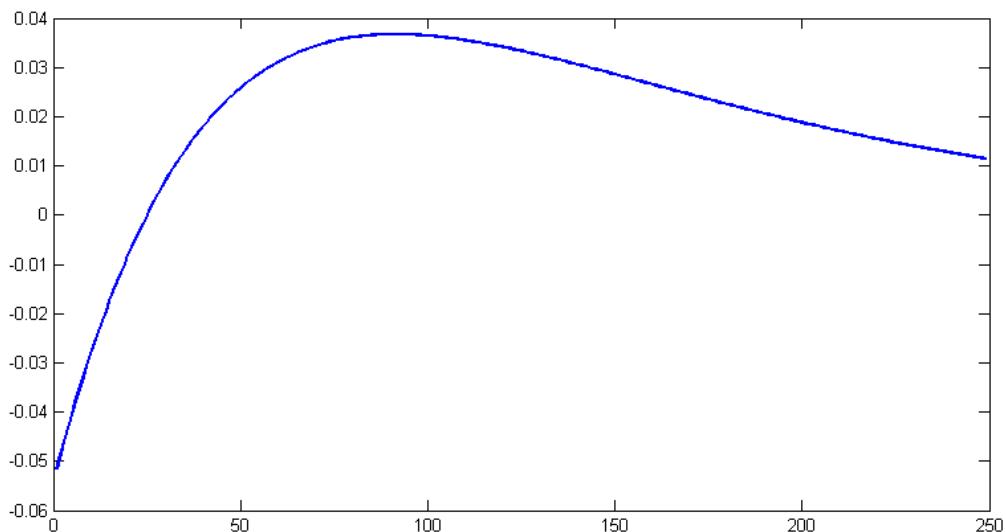


Figura 7.1 Gráfico da função transcendental y

Para um dos três problemas, mencionados anteriormente, foi proposta uma RNA desenvolvida no Matlabtm (seguindo o fluxo de projeto mostrado na seção 2.4) utilizando o toolbox de redes neurais. As topologias de cada uma das RNAs podem ser vistas na tabela a seguir.

Tabela 7.1 Características das três redes implementadas

	Vetores de entrada	Topologia	Nr. de Neurônios	Tipo Problema
Senóide	249	1-5-1	6	Aprox. Função
Iris	150	4-8-3-3	14	Classificação
Semeion	1593	256-10-10	20	Classificação

Como pode ser visto na Tabela 7.1, As redes implementadas tem respectivamente 1, 4, e 256 entradas; e 1, 3 e 10 saídas. O número de saídas das redes é igual ao número de classes do problema no intuito de melhorar a separação entre as classes (garantindo uma melhor taxa de acerto). Assim a técnica do vencedor leva tudo (*winner takes all*) foi aplicada, fazendo da saída, com maior valor, a vencedora.

Como medida de desempenho das redes implementadas, é mostrado a curva ROC (*Receiver Operator Characteristics*) para os dois problemas de classificação e o gráfico do erro das saídas da rede que aproxima a função senóide.

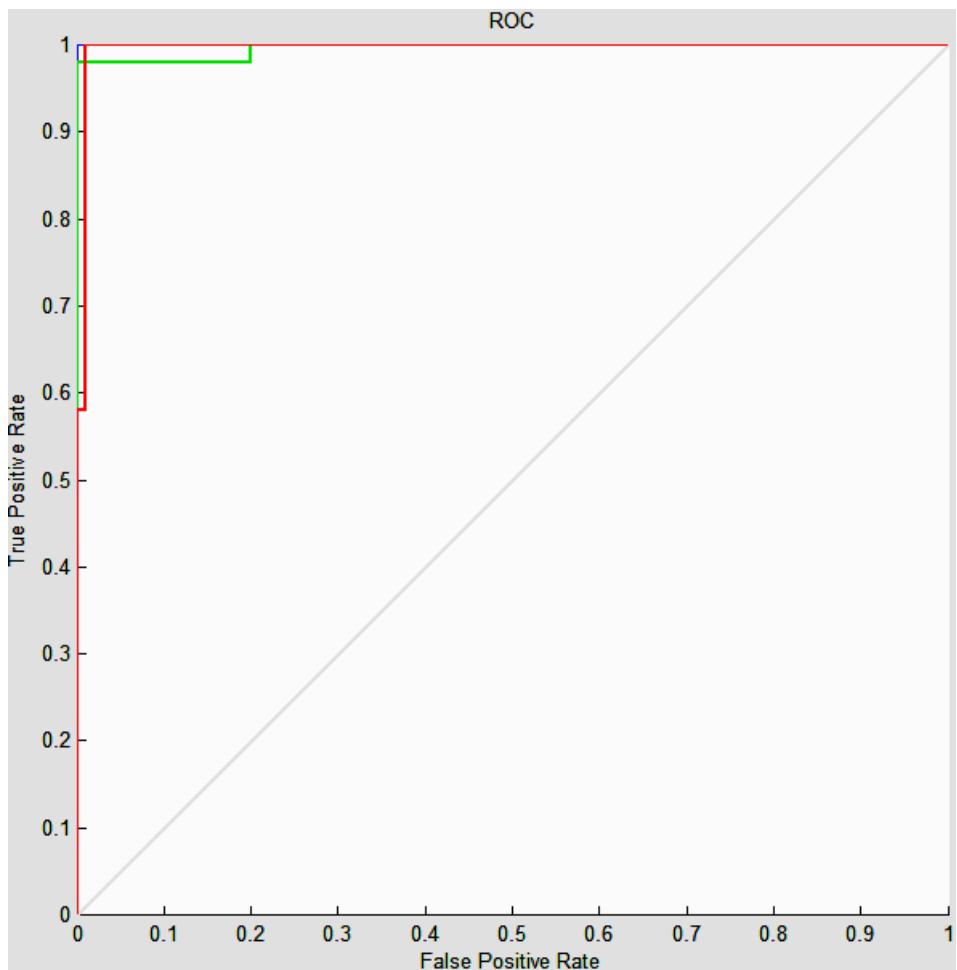


Figura 7.2 Curva ROC para a rede Íris

Pela curva ROC, um classificador tem melhor desempenho quanto mais sua curva se aproxima do canto superior esquerdo (maior área sob a curva) e a reta (45°) central é o desempenho do classificador aleatório (acerto de 50%). Nos gráficos da Figura 7.2 e da Figura 7.3, cada curva representa a curva ROC para cada uma das classes de cada problema (3 classes no Íris e 10 no Semeion). Observa-se que as curvas estão bem próximas do canto superior esquerdo e assim as redes implementadas para os problemas possuem bom desempenho.

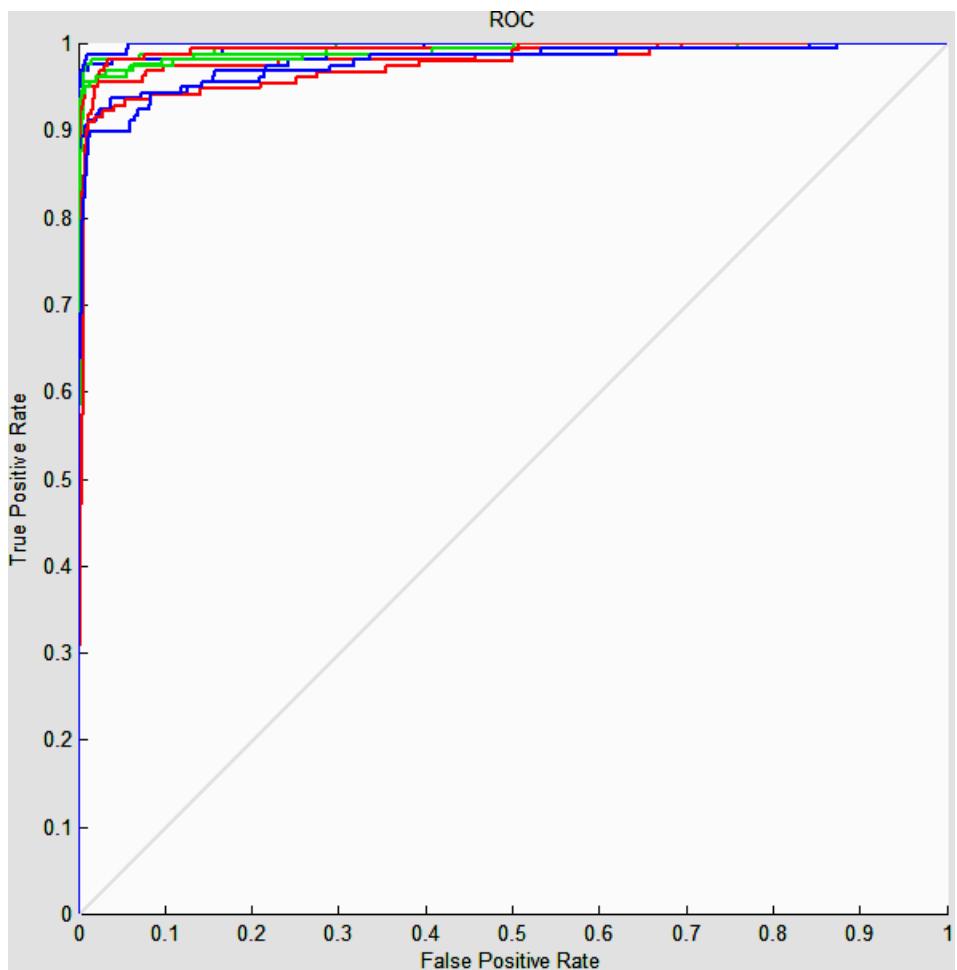


Figura 7.3 Curva ROC para a rede Semeion

A Figura 7.4 mostra o gráfico comparativo entre as saídas da função senóide e da rede, bem como a diferença entre elas. Observa-se que a rede neural aproxima bem a função no intervalo [30,250], porém no intervalo negativo a rede implementada não consegue acompanhar a função aproximada porque a função de ativação sigmoide tem valores no intervalo [0,1]. Nesse caso poderia ser utilizada a função tangente hiperbólica que tem valores de saída entre [-1,1].

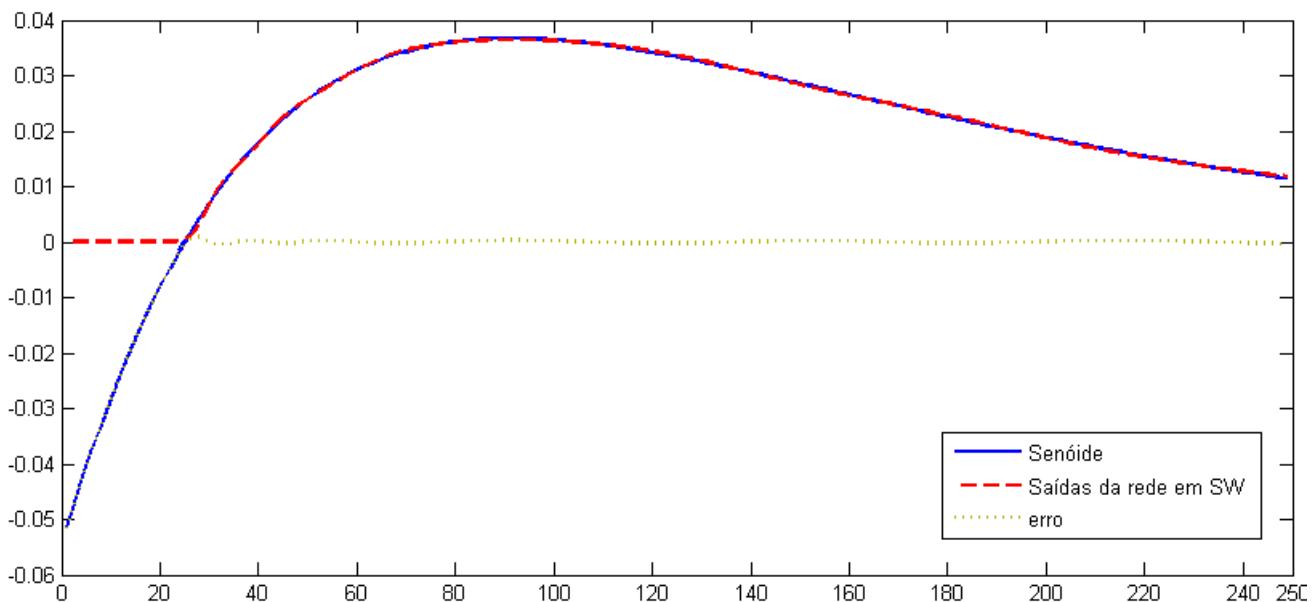


Figura 7.4 Gráfico do erro das saídas da rede que aproxima a senóide

7.1 Área

As redes neurais descritas no início deste capítulo foram implementadas em hardware seguindo a descrição da arquitetura proposta neste trabalho. Os pesos e bias foram extraídos do projeto da rede em Matlab. Foi utilizado o gerador automático de redes neurais proposto neste trabalho e as ferramentas de projeto de hardware Synplify Premier da Synopsys® e Quartus II 9.1 da Altera® para síntese e a ferramenta Questa da Mentreo Graphics para simulação pós-síntese com temporização.

Em todas as três redes neurais foi usado o módulo *lookup table* instanciado com a precisão de $k = 128$ (intervalo de 0.0078125) e range de [-5,5], i.e. uma *lookup table* com para a sigmoide com 1281 posições em cada uma das camadas das redes.

Na Tabela 7.2 pode-se observar a utilização de recursos para o FPGA da família Stratix III EP3SL50F484C2 fabricado pela Altera. Vê-se que a arquitetura proposta permitiu a implementação de redes com grande quantidade de entradas e neurônios sem que isso resultasse numa grande utilização de recursos de hardware. Dessa forma, o uso de módulos aritméticos de ponto flutuante pode ser feito e mesmo

assim ocupou-se apenas 30% de lógica do menor FPGA da família Stratix III.

Tabela 7.2 Utilização dos Recursos de Hardware

	Neurô-nios	Adiciona-dores	ALUTs	Registradores	Bits de Memória	Blocos de DSP
Senóide	6	4	4591 (12%)	5294 (14%)	116405 (2%)	8 (2%)
Íris	14	9	8782 (23%)	9791 (26%)	181901 (3%)	12 (3%)
Semeion	20	12	11297 (30%)	10967 (29%)	521847 (10%)	8 (2%)

Como pode ser visto na tabela o recurso mais utilizado é ALUTs, deixando o uso de memória bastante baixo. Isso quer dizer que se pode implementar aproximações da função de ativação, através das lookup tables, com precisões bem maiores.

7.2 Desempenho

O desempenho das três redes implementadas em FPGA, com sua equivalente implementação em software (linguagem C), foi comparado. Para a implementação em software, foi utilizado um servidor HP processador Xeon E5310 (1.6Ghz) quadricore, com 8GB de memória, rodando o SO Debian 64 bits e o compilador GCC. Os tempos de execução do hardware, do software e o ganho de desempenho (*speed-up*) são mostrados na Tabela 7.3.

Como pode ser visto na tabela existem três informações de desempenho para a rede Iris. Elas diferem apenas do número de exemplos rodados (150, 300 e 600). Nesse caso os tempos de execução indicam que o *speed-up* não varia com o crescimento do número de exemplos.

Outro experimento diz respeito ao uso da exploração do paralelismo feita pelas redes Semeion2 e 3. Nessas redes foram implementadas as replicações dos caminhos de

dados para a 1^a camada escondida. Ficando as redes Semeion, Semeion2 e Semeion3 com 1, 5, 10 caminhos de dados na 1^a camada escondida, respectivamente.

Todas as redes implementadas obtiveram fmax igual ou maior que 300MHz (igual a frequência máxima dos componentes de ponto flutuante utilizados da biblioteca da Altera). Isso indica que o crescimento da rede em número de camadas e de entradas não afetou o desempenho do circuito gerado. Não é necessário projetar redes com mais do que 2 camadas intermediárias, visto que estas já podem aproximar qualquer função (vide capítulo 2).

Tabela 7.3 Comparativo de desempenho HW - SW

	Exemplos	Topologia	Neurônios	Sw (ms)	Hw (ms)	Speed-up
Senóide	249	1-5-1	6	0.345	0.010137	34.03
Iris	150	4-8-3-3	14	0.517	0.019463	26.56
Iris2	300	4-8-3-3	14	1.0	0.037963	26.34
Iris3	600	4-8-3-3	14	2.0	0.074963	26.67
Semeion	1593	256-10-10	20	50.07	13.605	3.68
Semeion2	1593	256-10-10	20	50.07	6.808	8.10
Semeion3	1593	256-10-10	20	50.07	1.371	36.52

7.3 Precisão

Quanto à precisão dos dados computados pelas RNAs em FPGA, a Tabela 7.4 mostra os erros médio, máximo e o SSE (Sum Squared Error – soma dos erros quadráticos) para todos os exemplos de cada rede. Nesse caso observa-se que a ordem do SSE para a rede Semeion é maior do que nas outras redes. Isso parece intuitivo porque a rede Semeion tem bem mais exemplos do que as outras duas.

Os resultados da Tabela 7.4 dizem respeito aos resultados da camada de saída, ou seja, correspondem às saídas das redes, já considerados os efeitos de propagação

dos erros entre as camadas intermediárias.

A única fonte de erro da implementação em hardware é a aproximação das funções pela *lookup table*, já que as computações dos estados de ativação foram todas feitas na notação de ponto flutuante, da mesma forma que são computados os estados de ativação nas implementações em software.

Tabela 7.4 Erros das redes implementadas

	Erro médio	Erro máximo	SSE	Camadas
Semeion	6.949×10^{-4}	0.0241	0.0372	2
Íris	1.507×10^{-4}	0.0021	2.4×10^{-5}	3
Senóide	1.553×10^{-4}	0.0051	4.9×10^{-5}	2

Uma análise importante é saber se os erros numéricos, mesmo que pequenos, introduzem erros na classificação da rede. Isto é, se degradam o desempenho da rede. Utilizou-se a curva ROC como medida de comparação. A Figura 7.5 e a Figura 7.6 mostram essa comparação para as redes Iris e Semeion, respectivamente. Veja que é pequena a diferença entre as duas curvas e não houve degradação do desempenho (área sob a curva ROC) das redes implementadas em hardware.

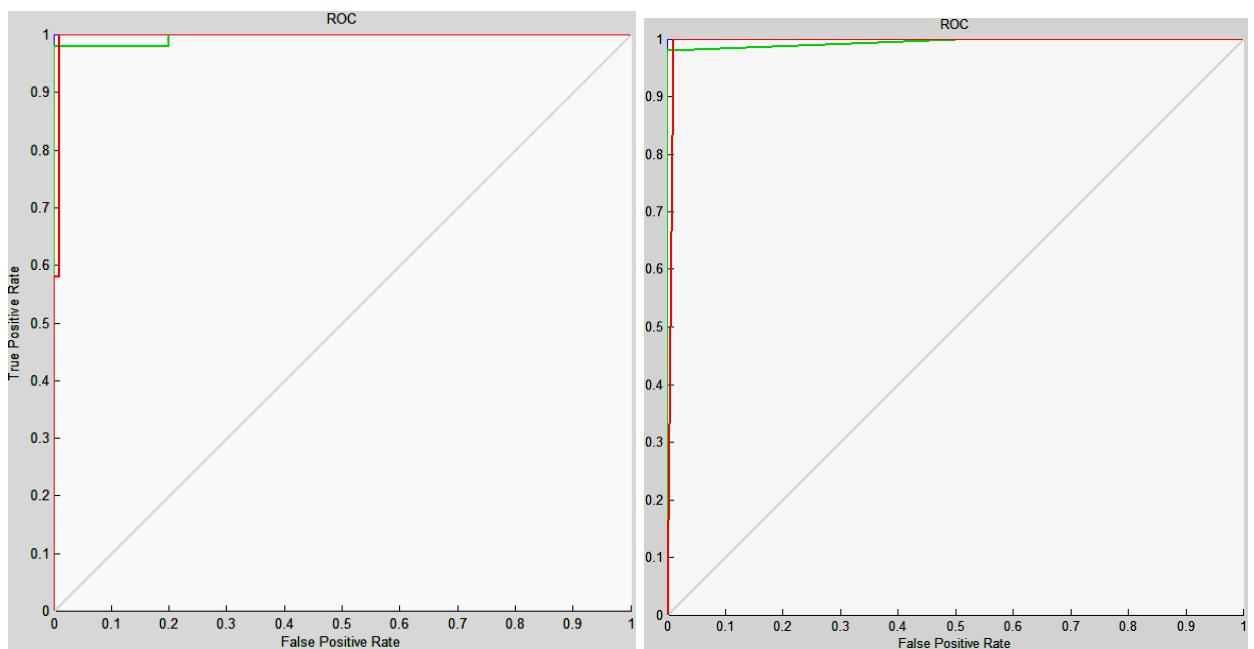


Figura 7.5 Comparativo entre as curvas ROC das rede Iris em SW e em HW(direita)

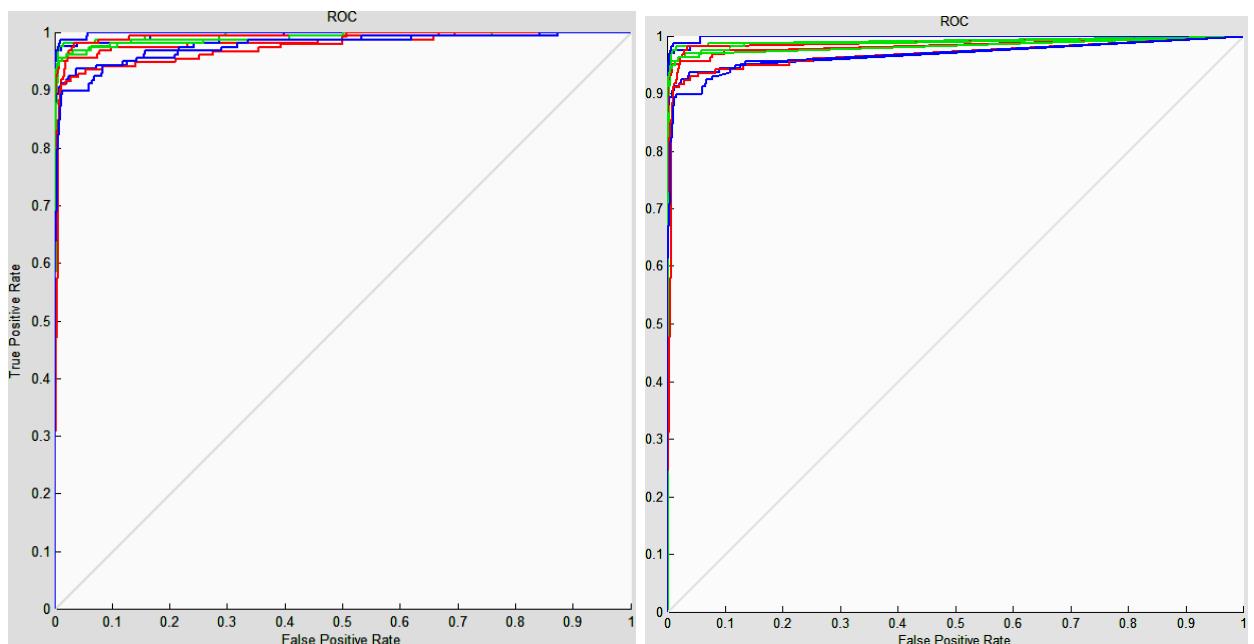


Figura 7.6 Comparativo entre as curvas ROC das rede Semeion em SW e em HW(direita)

Comparando a precisão das redes implementadas em software e em hardware na Figura 7.7 pode-se ver que a rede em hardware aproxima bem a função objetivo.

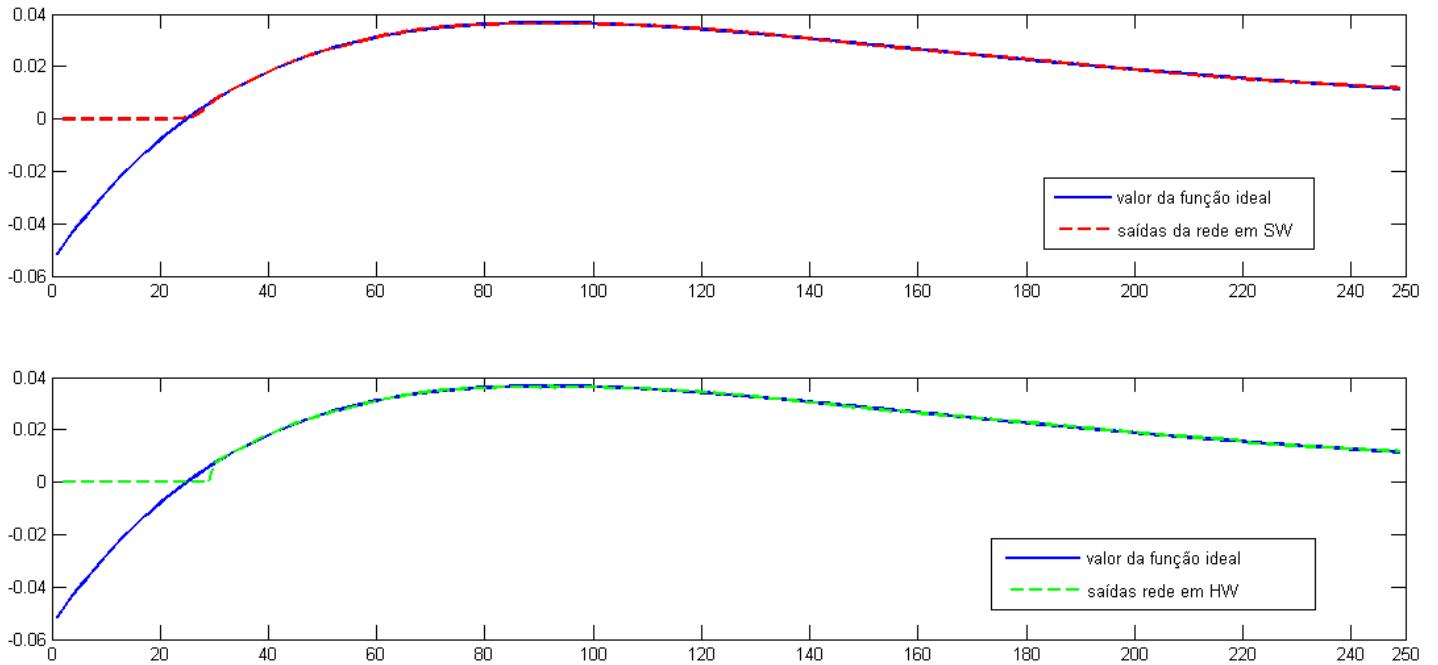


Figura 7.7 Gráfico da função senóide comparados com a implementação em SW(acima) e em HW

O único ponto que se afasta mais da senóide original, na rede em hardware, é por volta de $x = 30$. Isso deve-se à precisão empregada na *lookup table* (intervalos de 0.0078125) que, neste caso, foi menor do que o necessário. Caso a solução em hardware necessite ser mais precisa, então a solução seria utilizar uma *lookup table* com mais valores para uma maior resolução.

8. Conclusões e trabalhos futuros

Conclui-se este trabalho com uma arquitetura para implementação de RNAs MLP validadas em simulações pós-síntese com checagem temporal para FPGAs. A arquitetura foi projetada visando sua geração automática em hardware, permitindo que seus componentes possam ser implementados genericamente, na linguagem Systemverilog, através de parâmetros.

Um gerador automático também foi implementado para prover maior suporte ao projeto de RNAs em FPGA, minimizando o tempo de projeto, ao integrar a ferramenta CAD de RNAs Matlab ao gerador automático de hardware. E ainda permitindo a integração de outras ferramentas CAD de RNAs disponíveis.

O gerador foi escrito na linguagem C++ e validado no ambiente Linux. Ele gera o *top level* da RNA, suas camadas e interfaces entre elas também na linguagem Systemverilog.

Três exemplos de RNAs foram projetadas no Matlab e então implementadas segundo a descrição da arquitetura, sendo dois problemas de classificação de padrões e um de aproximação de funções. O erros numéricos das saídas foram comparados com as respostas fornecidas pela implementação em software (C++) das redes. Não houve detecção de redução da taxa de acertos, bem como redução da área sob a curva ROC.

O principal trabalho futuro seria a extensão deste trabalho para englobar outros tipos de RNAs como redes SOM, RBF, entre outras. As características dessas outras RNAs são bem diversas das MLP, porém uma implementação em hardware, igualmente, beneficiaria com um aumento no desempenho e no uso dessas RNAs em aplicações embarcadas.

A arquitetura desenvolvida neste trabalho foi apresentada no ICECS10 (17th IEEE International Conference on Electronics, Circuits, and Systems) realizado de 12 a 15 de Dezembro de 2010, na cidade de Atenas, Grécia sob o título "A high performance

full pipelined arquitecture of MLP Neural Networks in FPGA” disponível para consulta na biblioteca do IEEE explore.

9. Apêndice

9.1 Aproximações da sigmóide

```
function value = firstorder_improved(x)
if x > 0
    x1 = -x;
else
    x1 = x;
end
g = '0';
h = '.5+ x/4';
delta = 0.2638;

for i = 1:4,
    g_value = subs(g,x1);
    h_value = subs(h,x1);
    if g_value > h_value
        g_linha = g;
    else
        g_linha = h;
    end
    h = strcat('.5*',(' , g, '+,h,'+',num2str(delta),')');
    g = g_linha;
    delta = delta / 4;
end
g_value = subs(g,x1);
h_value = subs(h,x1);

if g_value < h_value
    value = h_value;
else
    value = g_value;
end
if x > 0
    value = 1-value;
end

function y = PLAN_appox(x)
if abs(x) >= 5
    y = 1;
elseif abs(x) >= 2.375
```

```

y = .03125 * abs(x) + .84375;
elseif abs(x) >= 1
    y = .125 * abs(x) + .625;
else
    y = 1/4 * abs(x) + .5;
end

if x < 0
    y = 1-y;
end

```

```

function y = piecewise_nd_order(x)
if x >= -4 & x < 0
    y = (( 1 - abs(x/4) )^2)/2;
elseif x >= 0 & x <= 4
    y = 1 - (( 1 - abs(x/4) )^2)/2;
end

```

10. Referências

Altera Accelerating High-Performance Computing With FPGAs [Online]. - Agosto de 2007. - 19 de janeiro de 2011. - URL: <http://www.altera.com/literature/wp/wp-01029.pdf>.

Amin H., Curtis K.M. e Hayes-Gill B.R. Piecewise linear approximation applied to nonlinear function of a neural network [Conferência] // IEEE Proc. Circuits - Devices Syst.. - 1997 . - pp. 313–317.

Basterretxea K., Tarela J. M. e Del Campo I. Approximation of sigmoid function and the derivative for hardware implementation of artificial neurons [Conferência] // IEEE Proc.- Circuits Devices Syst.. - 2004.

Berthelot E., Nouvel F. e Houzet D. Partial and dynamic reconfiguration of FPGAs: a top down design methodology for an automatic implementation [Conferência] // Proceedings 20th IEEE International Parallel & Distributed Processing Symposium. - 2006.

Braga A. P., Carvalho A. P. L. F. e Ludermir T. B. Redes Neurais Artificiais [Livro]. - [s.l.] : LTC, 2007.

Braga A.L.S. [et al.] VANNGen: a Flexible CAD Tool for Hardware Implementation of Artificial [Conferência] // International Conference on Reconfigurable Computing and FPGAs. - 2005.

Canas A. e al et FPGA Implementation of a Fully and Partially Connected MLP [Seção do Livro] // FPGA Implementations of Neural Networks / A. do livro Omondi A. R. e Rajapakse J. C.. - [s.l.] : Springer-Verlag, 2006.

Cray CrayXD1datasheet [Online]. - 2010. - 19 de janeiro de 2011. - http://www.hpc.unm.edu/~tlthomas/buildout/Cray_XD1_Datasheet.pdf.

FANN Fast Artificial Neural Network Library [Online]. - 2011. - 06 de 03 de 2011. - <http://sourceforge.net/projects/fann/>.

Girau B. FPNA: Applications and implementations [Seção do Livro] // FPGA Implementations of Neural Networks / A. do livro Omondi A. R. e Rajapakse J. C.. - [s.l.] : Springer-Verlag, 2006.

Girones R. G. e Agundis A. R. FPGA Implementation of Non-Linear Predictors [Seção do Livro] // FPGA Implementations of Neural Networks / A. do livro Omondi A. R. e Rajapakse J. C.. - [s.l.] : Springer-Verlag, 2006.

Gokhale M. B. e Graham P. S. Reconfigurable computing: Accelerating computation with field-programmable gate arrays [Livro]. - New York : Springer-Verlag, 2005.

GRAPE: A programmable multi-purpose computer for many-body simulations [Online] // The GRAPE Project. - 19 de janeiro de 2011. - <http://grape-dr.adm.s.u-tokyo.ac.jp/project-en.htm>.

Gugala K. e Rybarczyk A. Automatic synthesizable VHLD code generation from neural

networks models using Matlab [Conferência] // International Conference Mixed Design of Integrated Circuits & Systems. - 2009.

Huerta P. [et al.] Operating System for Symmetric Multiprocessors on FPGA [Conferência] // International Conference on Reconfigurable Computing and FPGAs. - 2008.

Hung A., Bishop W. e Kennings A. Symmetric Multiprocessing on Programmable Chips Made Easy [Conferência] // Proceedings of the conference on Design, Automation and Test in Europe. - 2005.

IEEE IEEE1076.3 work group web site [Online]. - 07 de 03 de 2011. - <http://www.vhdl.org/vhdlsynth/>.

Lee Y. and Ko S. B. FPGA implementation of a face detector using neural networks [Conference] // IEEE CCECE/CCGEI. - 2006.

Makino J. e Taiji M. Special-purpose Computers for Scientific Simulations: The GRAPE Systems [Livro]. - [s.l.] : John Wiley & Sons, 1998.

Omondi A. R., Rajapakse J. C. and Bajger M. FPGA Neurocomputers [Book Section] // FPGA Implementations of Neural Networks / book auth. Omondi A. R. Rajapakse J. C.. - [s.l.] : Springer-Verlag, 2006.

Oniga S. [et al.] FPGA Implementation of Feed-Forward Neural Networks for Smart Devices Development [Conferência] // International Symposium on Signals, Circuits and Systems. - 2009.

Petrie C. [et al.] High performance embedded computing using field programmable gate arrays [Conferência] // Proceedings of the 8th Annual Workshop on High-performance Embedded Computing. - 2004.

Potter J. L. The Massively Parallel Processor [Livro]. - [s.l.] : MIT Press, 1985.

Rabuñal J. R. e Dorado J. Artificial neural networks in real life applications [Livro]. - [s.l.] : Idea group publishing, 2006.

Rocha R. C. F. DESENVOLVIMENTO DE UMA PLATAFORMA RECONFIGURÁVEL PARA MODELAGEM 2D, EM SÍSMICA, UTILIZANDO FPGAS. // Dissertação de Mestrado. - 2010.

SGI SGI RASC RC100 blade datasheet [Online]. - 2005. - 19 de janeiro de 2010. - http://www.silicongraphics.ru/pdf/rasc_data.pdf.

Soares A.M. [et al.] Field Programmable Gate Array (FPGA) Based Neural Network Implementation of Motion Control and Fault Diagnosis of Induction Motor Drive [Conferência] // IEEE International Conference on Industrial Technology . - 2006.

Souza V. L., Medeiros V. W. and de Lima M. E. Architecture for Dense Matrix Multiplication on a High- Performance Reconfigurable System [Conference] // Proceedings of the 21st Annual Symposium on integrated Circuits and System Design. - Natal : [s.n.], 2009.

Sutherland S., Davidmann S. e Flake P. SystemVerilog For Design [Livro]. - [s.l.] : Springer, 2006. - Vol. Segunda edição.

top500 [Online]. - janeiro 19, 2011. - <http://www.top500.org/overtime/list/35/archtype>.

UCI repository [Online]. - 10 de 02 de 2011. - <http://archive.ics.uci.edu/ml/index.html>.

Underwood K. D. e Hemmert K. S. Closing the gap: trends in sustainable floating-point BLAS performance [Conferência] // Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. - 2004.

Zhang M., Vassiliadis S. e Delgado-Frias J.G. Sigmoid generators for neural computing using piecewise approximations [Conferência] // IEEE Trans. Comput.. - 1996. - pp. 1045-1049.

Assinaturas

Edna Natividade da Silva Barros

Orientadora

Antonyus Pyetro do Amaral Ferreira

Aluno/Autor

Recife, 15 de março de 2011