Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Projeto e Treinamento de Redes Neurais Recorrentes utilizando Computação Estocástica para Síntese em FPGA

Luigi Nunes Gil

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Marcus Vinicius Lamar

Brasília
2019

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Engenharia da Computação

Coordenador: Prof. Dr. José Edil Guimarães de Medeiros

Banca examinadora composta por:

Prof. Dr. Marcus Vinicius Lamar (Orientador) — CIC/UnB

Prof. Dr. Marcelo Grandi Mandelli — CIC/UnB

Prof. Dr. Marcelo Antônio Marotta — CIC/UnB

CIP — Catalogação Internacional na Publicação

Gil, Luigi Nunes.

Projeto e Treinamento de Redes Neurais Recorrentes utilizando Computação Estocástica para Síntese em FPGA / Luigi Nunes Gil. Brasília : UnB, 2019.
79 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2019.

1. computação estocástica, 2. redes neurais recorrentes,
3. retropropagação, 4. FPGA

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Projeto e Treinamento de Redes Neurais Recorrentes utilizando Computação Estocástica para Síntese em FPGA

Luigi Nunes Gil

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Marcus Vinicius Lamar (Orientador)
CIC/UnB

Prof. Dr. Marcelo Grandi Mandelli
CIC/UnB

Prof. Dr. Marcelo Antônio Marotta
CIC/UnB

Prof. Dr. José Edil Guimarães de Medeiros
Coordenador do Curso de Engenharia da Computação

Brasília, 12 de dezembro de 2019

Resumo

Este trabalho apresenta o treinamento e implementação de redes neurais recorrentes utilizando computação estocástica. Um estudo teórico acerca de redes neurais foi feito, a fim de que fosse possível compreender os algoritmos de treinamento e utilização das redes adotadas no projeto. Foram implementadas redes neurais tradicionais e redes neurais recorrentes, além de redes neurais recorrentes utilizando computação estocástica em software, com a finalidade de validar a utilização de tal paradigma na construção destas redes para então sintetizá-las em hardware, através de chips FPGA, de modo que sejam avaliadas questões como precisão da saída, tamanho do circuito digital final e comparação com implementações utilizando ponto fixo e ponto flutuante. A utilização de computação estocástica permite a construção de circuitos digitais mais simples para realizar operações necessárias em redes neurais, possibilitando que grandes redes possam ser sintetizadas em FPGA e problemas mais complexos possam ser resolvidos através de hardware.

Palavras-chave: computação estocástica, redes neurais recorrentes, retropropagação, FPGA

Abstract

This paper presents the implementation of recurrent neural networks using stochastic computation for training. A theoretical study about neural networks was made, so that it was possible to understand the training algorithms and use of networks adopted in the project. Traditional neural networks and recurrent neural networks were implemented, as well as recurrent neural networks using stochastic computation in software, in order to validate the use of such a paradigm in the construction of these networks and then synthesize them in hardware, using FPGA chips, so that issues such as output accuracy, final digital circuit size, and comparison with fixed point and floating point implementations are evaluated. The use of stochastic computing allows the construction of simpler digital circuits to perform necessary operations in neural networks, allowing large networks to be synthesized in FPGA and more complex problems to be solved through hardware.

Keywords: stochastic computing, recurrent neural networks, backpropagation, FPGA

Sumário

1 Introdução **1**
1.1 Problema 1
1.2 Objetivos 1
1.3 Organização 2

2 Fundamentação Teórica **3**
2.1 Computação Estocástica 3
2.1.1 Representação Estocástica de um Número 3
2.1.2 Operações Aritméticas com Números Estocásticos 4
2.1.3 Conversão de números binários para números estocásticos 7
2.1.4 Geradores de Números Pseudo-Aleatórios e Precisão 9
2.1.5 Aplicações 10
2.2 Redes Neurais 12
2.2.1 Benefícios no uso de redes neurais artificiais 12
2.2.2 Neurônio 14
2.2.3 Função de Ativação 15
2.2.4 Estruturas de Redes Neurais 18
2.2.5 Treinamento por Retropropagação de Erros 22
2.3 Dispositivos Eletrônicos Reconfiguráveis 25

3 Sistema Proposto **28**
3.1 Implementação em *Software* 28
3.2 Implementação em FPGA 33
3.2.1 Módulo Rede Neural 34
3.2.2 Neurônios estocásticos 36
3.2.3 Camadas Neurais 39
3.2.4 Sinapses 41
3.2.5 Controle da Rede Neural 44
3.3 Treinamento da Rede Neural Recorrente Estocástica 45

4 Resultados Obtidos **46**
4.1 Estudos de Caso 47
4.1.1 Problema do XOR Temporal 47
4.1.2 Reconhecimento de Fonema 50
4.2 Comparação com a Literatura 61

5 Conclusão **65**

Referências **67**

Lista de Figuras

2.1 Multiplicação de *bit-streams* com probabilidades 4/8 e 6/8. 5
2.2 Multiplexador para soma de S_1 e S_2, com entrada seletora S_3. 6
2.3 Circuitos conversores: (a) binário-estocástico; (b) estocástico-binário 7
2.4 Circuito estocástico para implementar a Equação 2.6 8
2.5 LFSR de 5 bits. 10
2.6 Controlador para motor indutivo baseado em computação estocástica [1] . 11
2.7 Modelo de um neurônio artificial. 14
2.8 Função Threshold. 15
2.9 Função Logística. 16
2.10 Tangente Hiperbólica. 17
2.11 Derivada da Tangente Hiperbólica. 18
2.12 Neurônio com entradas e saídas. 18
2.13 Exemplo de rede neural artificial de uma camada. 19
2.14 Exemplo de rede neural artificial multi-camada. 20
2.15 Unidade de atraso. 21
2.16 Exemplo de rede neural recorrente de Elman. 21
2.17 Exemplo de rede neural de recorrente de Jordan. 22
2.18 Rede neural de um neurônio. 23
2.19 Diagrama de blocos do Elemento Lógico do FPGA CycloneIV. 26

3.1 Módulos principais do sistema. 33
3.2 Esquema de *feedforward* de uma rede neural recorrente com uma camada oculta. 34
3.3 Funcionamento interno do Banco de Pesos. 35
3.4 Estrutura interna do módulo "Pesos" do Banco, responsável pelo armazenamento e processamento de um peso sináptico individual. 36
3.5 Circuito utilizado para conversão estocástica dos pesos sinápticos. 37
3.6 Visão geral de um neurônio estocástico. 37
3.7 Diagrama da tangente hiperbólica estocástica. 38
3.8 Tangente hiperbólica estocástica implementada. 39

3.9 Diagrama que descreve uma camada oculta da rede. 40
3.10 Implementação da camada de saída. 41
3.11 Blocos de circuitos responsáveis pela lógica sináptica. 42
3.12 Circuito sináptico. 43
3.13 Circuito sináptico reverso, usado na retropropagação de erros. 44
3.14 Códigos de operação da rede neural no *FPGA* 45

4.1 Representações dos valores de saída -1, 0 e 1 no FPGA 49
4.2 Sinais do problema *phoneme* 51
4.3 Estrutura da rede de Elman no Matlab com 5 neurônios na camada escondida. 52
4.4 Média de porcentagem de acerto. 52
4.5 Respostas das redes 53
4.6 Média de porcentagem de acerto para 10 mil épocas. 54
4.7 Resposta da rede com 5 neurônios treinada com 10 mil épocas. 55
4.8 Média de porcentagem de acerto para 20 mil épocas. 56
4.9 Resposta da rede com 15 neurônios treinada com 20 mil épocas. 57
4.10 Média de porcentagem de acerto para 40 mil épocas. 57
4.11 Respostas das redes treinadas com 40 mil épocas. 59
4.12 Média de porcentagem de acerto para 50 mil épocas. 60
4.13 Resposta rede com 10 neurônios treinada com 50 mil épocas. 61

Lista de Tabelas

4.1 Tabela Verdade da função XOR. 48
4.2 Resposta da rede com 2, 3 e 4 neurônios na camada escondida. 48
4.3 Requisitos Físicos para Sintetizar Rede com 2 neurônios. 50
4.4 Melhores Redes treinadas pelo Matlab. 53
4.5 Melhores Redes para 10 mil épocas. 55
4.6 Melhores Redes para 20 mil épocas. 56
4.7 Melhores Redes para 40 mil épocas. 58
4.8 Melhores Redes para 50 mil épocas. 60
4.9 Requisitos Físicos para Sintetizar Rede com 15 neurônios. 61
4.10 Requisitos Físicos de síntese da rede neural recorrente em ponto flutuante para problema XOR temporal [2] 62
4.11 Requisitos físicos para sintetizar uma rede neural recorrente estocástica com 4 neurônios. 62
4.12 Requisitos físicos da rede neural recorrente em ponto fixo com 12 neurônios na camada escondida e 2 neurônios na camada de saída [3] 63
4.13 Requisitos físicos da rede neural estocástica com 12 neurônios na camada escondida e 2 neurônios na camada de saída. 63

Lista de Abreviaturas e Siglas

ASIC Circuito Integrado de Aplicação Específica (do inglês *Application Specific Integrated Circuit*). 1

FPGA Arranjo de Portas Programável em Campo (do inglês *Field-Programmable Gate Array*). 1–3, 11, 25–28, 39, 45, 49, 50, 61, 63–66

GDM Gradiente Descendente com Momentum. 22

LFSR Registrador de Deslocamento com realimentação Linear (do inglês *Linear-Feedback Shift Register*). 10, 35

LUT *Look-Up Table*. 26

MIF Arquivo de Inicialização de Memória (do inglês *Memory Initialization File*). 46, 49

MLP Perceptron Multicamadas (do inglês *Multilayer Perceptron*). 20, 39

RAM Memória de Acesso Aleatório (do inglês *Random Access Memory*). 44, 49

SNG Gerador de Números Estocásticos (do inglês *Stochastic Number Generator*). 8, 10

TDNN Redes Neurais com Atraso Temporal (do inglês *Time-Delay Neural Networks*). 50

Capítulo 1

Introdução

O interesse na implementação de redes neurais artificiais em *hardware* não é atual [4]. Tal interesse baseia-se na expectativa de aceleração do processamento computacional de dispositivos reconfiguráveis, como o Arranjo de Portas Programável em Campo (do inglês *Field-Programmable Gate Array*) (FPGA), em relação a implementações em *software*, como é comumente feito. A vantagem na utilização de dispositivos reconfiguráveis em relação a *chips* de Circuito Integrado de Aplicação Específica (do inglês *Application Specific Integrated Circuit*) (ASIC) fabricados é percebida na facilidade em reestruturar a organização topológica da arquitetura utilizada, possibilitando a modificação do sistema com base nas necessidades da aplicação.

1.1 Problema

A implementação de operações aritméticas utilizando ponto fixo e ponto flutuante em *hardware* reconfigurável consome bastante recurso disponível pelo dispositivo, o que dificulta a síntese de redes neurais recorrentes em FPGA. Através da utilização de conceitos e técnicas de computação estocástica, é possível implementar circuitos digitais mais baratos em termos de recursos necessários e, dessa forma, utilizar redes maiores em *hardware* para a solução de problemas mais complexos que não seriam possíveis em ponto flutuante e ponto fixo.

1.2 Objetivos

O objetivo principal deste trabalho é o estudo e utilização de conceitos e técnicas de computação estocástica para a implementação de redes neurais para aplicações em tempo real em FPGA. Para que seja possível verificar a eficácia da solução proposta para o problema apresentado, as seguintes metas foram estabelecidas:

1. Implementar um modelo de rede neural recorrente de modo que seja simulado o comportamento estocástico que será observado no circuito digital.
2. Treinar o modelo desenvolvido com problema XOR temporal, problema clássico da literatura para validar o funcionamento da implementação.
3. Treinar o modelo com problema de identificação de fonema, com objetivo de comprovar a convergência da implementação para problemas mais complexos que o XOR temporal.
4. Carregar um modelo de rede neural recorrente estocástica em FPGA com pesos das conexões neurais obtidas das melhores redes treinadas para o problema XOR temporal e verificar seu funcionamento no sistema final.
5. Comparar a quantidade de recursos necessários para síntese em FPGA de rede neural recorrente utilizando computação estocástica com outras implementações da mesma arquitetura de rede.

1.3 Organização

Este trabalho é composto por cinco capítulos, sendo eles a Introdução, no qual são identificados o problema a resolvido e a solução proposta para tal. Os conceitos e base teórica relevantes ao problema e técnicas envolvidas são apresentadas no Capítulo 2, Revisão Teórica, de modo que o embasamento teórico necessário para o desenvolvimento da solução seja apresentado. O Capítulo 3 elabora os processos utilizados, de maneira detalhada, para a implementação de uma rede neural recorrente simulando o comportamento estocástico do sistema em *software*, os algoritmos envolvidos e a motivação para tal. No Capítulo 4, avalia-se o sistema desenvolvido com problema clássicos da literatura, além de problema mais complexo para comprovar o funcionamento da solução proposta. Também é abordado comparativos de implementação de redes neurais recorrentes em *hardware* para dispositivos reconfiguráveis, onde compara-se a implementação estocástica e implementações em ponto fixo e ponto flutuante. Por fim, o Capítulo 5 apresenta as conclusões em torno do trabalho, além de trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo serão abordados os tópicos que envolvem o trabalho feito nesta monografia. Primeiramente, é apresentado ao leitor o conceito de computação estocástica, motivações para uso e operações feitas no domínio estocástico. Em sequência, é feita uma revisão a respeito de redes neurais artificiais, mostrando ao leitor os principais componentes de uma rede neural artificial e como introduzir recorrência na arquitetura, possibilitando que uma rede tenha capacidade de resolver problemas no tempo. Por fim, uma introdução aos dispositivos eletrônicos reconfiguráveis, destacando-se dispositivos FPGA.

2.1 Computação Estocástica

A computação estocástica surgiu como alternativa de baixo custo em relação a computação lógica binária convencional. Essa forma de computação possibilita a representação e processamento de dados na forma de probabilidades, empregando circuitos com baixa complexidade nas unidades aritméticas [5].

Algumas características da computação estocástica são tolerância a erros, baixo consumo de energia, alta confiabilidade e tamanho reduzido dos circuitos eletrônicos devida a baixa complexidade das unidades aritméticas [5]. Essas funcionalidades possibilitam colocar a computação estocástica como boa alternativa a metodologias convencionais em aplicações variadas.

2.1.1 Representação Estocástica de um Número

Neste tipo de computação, os números são representados, assim como em outras metodologias, como uma sequência de *bits* denominada *bit-streams*. Essas sequências podem ser processadas por circuitos digitais simples e podem ser interpretadas como probabilidades, tanto em circunstâncias normais quanto em falhas [5].

Tomando-se como exemplo um *bit-stream* S contendo 25% dos *bits* 1 e 75% dos *bits* 0. Tal *bit-stream* pode ser interpretado como a probabilidade $p = 0.25$, uma vez que a probabilidade de se observar um *bit* 1 em uma posição arbitrária de S é p.

É importante notar que o tamanho do *bit-stream* ou sua organização não influenciam no valor da probabilidade. Por exemplo, as seguintes sequências de *bit-stream*

- 1 0 0 0
- 0 1 0 0
- 0 1 0 0 0 1 0 0

são representações possíveis para a probabilidade $p = 0.25$. Nota-se que a probabilidade, então, é dependente da razão da quantidade de *bits* 1 em relação ao tamanho do *bit-stream*. Tais *bit-streams* associados a probabilidades são chamados de *números estocásticos*.

2.1.2 Operações Aritméticas com Números Estocásticos

Uma das características mais atrativas na utilização da computação estocástica é que tal técnica possibilita a utilização de implementações de baixo custo para operações aritméticas. Elementos lógicos simples são utilizados para computação de *bit-streams*, evitando a utilização de grandes circuitos somadores e multiplicadores síncronos, que acabam consumindo muito recurso de *hardware* [5].

Representação Bipolar

O número estocástico é dado como uma probabilidade de ocorrência de *bit* 1 em cada posição de um *bit-stream* S, dessa forma, espera-se que o intervalo dessa probabilidade seja $[0, 1]$. Entretanto, é possível particionar tal intervalo de probabilidade, tornando possível a representação de valores positivos e negativos.

Considerando uma quantidade normalizada $\hat{x}$ ao intervalo $-1 \leq \hat{x} \leq 1$ e p a probabilidade geradora de X:

$$p = p(X = 1) = \frac{1}{2}\hat{x} + \frac{1}{2} \quad (2.1)$$

O valor de $\hat{x} = 1$ é caracterizado por uma sequência de *bits* 1. Uma sequência de *bits* 0 caracteriza o valor de $\hat{x} = -1$. Considerando o caso onde a quantidade de *bits* 1 e 0 é igual, ou seja, onde a probabilidade deveria ser $p = 0.5$, a sequência representaria uma quantidade cuja magnitude é igual a zero.

$$\hat{x} = 2p - 1 \quad (2.2)$$

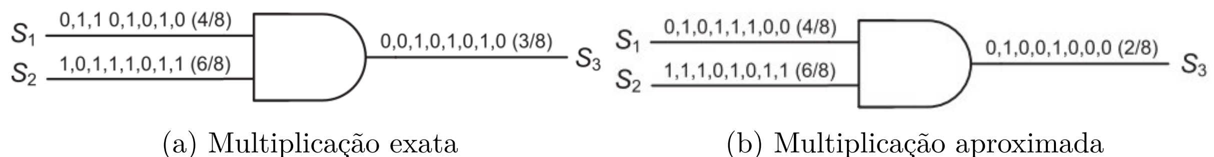
O posicionamento de valores próximos ao valor zero no centro do intervalo da probabilidade geradora, ou seja, onde a acurácia é mínima, apresenta menores erros na estimativa, uma vez que a magnitude é naturalmente pequena. Tal característica demonstra vantagem na utilização da representação bipolar para números estocásticos.

Adotou-se tal representação para os números neste trabalho pois a natureza desta representação é simples e compatível com os conceitos envolvidos a serem apresentados no decorrer do texto.

Multiplicação

A multiplicação de números estocásticos, por sua natureza probabilística, pode ser computada através da operação AND, utilizando apenas uma simples porta lógica no circuito [5].

Considerando-se dois *bit-streams* binários como entrada de uma porta lógica AND, cujas probabilidades de se observar um *bit* 1 em cada posição dos *bit-streams* seja, respectivamente, p_1 e p_2, então a probabilidade de se observar um *bit* 1 em cada posição do *bit-stream* de saída será dada por $p_1 \times p_2$, assumindo que ambos os *bit-streams* de entrada são independentes, isto é, não correlacionados.



(a) Multiplicação exata

(b) Multiplicação aproximada

Figura 2.1: Multiplicação de *bit-streams* com probabilidades 4/8 e 6/8.

Na Figura 2.1 é possível observar duas situações diferentes onde ocorrem a multiplicação de dois *bit-streams* S_1 e S_2, cujas probabilidades são, respectivamente, 4/8 e 6/8. A situação ilustrada por *(a)* mostra uma multiplicação exata dos *bit-streams*, onde a saída S_3 é 3/8. No caso de *(b)*, temos um resultado aproximado, onde S_3 é 2/8, um valor aproximado do produto exato 3/8.

Tomando-se como exemplo a situação da Figura 2.1, é possível perceber que uma inversão de *bit* na saída resulta em uma diferença pequena em relação ao resultado exato da operação estocástica. Caso o resultado da operação $4/8 \times 6/8$ seja 2/8 ou 4/8, ainda são valores próximos de 3/8. Se considerarmos o mesmo valor de 3/8 em representação binária convencional com valor 0.011, a inversão de um *bit* de ordem alta causaria um erro muito grande no resultado final. Por exemplo, caso o segundo *bit* da esquerda para a direita seja invertido, gerando o valor 0.111, a mudança no resultado seria de 3/8 para 7/8. A representação de números estocásticos não pondera a posição dos *bits*.

Adição e Subtração

Como o número estocástico é interpretado como uma probabilidade, naturalmente seu intervalo corresponde ao intervalo fechado $[0,1]$. Dessa forma, uma simples soma de dois *bit-streams* S_1 e S_2 torna-se inconveniente no cenário estocástico, uma vez que o resultado desta soma encontra-se no intervalo $[0,2]$.

Deste modo, através da utilização de um multiplexador de duas entradas e uma entrada seletora, conforme Figura 2.2 pode computar a soma dos números estocásticos $p(S_1)$ e $p(S_2)$. A entrada seletora do multiplexador, a qual será chamada de S_3, corresponde a probabilidade $p(S_3) = 1/2$. Tal número pode ser fornecido por um gerador de números pseudo-aleatórios. Através dessa configuração, a saída S_4 pode ser representada pela Equação 2.3

$$p(S_4) = p(S_3)p(S_1) + (1 - p(S_3))p(S_2) = (p(S_1) + p(S_2))/2 \tag{2.3}$$

dessa forma, a entrada seletora S_3 escala a soma dos *bit-streams* em $1/2$, ou seja

$$S_4 = \frac{S_1 + S_2}{2}. \tag{2.4}$$

A Figura 2.2 ilustra a configuração do multiplexador em questão para o somatório de dois números estocásticos S_1 e S_2, ambos com 8 *bits*, cujos valores de probabilidade são, respectivamente, 7/8 e 3/8. Uma sequência de 8 *bits* é utilizada como entrada S_3 com valor 4/8 (1/2), a fim de satisfazer a equação 2.1. É possível perceber que, para a situação ilustrada, a soma ocorre da seguinte forma

$$p(S_4) = (7/8 + 3/8)/2 = 5/8 \tag{2.5}$$

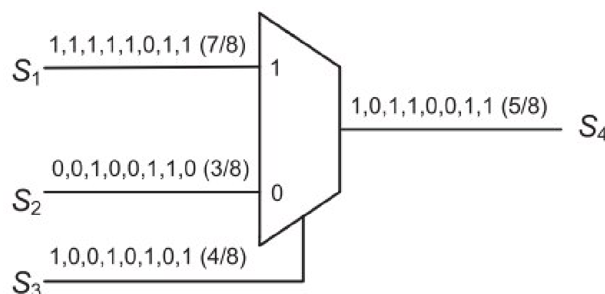


Figura 2.2: Multiplexador para soma de S_1 e S_2, com entrada seletora S_3.

Para realização de operações de subtração, basta negar uma das entradas S_1 ou S_2 e computar a soma normalmente.

2.1.3 Conversão de números binários para números estocásticos

Tratando-se de sistemas estocásticos, circuitos conversores de números binários para números estocásticos, e a conversão contrária também, são elementos de muita importância na modelagem de computadores estocásticos [5].

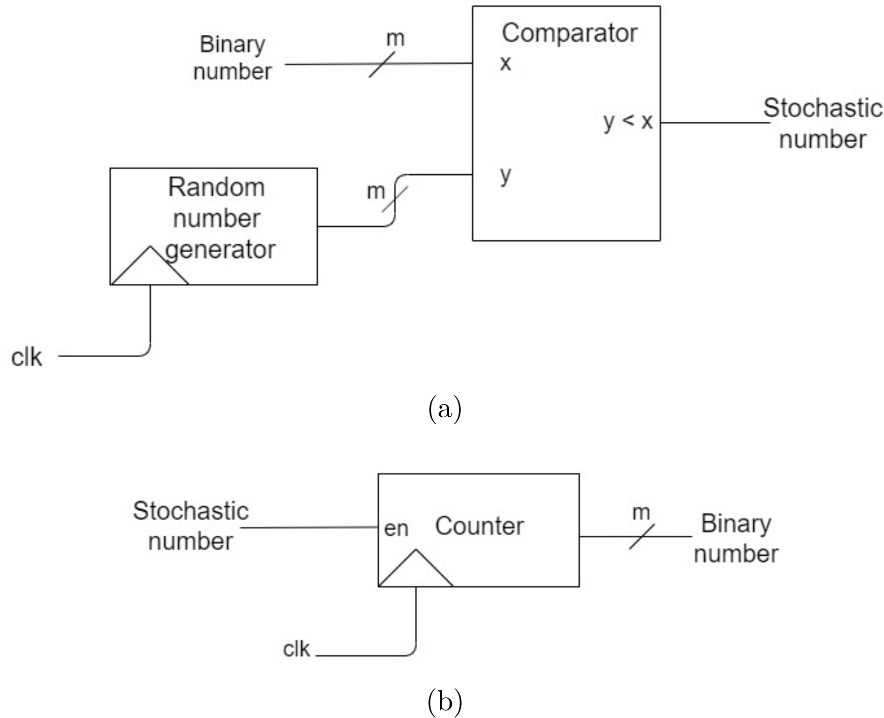


Figura 2.3: Circuitos conversores: (a) binário-estocástico; (b) estocástico-binário

A conversão de números binários para números estocásticos ocorre de acordo com o esquema apresentado em Figura 2.3a. O processo de conversão ocorre de modo que seja gerado um número de m *bits* binário aleatório a cada ciclo de *clock* por meio de um gerador de números aleatórios, ou pseudo-aleatórios, e então compara com o número binário de m *bits* da entrada. A saída do comparador assume o valor 1 caso o número aleatório seja menor que o número binário e, caso contrário, assume o valor 0. Caso seja assumido que os números aleatórios estejam uniformemente distribuídos no intervalo fechado $[0, 1]$, então a probabilidade de obter um valor 1 na saída do comparador a cada ciclo de *clock* é igual a entrada binária do conversor interpretado como um número fracionário.

No caso da conversão de números estocásticos para números binários o processo ocorre de forma mais simples. O valor de probabilidade p depende da quantidade de *bits* 1 que existem no *bit-stream* de interesse. Basta, então, contar a quantidade de *bits* 1 para se ter o valor de p. A Figura 2.3b ilustra um circuito com um contador para realizar a conversão.

Para ilustrar o funcionamento de um circuito estocástico, desde uma entrada binária convencional, sua conversão para sequência estocástica através de um gerador de número

estocástico, o processamento desses sinais e a conversão para binário, será utilizado como exemplo a Figura 2.4 [6], cujo circuito apresentado resolve a seguinte equação

$$z = x_1 x_2 x_4 + x_3(1 - x_4). \tag{2.6}$$

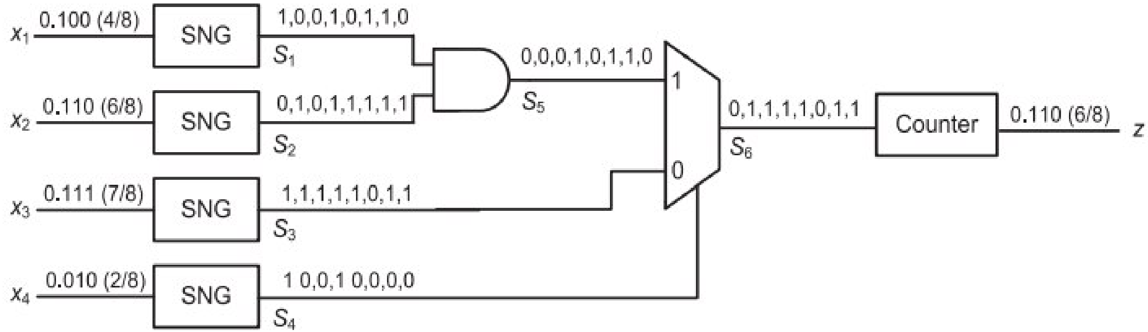


Figura 2.4: Circuito estocástico para implementar a Equação 2.6

As entradas x_1, x_2, x_3, x_4 são dadas como números binários convencionais em ponto fixo e, para operá-los estocasticamente, deve-se convertê-los para *bit-streams* através de um Gerador de Números Estocásticos (do inglês *Stochastic Number Generator*) (SNG).

Levando-se em consideração que os *bit-streams* S_1, S_2, S_3 e S_4 têm probabilidades, respectivamente, iguais a 4/8, 6/8, 7/8 e 2/8. O valor do *bit-stream* S_5 é o resultado da multiplicação de S_1 e S_2, ou seja, 3/8. Desse modo, a probabilidade de um *bit* 1 em S_6 é a probabilidade de um *bit* 1 em S_4 e S_5, somado com a probabilidade de um *bit* 0 em S_4 e um *bit* 1 em S_3. Tal lógica pode ser escrita da seguinte forma

$$p(S_6) = p(S_4 \wedge S_5) + p(S_4 \wedge S_3) - p(S_4)p(S_1)p(S_2) + (1 - p(S_4))\, p(S_3). \tag{2.7}$$

O valor de S_6 é a representação estocástica da Equação 2.6, onde o contador no final do circuito é responsável pela conversão do número estocástico para um número binário convencional.

É importante ressaltar que o resultado final z, na Figura 2.4 é 6/8 somente se houverem seis *bits* 1 no *bit-stream* S_6 em oito ciclos de *clock*, caso contrário, o contador conversor retornará um valor diferente de 6/8. A probabilidade do contador gerar exatamente seis *bits* 1 é dada por

$$P\{z = 6/8\} = \binom{8}{6}(6/8)^6(2/8)^2 \cong 0.31 \tag{2.8}$$

o que implica dizer que há uma chance de 69% do contador não gerar seis *bits* 1 na saída, o que causaria falta de precisão no resultado final.

A preocupação com precisão na computação estocástica sempre esteve presente [5]. Flutuações inerentes de números aleatórios, e a correlação entre os números estocásticos a serem processados são fatores que levam a imprecisões. Como será visto adiante, fontes geradoras de números puramente aleatórios não costumam ser utilizados em projetos de computadores estocásticos, uma vez que geradores determinísticos ou pseudo-aleatórios entregam resultados melhores como fontes de geradores de números estocásticos, de forma prática e teórica. Tais componentes podem ser utilizados para construção de circuitos estocásticos com alta precisão [5].

2.1.4 Geradores de Números Pseudo-Aleatórios e Precisão

Conforme já foi abordado neste texto, um número estocástico é formado por uma sequência de de *bits* de tamanho n com n_1 1s e $1 - n_1$ 0s, cuja representação é o número $\frac{n_1}{n}$. Dessa forma, a representação de um valor na forma estocástica não é única, havendo mais formas de representar um número de mesma quantidade de *bits*. Como o sistema numérico estocástico é redundante, o qual existem $\binom{n}{n_1}$ possíveis representações para cada valor de $\frac{n_1}{n}$.

Tomando-se como exemplo um número cujo valor de $n = 4$, existem, então, seis formas de representar a fração $\frac{2}{4}$:

- 0 0 1 1
- 0 1 0 1
- 0 1 1 0
- 1 0 0 1
- 1 0 1 0
- 1 1 0 0

É importante perceber que as sequências de n *bits* podem representar apenas números dentro do conjunto $\frac{0}{n}, \frac{1}{n}, \frac{2}{n}, ..., \frac{n-1}{n}, \frac{n}{n}$, ou seja, somente uma pequena parcela de números reais do intervalo $[0, 1]$ pode ser expressado de forma exata por computação estocástica. De forma resumida, uma sequência de *bits* S de tamanho n pode ser considerado como um número estocástico $\hat{p}$ caso possa ser interpretado como um número racional $p = \frac{n_1}{n}$, tomando-se em conta a quantidade de *bits* 1 no *bit-stream* S como n_1.

Flutuações aleatórias na representação de números estocásticos e correlações entre números que estão sendo comparados entre si entre os números são situações que influenciam na precisão da computação estocástica.

Tomando-se como duas sequências binárias S_1 e S_2, onde $S_1 = (S_1(1), S_1(2), ..., S_1(n))$ e $S_2 = (S_2(1), S_2(2), ..., S_2(n))$, é possível afirmar que tais sequências são independentes, isto é, não correlacionadas se, e somente se,

$$\sum_{i=1}^{n} S_1(i)S_2(i) = \frac{\sum_{i=1}^{n} S_1(i) \times \sum_{i=1}^{n} S_2(i)}{n} \tag{2.9}$$

caso contrário, as sequências são consideradas correlacionadas [7].

A fim de exemplificar o efeito da correlação, consideremos dois números estocásticos binários de 8 *bits*: $S_1 = (1,1,1,1,0,0,0,0)$; $S_2 = (0,1,0,1,0,1,0,1)$, onde ambos os números apresentam $p = 0.5$. Pela definição da Equação 2.9, ambas as sequências são não correlacionadas. Caso seja realizada uma operação *and* entre as duas sequências, é obtido $S_1 \times S_2 = (0,1,0,1,0,0,0,0) = 0.25$. Considerando-se agora dois números correlacionados, onde $S_1 = S_2 = (0,0,0,0,1,1,1,1) = 0.5$, a operação *and* entre tais sequências resulta em um valor incorreto $S_1 \times S_2 = (0,0,0,0,0,0,0,0) = 0$.

De modo que tais imprecisões sejam reduzidas, é preciso construir SNGs que sejam capazes de produzir sequências de *bits* não correlacionados e aleatórios. O Registrador de Deslocamento com realimentação Linear (do inglês *Linear-Feedback Shift Register*) (LFSR) é um gerador de números pseudoaleatório que possui tais características e, muitas vezes, é proposto para projetos de computação estocástica [8]. A Figura 2.5 apresenta um diagrama de um LFSR de 5 *bits*.

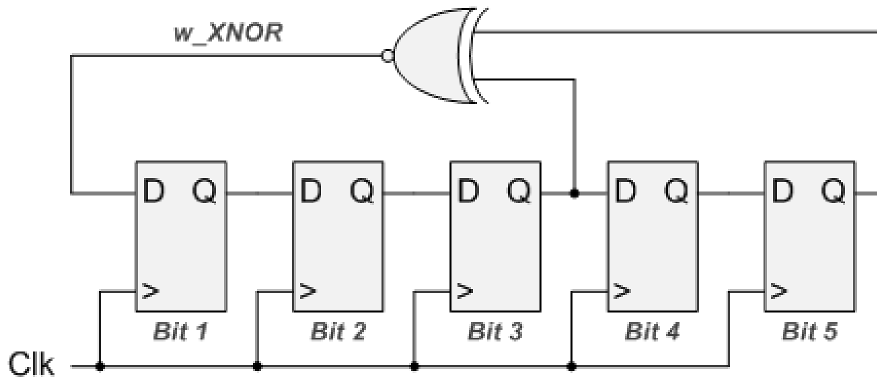


Figura 2.5: LFSR de 5 bits.

Os LFSR possuem uma cadeia de *m flip-flops* e podem transitar por $n = 2^m$ - *1* estados distintos, uma vez que o estado em que todos os *bits* de saída são 0 é desconsiderado.

2.1.5 Aplicações

A computação estocástica tem sido objeto de estudo para diversos tipos de aplicações. Além dos operações já citadas, como soma e multiplicação de sequências binárias, aplica-

ções com operações de divisão e raiz quadrada [9], operações matriciais [10], aritmética polinomial [11][12] e análise de confiabilidade [13][14].

Como probabilidades são quantidades analógicas intrinsecamente, a computação estocástica foi adotada para tarefas de computação analógicas ou híbridas (analógico-digitais) através do processamento digital de sinais analógicos [15][16].

Aplicações envolvendo redes neurais [17][18][19][20] e sistemas de controle [21][22] estão sendo estudadas mais recentemente em conjunto com computação estocástica, além de serem aplicações muito relacionadas com computação analógica.

É possível ilustrar um exemplo de *Zhang* e *Li* [1], onde uma unidade de controle para um motor indutivo é projetado para integrar vários algoritmos baseados em computação estocástica e diversas redes neurais. Tal controlador é implementado em FPGA e apresenta alta performance e baixo custo de *hardware* em relação a microprocessadores convencionais projetados para a mesma aplicação. A Figura 2.6 mostra um diagrama de blocos para ilustrar o motor em questão.

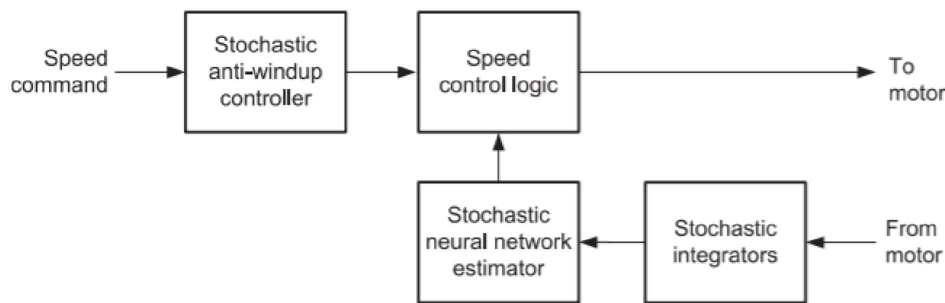


Figura 2.6: Controlador para motor indutivo baseado em computação estocástica [1]

Os *stochastic integrators* executam função no formato $y(n) = x(n) + y(n-1)$ em números no domínio estocástico, enquanto o *stochastic anti-windup controller* implementa um complexo algoritmo cuja finalidade é limitar qualquer mudança no comando de velocidade (*input* do sistema) que leve o motor a operar de maneira imprópria. O módulo *stochastic neural network estimator* implementa um processamento *feedforward* em tempo real, o qual requer uso intenso de recursos computacionais. Uma operação realizada por esse módulo computacionalmente custosa, por exemplo, é o cálculo da função tangente hiperbólica, conhecida como função *tansig* definida por

$$tansig(x) = \frac{2}{(1 - e^{-2x})} - 1. \tag{2.10}$$

Além de otimizar custo de performance do sistema, o projeto do computador estocástico de *Zhang e Li* apresentou vantagens também em redução de tamanho dos circuitos e esforço para verificação.

Outra área promissora para computação estocástica é a área de processamento de imagens. Operações feitas em *pixels* de uma imagem de entrada são comuns em aplicações de manipulação de imagens. Apesar dessas operações, em sua maioria, serem simples, a grande quantidade de *pixels* envolvidos em determinadas transformações ou imagem de entrada tornam o processo extremamente intenso em termos de computação. Conforme demonstrado por *Hammadou et al.* [23], implementado-se funções de processamento de imagem utilizando computação estocástica é possível obter um sistema de baixo custo e altamente paralelizado

2.2 Redes Neurais

As redes neurais foram inspiradas pelo cérebro humano, pela capacidade deste de realizar computações de modo totalmente diferente de computadores digitais convencionais. O cérebro possui uma estrutura complexa, não-linear e altamente paralela, além de ser capaz de organizar seus componentes, chamados de neurônios, a fim de realizar determinadas operações como, por exemplo, reconhecimento de padrões, percepção e controle motor de maneira mais rápida que computadores atuais [24].

Dessa forma, a modelagem de redes neurais foi feita tomando como base o funcionamento do cérebro e sua organização em pequenas unidades de processamento, os neurônios, de modo que a estrutura resultante se tornasse massivamente paralela e distribuída, com natural propensão a armazenar conhecimento através de experiências passadas. As semelhanças entre o modelo de rede neural e o cérebro se dá em dois aspectos [24].

1. O conhecimento é adquirido através do ambiente no qual a estrutura está inserida através de um processo de aprendizagem.
2. A interconexão entre as unidades de processamento, conhecidos como pesos sinápticos, são utilizados para armazenar o conhecimento adquirido.

O processo de aprendizagem de uma rede neural artificial é conhecido como algoritmo de aprendizagem, uma função capaz de alterar os pesos das interconexões entre os neurônios com objetivo de atingir um determinado resultado [24].

2.2.1 Benefícios no uso de redes neurais artificiais

A utilização de redes neurais oferece as seguintes capacidades e propriedades:

Não linearidade As unidades de processamento podem ser lineares ou não lineares. Caso uma rede neural cujas interconexões são feitas por neurônios não lineares,

essa rede naturalmente é não linear. Dessa forma, é possível distribuir pela rede o comportamento não linear dos sinais de entrada, em situações que tais sinais sejam originados de fontes não lineares.

Mapeamento Entrada-Saída A rede neural aprende através da apresentação de sinais de entrada e comparação de sua saída com a saída esperada.

Adaptatividade Redes neurais são construídas com a característica de adaptar-se ao ambiente em que estão, dessa forma é possível retreinar uma rede caso haja alguma pequena mudança no ambiente anterior, ou alterar os pesos em tempo real em ambientes não-estacionários.

Resposta Evidencial Tratando-se de classificação de padrões, uma rede neural pode fornecer informação não apenas de qual padrão escolher, mas também a confiança na decisão tomada. Tal característica possibilita rejeitar padrões ambíguos, aumentando a performance de classificação da rede.

Informação Contextual O conhecimento é representado pela estrutura e estado de ativação da rede neural. Dessa forma, todo neurônio pode ser afetado por qualquer outro neurônio daquela estrutura.

Tolerância a falhas Uma rede neural implementada em hardware, por exemplo, tem o potencial a ser tolerante a falha, ou capaz de manter uma computação robusta sob situações adversas, pois a queda de performance decai levemente sob tais circunstâncias. Isso ocorre pois a rede neural naturalmente distribui e armazena a informação armazenada por toda a rede.

Implementação VLSI A natureza massivamente paralela de rede neural a torna capaz de computação extremamente rápida para algumas tarefas. Tal característica permite a implementação de redes neurais utilizando tecnologia *very-large-scale-integrated* (VLSI).

Uniformidade de Análise e Construção Redes neurais beneficiam-se da universalidade como processadores de informações. Tal afirmação é possível pois, para resolver problemas com redes neurais, utiliza-se a mesma notação para cada um deles. Neurônios, por exemplo, são comuns a qualquer rede neural. Teorias e algoritmos de aprendizado podem ser compartilhados com outras aplicações de rede neural, além de redes modulares, que podem ser construídas a partir de integrações entre módulos até a construção final da rede.

2.2.2 Neurônio

Neurônios são unidades de processamento de informação essenciais para a implementação e operação de uma rede neural. A Figura 2.7 mostra um modelo de neurônio artificial. É possível observar três elementos neuronais básicos:

1. Conexões (ou sinapses) entre o neurônio e as entradas x_i, as quais são ponderadas por pesos w_i.
2. Um somador para operar sobre cada sinal de entrada ponderado pelo peso de sua sinapse. Tal operação é vista como uma combinação linear.
3. Uma função de ativação não-linear φ para limitar a amplitude do sinal de saída neuronal. Tipicamente, a saída normalizada de um neurônio costuma ser o intervalo fechado [0, 1] ou [-1, 1].

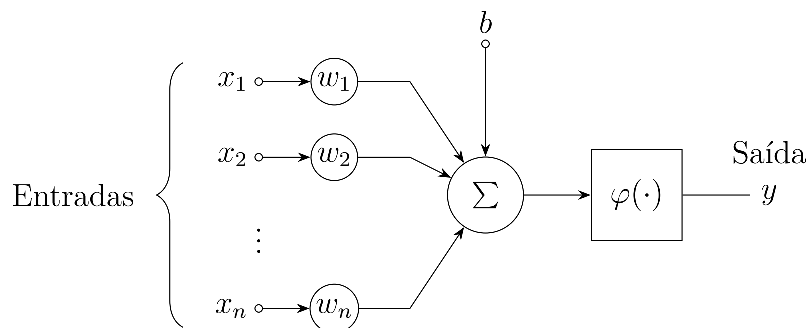


Figura 2.7: Modelo de um neurônio artificial.

O modelo acima, além das entradas x, inclui também uma entrada chamada *viés*, denotado normalmente como b. Tal termo afeta a rede aumentando ou diminuindo a entrada da função de ativação φ.

É possível descrever matematicamente um neurônio utilizando a seguinte equação

$$y = \varphi\left(b + \sum_{i=1}^{n} x_i w_i\right) \tag{2.11}$$

onde y é o sinal de saída resultante; b é o *viés*; φ é a função de ativação; x_1, x_2, ..., x_n são os sinais de entrada e w_1, w_2, ..., w_n são os pesos sinápticos.

O resultado da combinação linear formada pela multiplicação das entradas pelos pesos é chamado de potencial de ativação, o qual é ativado pela função de ativação com a finalidade de limitar o sinal de saída do neurônio a um intervalo limitado, de modo que seja imitado o comportamento dos neurônios biológicos [24].

2.2.3 Função de Ativação

As funções de ativação são funções matemáticas não-lineares, onde temos as principais descritas a seguir.

Threshold

Essa função de ativação é modelada matematicamente por

$$\varphi(u) = \begin{cases} 1, & \text{se } u \geq 0 \\ 0, & \text{se } u < 0 \end{cases} \tag{2.12}$$

a qual limita a saída do neurônio ao intervalo fechado $[0, 1]$. Tal mapeamento força potenciais negativos a 0 e potenciais positivos em 1.

A Figura 2.8 ilustra o comportamento da função *threshold*.

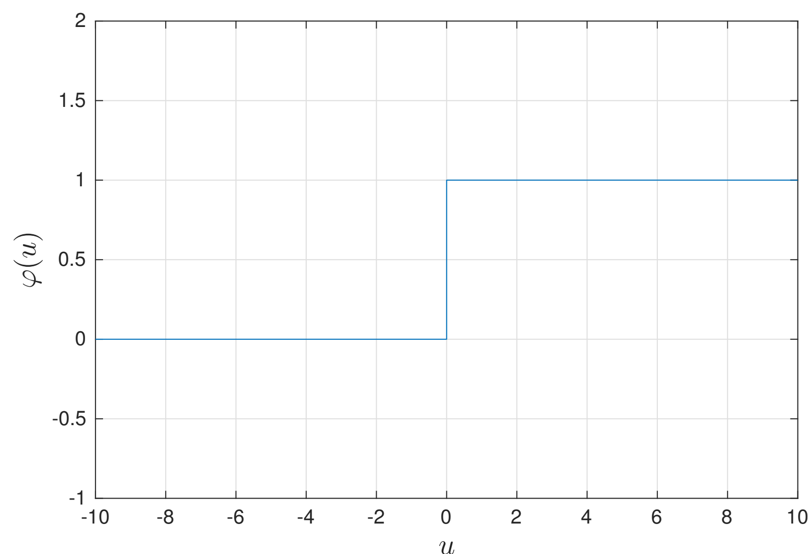


Figura 2.8: Função Threshold.

Aplicando-se a função *threshold* como função de ativação de um neurônio, obtemos um modelo conhecido como Modelo de McCulloch-Pitts [24].

Modelos alternativos foram propostos para essa função, especialmente em relação ao intervalo. O intervalo $[-1, 1]$ é um exemplo de alternativa comum onde varia-se o intervalo da função.

$$\varphi(u) = \begin{cases} 1, & \text{se } u \geq 0 \\ -1, & \text{se } u < 0 \end{cases} \tag{2.13}$$

Sigmoid

A função *Sigmoid* é a mais comum função de ativação utilizado em projetos de redes neurais artificiais. Tal função apresenta um gráfico em forma de *S*, permitindo um balanço entre comportamento linear e não linear.

Um exemplo de função sigmoidal é a função logística, definida da seguinte forma

$$\varphi(u) = \frac{1}{1 + e^{-au}} \tag{2.14}$$

na qual a é o termo que determina a inclinação da curva no gráfico. A Figura 2.9 ilustra a função logística com a variado, de modo que se perceba a mudança na inclinação.

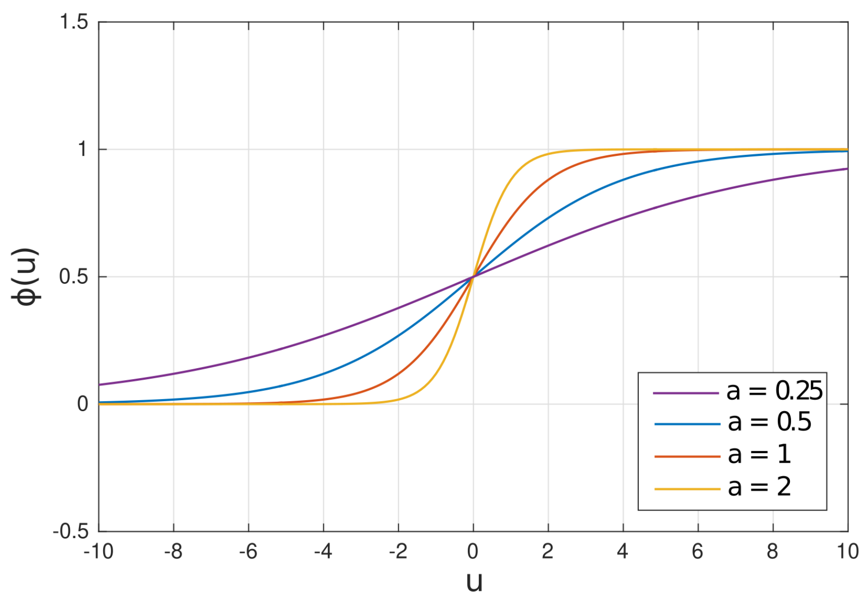


Figura 2.9: Função Logística.

Nota-se que, para valores cada vez maiores de a, a função sigmoide aproxima-se da função de *threshold*.

Uma alternativa à função logística é a tangente hiperbólica, a qual é mais adequada em casos onde deseja-se limitar o resultado a um intervalo bipolar. Uma função de ativação que seja baseada na tangente hiperbólica pode ser apresentada da seguinte forma

$$\varphi(u) = \frac{e^{u} - e^{-u}}{e^{u} + e^{-u}} \tag{2.15}$$

na qual o termo a, novamente, determina a inclinação da curva, como pode ser observado na Figura 2.10. Como é possível observar, o resultado da função hiperbólica é limitado ao intervalo (-1, 1).

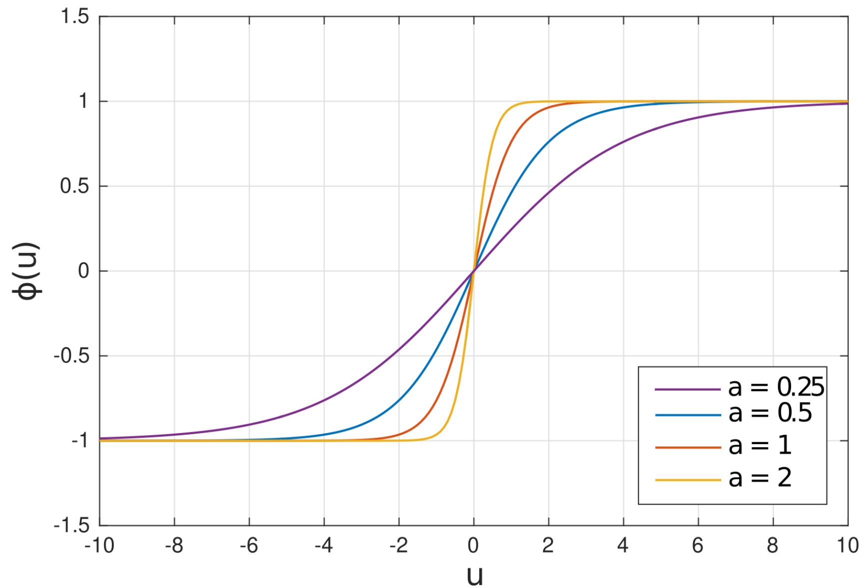


Figura 2.10: Tangente Hiperbólica.

O algoritmo de aprendizado que será abordado mais a frente, conhecido como técnica de retropropagação de erros, utiliza a derivada da função de ativação, o qual pode aumentar a complexidade do projeto digital da rede neural.

A derivada de uma função sigmoidal é facilmente calculável com base na primitiva. A função a seguir apresenta o modelo matemático para se obter a derivada da função logística

$$\frac{d}{du}\varphi(u) = \varphi(u)(1 - \varphi(u))a. \tag{2.16}$$

Já para a derivada da tangente hiperbólica, o modelo matemático é o seguinte

$$\frac{d}{du}\varphi(u) = a(1 - \varphi(u)^2). \tag{2.17}$$

A Figura 2.11 ilustra o gráfico da derivada da função tangente hiperbólica.

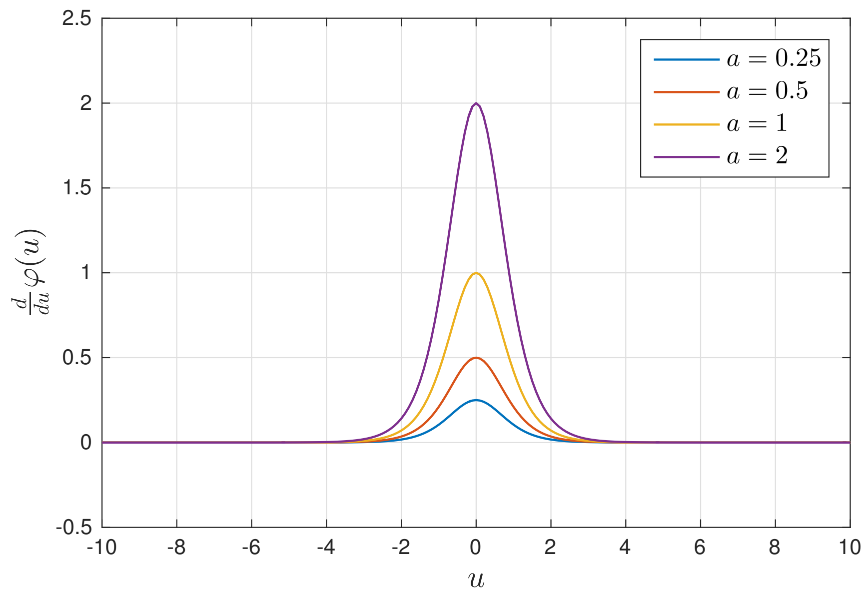


Figura 2.11: Derivada da Tangente Hiperbólica.

2.2.4 Estruturas de Redes Neurais

A forma como os neurônios são dispostos e conectados em uma rede neural está intimamente ligado ao algoritmo de aprendizado utilizado na fase de treinamento da rede em questão. Tal forma de organização dos neurônios para modelagem de uma rede neural artificial é chamada de *arquitetura*.

De modo geral, os neurônios são modelados de acordo com a Figura 2.12.

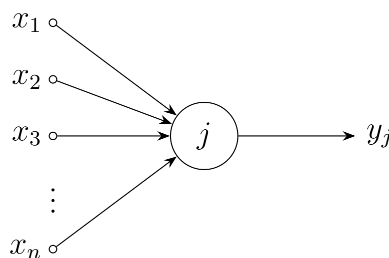


Figura 2.12: Neurônio com entradas e saídas.

O neurônio apresentado possui sinais x de entrada, não incluindo o *viés*, uma vez que este é considerado parâmetro interno de cada unidade. O sinal y_j representa a saída processada do neurônio e pode servir como sinal de entrada para neurônios seguintes ou sinal de saída da rede.

Os modelos arquiteturais mais clássicos de redes neurais baseiam-se na divisão por camadas, a qual consiste numa separação da rede em camadas com uma quantidade n de neurônios que possuem sinapses com todos os neurônios da camada seguinte.

As redes *Single-Layer*, ou redes de camada única, mostrada na Figura 2.13, possuem duas camadas em sua topologia: a camada de entrada, composta pelos sinais de entrada da rede neural, e a camada de saída, a qual possui neurônios que recebem os sinais da camada de entrada e geram os sinais processados de saída. Tal arquitetura é a mais simples, visto que possui apenas uma camada de processamento.

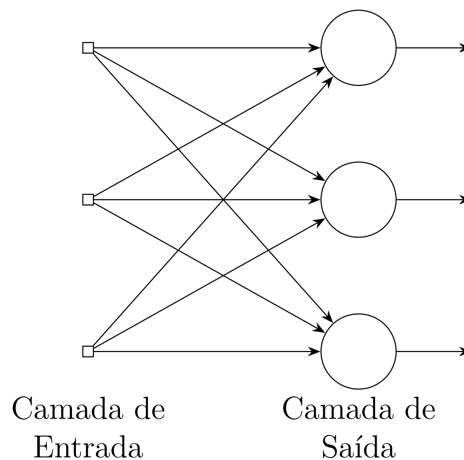


Figura 2.13: Exemplo de rede neural artificial de uma camada.

À medida que acrescenta-se o número de camadas de processamento é possível resolver problemas mais complexos. As camadas adicionais, conhecidas como *Hidden-Layers*, ou camadas ocultas, permite a computação de problemas que não são possíveis de se resolver com uma camada. O nome dado a essa arquitetura é *Multi-Layer Network*, ou rede multi-camada. A Figura 2.14 ilustra um exemplo de rede neural artificial multi-camada.

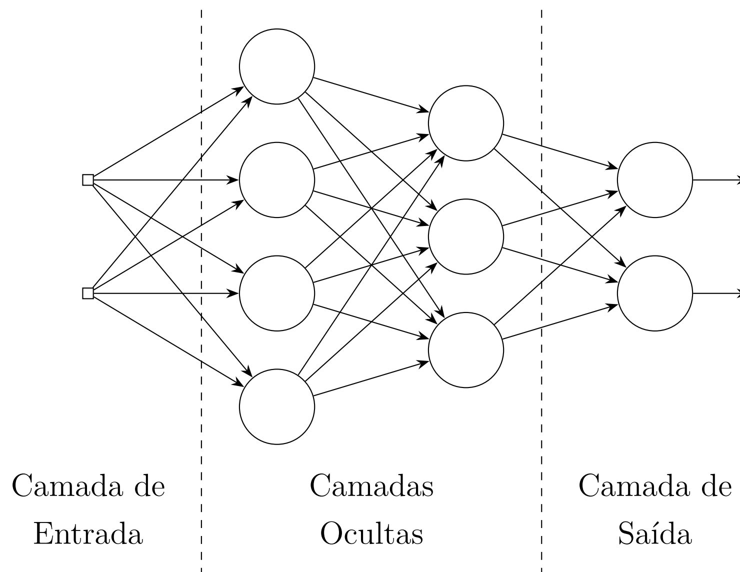


Figura 2.14: Exemplo de rede neural artificial multi-camada.

A rede multi-camada é comumente chamada também de rede *feedforward*, pois há um fluxo direto da camada de entrada até a camada de saída da rede, não havendo conexões cíclicas entre os neurônios. Além disso, uma rede *feedforward* pode ser totalmente conectada quando cada neurônio apresenta uma sinapse com cada neurônio da camada seguinte. Tal organização da rede é denominada Perceptron Multicamadas (do inglês *Multilayer Perceptron*) (MLP).

Para identificar a estrutura de uma rede, é comum associar a organização de uma arquitetura *feedforward* a uma nomenclatura que corresponde a estrutura dela. Tomando-se como exemplo uma rede neural MLP com 2 sinais na camada de entrada, 4 neurônios na primeira camada oculta, 3 neurônios na segunda camada oculta e 2 neurônios na camada de saída é possível identificá-la pela definição "2-4-3-2", onde os números identificam a quantidade de neurônios em cada camada, conforme rede neural apresentada na Figura 2.14.

Redes Neurais Recorrentes

Uma classe diferente de redes neurais artificiais são as redes neurais recorrentes, uma vez que sua principal característica é a retroalimentação de camadas. Diferentemente de redes com arquitetura *feedforward*, as redes recorrentes possuem uma unidade de memória que, de forma análoga ao neurônio como unidade de processamento, esses elementos são vistos como neurônios que armazenam a saída de uma unidade de processamento.

Redes neurais recorrentes introduzem a ideia de tempo, pois sinais de saída de uma determinada iteração são armazenados e utilizados como sinais de entrada na próxima iteração. Tomando como exemplo um neurônio cujo sinal de entrada no instante de tempo

t é definido por $x(t) = (x_1, x_2, ..., x_n)$, ao colocar essa unidade de atraso na saída $y(t)$, no instante de tempo $t + 1$ a saída do neurônio em questão será $y(t + 1)$ e a saída da unidade de atraso será $y(t)$. A Figura 2.15 ilustra um modelo da unidade de atraso em questão [8].

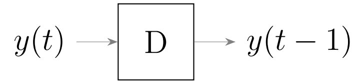


Figura 2.15: Unidade de atraso.

Tais unidades de atraso são inseridos em determinadas camadas da rede neural, formando uma nova camada de memória denominada camada de contexto. Os elementos da camada de contexto são visto como neurônios regulares. Os sinais de entrada e saída da camada de contexto são, respectivamente, a saída de uma determinada camada e a entrada dessa mesma camada. A Figura 2.16 demonstra a estrutura de uma rede neural recorrente.

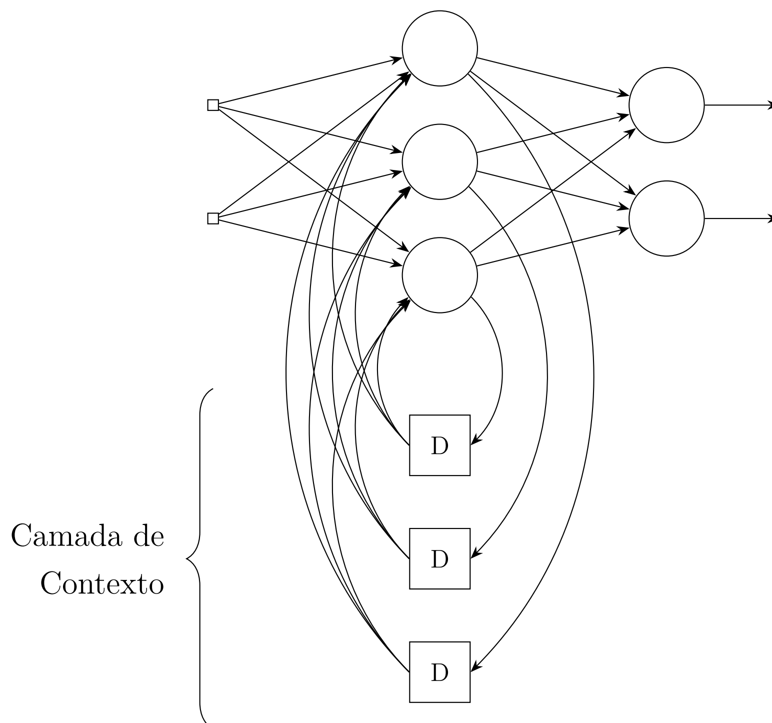


Figura 2.16: Exemplo de rede neural recorrente de Elman.

A Figura 2.16 ilustra uma rede neural proposta por Jeffrey L. Elman [8], o qual o modelo matemático para tal rede é

$$h_t = \varphi\left(b + \sum_{i=1}^{n} x_i w_i + h_{t-1} u_i\right) \tag{2.18}$$

onde h_t é a saída processada de uma camada escondida; φ é a função de ativação da camada; x_i e w_i são, respectivamente, os sinais de entrada da camada escondida e os pesos sinápticos; h_{t-1} e u_i são, respectivamente, as saídas da camada escondida no instante de tempo $t-1$ e os pesos sinápticos entre a camada de contexto e a camada escondida [8].

Existe uma variação de rede neural recorrente, conhecida como rede neural de Jordan [25], a qual o modelo matemático é dado por

$$h_t = \varphi\left(b + \sum_{i=1}^{n} x_i w_i + y_{t-1} u_i\right). \tag{2.19}$$

A Figura 2.17 ilustra uma rede recorrente de Jordan com 3 neurônios na camada escondida.

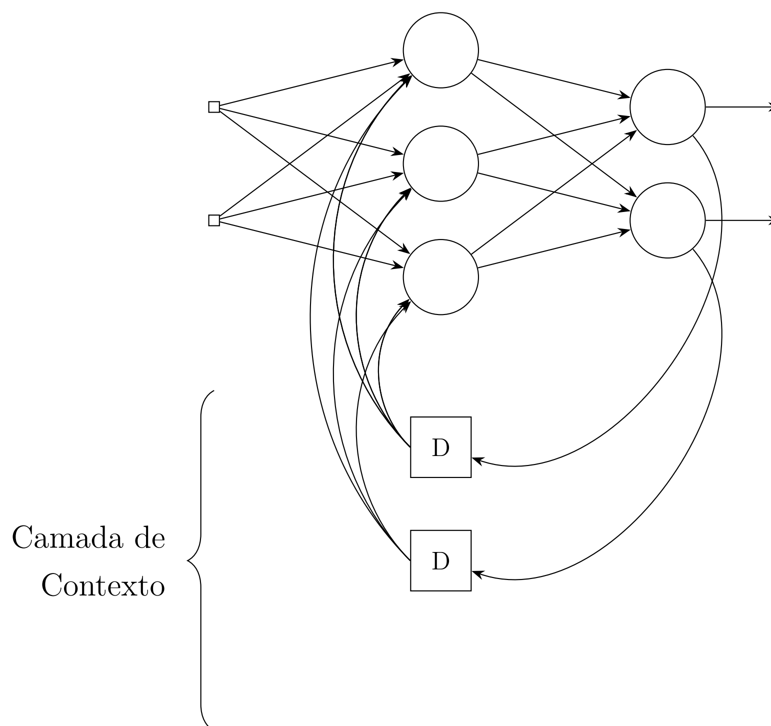


Figura 2.17: Exemplo de rede neural de recorrente de Jordan.

Da mesma forma que a camada de contexto de Elman recebe como entrada a saída da camada escondida, a camada de contexto de Jordan, chamada de camada de estado, recebe a saída $y(t)$ da rede para retroalimentar a camada escondida no instante de tempo $t+1$.

2.2.5 Treinamento por Retropropagação de Erros

O algoritmo de treinamento utilizado neste trabalho é chamado Gradiente Descendente com Momentum (GDM) e seu funcionamento acontece por meio de retropropagação de

erros, ou *Error Backpropagation*. Tal técnica é utilizada em conjunto com um método de otimização na atualização dos pesos sinápticos de uma rede neural com o objetivo de fazê-la compreender determinado comportamento e encontrar padrões de saída em relação às entradas [26].

Ao tentar encontrar um ponto mínimo para determinada função, diversas iterações são feitas de modo que, a cada nova iteração, a direção inversa do gradiente da função é tomada. De modo que fique mais clara a compreensão, tomemos uma função *F(x)*, diferenciável, então a iteração $n + 1$ ocorre de acordo com

$$x_{n+1} = x_n - \eta \nabla F(x_n) \tag{2.20}$$

onde x_n é a posição corrente, ∇F é o gradiente da função, e η é o fator de otimização, conhecido como *Fator de Aprendizagem*.

A fim de ilustrar o funcionamento do treinamento, considere a seguinte rede neural de um neurônio com duas entradas x_1 e x_2 e uma saída y, mostrada na Figura 2.18.

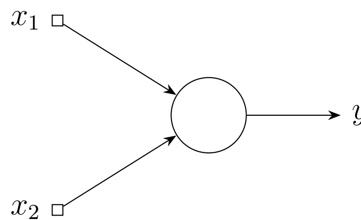


Figura 2.18: Rede neural de um neurônio.

As entradas x_1 e x_2 resultam em uma saída y. Para de mensurar a diferença entre a saída da rede e a saída esperada y_e, utiliza-se a equação do erro quadrático

$$E = \frac{1}{2}(y_e - y)^2 \tag{2.21}$$

A partir da Equação 2.11, o erro E é definido em função de um determinado peso w_i. Calcula-se seu gradiente da seguinte forma

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial o}\frac{\partial o}{\partial u}\frac{\partial u}{\partial w_i} \tag{2.22}$$

onde u é o potencial de ativação do neurônio cuja saída é o. É importante observar, a partir da equação do potencial de ativação dado por

$$u = b + \sum_{i=1}^{n} x_i w_i \tag{2.23}$$

que apenas a entrada x_i do neurônio depende de w_i. Dessa forma, a última derivada parcial é dada simplesmente por

$$\frac{\partial u}{\partial w_i} = x_i. \tag{2.24}$$

O restante dos termos que compõem o gradiente do erro são independentes de um determinado peso sináptico, sendo reutilizados no cálculo de todos os incrementos relativos a um certo neurônio. Eles formam o que é conhecido como "erro local" de um neurônio, expressado matematicamente por

$$\delta_j = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial u_j} = \upsilon_j\ \dot{\varphi}(u_j) \tag{2.25}$$

onde δ_j é o erro local do neurônio j com saída o_j e potencial de ativação u_j. Nota-se que a derivada da função de ativação, $\dot{\varphi}(u_j)$, do neurônio é o segundo termo do erro local, tal fato mostra como a escolha de uma função facilmente diferenciável é importante para a implementação do algoritmo em questão.

O termo υ é conhecido como *erro retropropagado*. Tal termo relaciona mudanças no erro quadrático da saída final com as mudanças na ativação de um determinado neurônio. O cálculo deste fator difere de acordo com a camada do neurônio em questão. É importante lembrar que na camada de saída, as próprias ativações dos neurônios são as saídas da rede presentes no erro quadrático. Dessa forma, o cálculo de υ é feito utilizando

$$\upsilon = \frac{\partial}{\partial y}\frac{(y_e - y)^2}{2} = y - y_e. \tag{2.26}$$

Para camadas mais internas da rede, um método recursivo para o cálculo do erro retropropagado é utilizado. A Equação 2.27 mostra a recursividade aplicada de modo que seja possível obter os erros retropropagados de um neurônio em função dos erros locais das camadas seguintes [26].

$$\upsilon_j = \sum_k \delta_k w_{jk} \tag{2.27}$$

A Equação 2.27 mostra a motivação por trás do nome do algoritmo de aprendizado *Backpropagation*, utilizando-se de erros locais de camadas posteriores retropropagados para camadas anteriores. Por fim, baseando-se na Equação 2.20, o ajuste dos pesos sinápticos é dado a partir da seguinte equação

$$\Delta w_{ij} = -\eta\ \delta_j\ o_i \tag{2.28}$$

onde Δw_{ij} define o incremento do peso sináptico conectando o neurônio i ao neurônio j; δ_j o erro local do neurônio j; e o_i a ativação do neurônio i caso a camada i seja uma camada anterior. Caso a camada anterior seja a camada de entrada, o_i é a entrada x_i da rede.

Para realizar o treinamento de uma Rede Recorrente, algumas modificações nos algoritmos de *feedfoward* e *backpropagation* devem ser feitos.

Ao apresentar a rede uma sequência de entradas $x = [x_1, x_2, ..., x_n]$, a propagação de sinais da rede acontecem da mesma forma como numa rede de *feedfoward* tradicional. Entretanto, com a adição da camada de contexto, a ativação da camada oculta é propagada para a camada seguinte e para a camada de contexto. Essa última recebe o sinal de ativação da camada oculta por cópia, ou seja, o peso da sinapse entre a camada oculta e a camada de contexto é sempre 1.

O processo de *backpropagation* ocorre de forma um pouco mais complicada que o *feedfoward*, que basicamente copia os valores de ativação da camada oculta para a camada de contexto. No *backpropagation* os valores de ativação do passo anterior são considerados como entrada do passo corrente.

Para o cálculo de ativação da camada escondida considerando as entradas da camada de contexto, é utilizada equação

$$h_t = \varphi\left(b + \sum x_t w_h + \sum h_{t-1} w_c\right) \tag{2.29}$$

sendo h_t o sinal de ativação de uma camada escondida, b o viés de entrada, x_t as entradas no tempo t, w_h o peso da sinapse entre a camada de entrada e a camada escondida, h_{t-1} o sinal de ativação da camada escondida no passo anterior e w_c o peso das sinapses de retroalimentação da camada de contexto para a camada escondida.

A atualização dos pesos da camada de contexto é feito da mesma forma no algoritmo descrito para a rede *feedfoward* tradicional pois a camada de contexto é considerada como uma entrada da camada escondida, logo, as atualizações feitas nos pesos de entrada também são aplicadas nos pesos de contexto.

2.3 Dispositivos Eletrônicos Reconfiguráveis

Esses dispositivos têm como principal característica um circuito formado por um arranjo bidimensional de elementos lógicos programáveis. Estes elementos lógicos possuem a capacidade de computar operações *booleanas*, bem como armazenar tais resultados. Uma classe muito utilizada deste tipo de dispositivo é o Arranjo de Portas Programável em Campo (do inglês *Field-Programmable Gate Array*) (FPGA).

Neste trabalho foi utilizado um FPGA da família *Cyclone IV* [27] como plataforma de desenvolvimento e testes das redes neurais e módulos estocásticos. A Figura 2.19

apresenta, utilizando diagrama de blocos, a estrutura de um elemento lógico deste FPGA. É importante ressaltar que tal estrutura varia de acordo com a família e fabricante do *chip*.

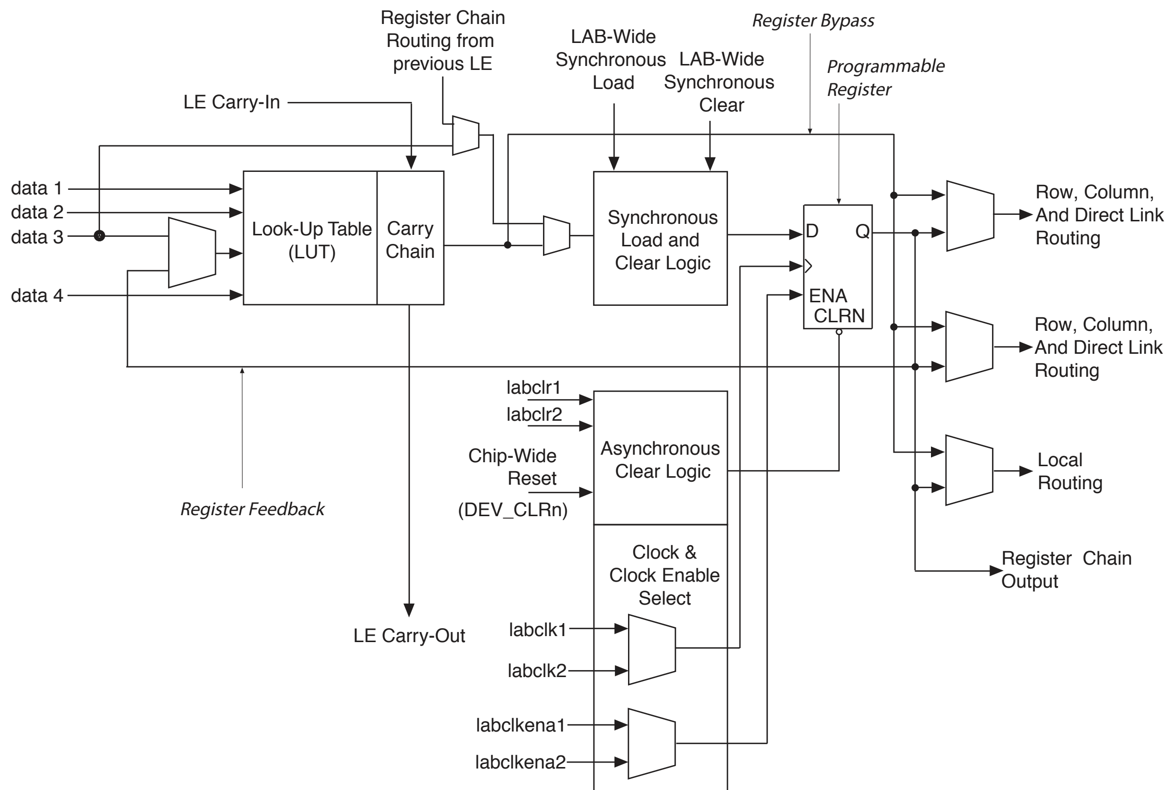


Figura 2.19: Diagrama de blocos do Elemento Lógico do FPGA CycloneIV (Fonte: [27]).

O circuito de um FPGA possui como característica principal para tornar-se reconfigurável uma estrutura chamada *Look-Up Table* (LUT). A LUT é uma estrutura formada por uma pequena memória que associa uma combinação de variáveis de entrada a um determinado resultado. No caso do FPGA *Cyclone IV*, as LUTs são capazes de implementar funções arbitrárias de 4 variáveis. Para programar um *chip* FPGA, é necessário carregar as LUTs de modo que as operações desejadas sejam executadas.

Um elemento lógico, além de realizar operações *booleanas*, contém um *flip-flop* do tipo D, o qual possibilita ao circuito o armazenamento de dados, mostrado como *Programmable Register* na Figura 2.19.

É importante ressaltar que os circuitos reconfiguráveis podem sofrer com atraso em determinadas partes do circuito por conta do roteamento de sinais. Caminhos longos no circuitos aumentam consideravelmente o tempo de percurso dos sinais, fato que limita a máxima frequência de chaveamento do circuito.

Dispositivos reconfiguráveis mais modernos, para certas estruturas, possuem otimizações em suas conexões. Tais estruturas, por exemplo, são somadores e contadores, o que permite cadeias de registradores a atuarem com alta velocidade. Entretanto, circuitos projetados para implementação em FPGA devem levar em conta, além da intrínseca limitação lógica, tamanho de memória e roteamento na frequência máxima de chaveamento.

No capítulo seguinte serão apresentadas estruturas eletrônicas projetas e implementadas no FPGA *Cyclone IV*, e serão analisadas a eficiência de cada estrutura, bem como a quantidade de elementos lógicos para sintetizá-las.

Capítulo 3

Sistema Proposto

Neste capítulo é tratada, de forma detalhada, a implementação da solução proposta para síntese de rede neural recorrente em *hardware* utilizando computação estocástica.

Primeiramente, será abordada uma implementação em *software* da rede neural recorrente, cujo comportamento estocástico será simulado através das operações feitas pela rede, de modo que sejam respeitados os conceitos apresentados no capítulo anterior sobre computação estocástica. Em seguida, será apresentado o circuito digital implementado em *FPGA*, utilizando-se a linguagem de descrição de *hardware SystemVerilog*.

3.1 Implementação em *Software*

O treinamento de redes neurais recorrentes estocásticas pode ser realizado através de algoritmos implementados em software. Neste trabalho, foi utilizado o software MATLAB para a implementação e simulação das redes.

O algoritmo utilizado tem como objetivo simular o comportamento estocástico da rede. Isto é feito através da imposição do intervalo probabilístico aos pesos sinápticos e da utilização da função de ativação estocástica. Se atendo a estas exigências, a implementação em software essencialmente aplica o método estocástico, de modo a aproximar os resultados obtidos via computação estocástica simulada em software com o circuito digital implementado em FPGA.

Uma estrutura contendo informações sobre o estado da rede é instanciada, de modo que cada camada i da rede, de tamanho L_i, tenha os seguintes atributos:

- $Potenciais_i$: um vetor de L_i potenciais de ativação.
- $Ativações_i$: um vetor de L_i ativações.
- $ErrosLocais_i$: um vetor de L_i erros locais.

- *$Pesos_i$*: uma matriz de L_i linhas e $L_{i-1}+1$ colunas, onde cada linha j contém os L_{i-1} pesos sinápticos do neurônio j além do viés correspondente. Para camadas ocultas, o tamanho dessa matriz é L_i linhas e $L_{i-1}+1+L_i$ colunas, onde cada linha j contém os L_{i-1} pesos sinápticos do neurônio em questão e do viés. Considera-se também os pesos sinápticos da camada de contexto da camada oculta i.

- $\Delta Peso_i$: matriz de mesmo tamanho da matriz $Pesos_i$. Tal matriz é somada à matriz $Pesos_i$ na fase de atualização do *backpropagation*. A cada passo, os fatores de correção são armazenados nessa matriz.

- $Contexto_i$: um vetor de L_i ativações da camada i no passo anterior da iteração corrente.

O Código-fonte 3.1 mostra como é gerada a estrutura de dados da rede neural no Matlab, chamada de *layers*.

Código 3.1: Instância da estrutura de dados da rede neural recorrente estocástica no Matlab

```
1
2 % -- Setup the network
3     layerCount = length(layerSizes);
4     layers = struct('size', num2cell(layerSizes));
5
6     for i = 2:layerCount
7         layers(i).potential = zeros(layers(i).size, 1);
8         layers(i).activation = zeros(layers(i).size, 1);
9         layers(i).localError = zeros(layers(i).size, 1);
10        layers(i).synapseCount = layers(i-1).size + 1;
11        if (i ~= layerCount)
12            layers(i).synapseCount = layers(i).synapseCount + layers(i).
    size;
13            layers(i).context = zeros(layers(i).size, 1);
14        end
15        layers(i).weights = 2 * rand(layers(i).size, layers(i).
    synapseCount) - 1;
16        layers(i).dw = zeros(layers(i).size, layers(i).synapseCount);
17    end
```

Com isso, o Algoritmo 1 é capaz de executar o processamento *feedforward* da rede de maneira simples. Os potenciais de ativação são calculados através da multiplicação dos pesos e as ativações são propagadas às camadas seguintes, incluindo as camadas de contexto. O algoritmo possui como entrada os próprios sinais de entrada da rede e as saídas são diretamente definidas pelas ativações dos neurônios da última camada.

Algoritmo 1 *Feedforward* de uma rede L_1-L_2-$\cdots$-L_n

1: $Ativações_1 \leftarrow$ entradas
2: **for** $i = 2 \rightarrow n-1$ **do**
3: $\quad Potenciais_i \leftarrow Pesos_i \times [Ativações_{i-1};\ Contexto_i]$
4: $\quad Potenciais_i \leftarrow Potenciais_i \div (L_{i-1} + 1 + L_i)$
5: $\quad Ativações_i \leftarrow \varphi(Potenciais_i)$
6: $\quad Contexto_i \leftarrow Ativações_i$
7: **end for**
8: $Potenciais_n \leftarrow Pesos_n \times Ativações_{n-1}$
9: $Potenciais_n \leftarrow Potenciais_n \div (L_{n-1} + 1)$
10: $Ativações_n \leftarrow \varphi(Potenciais_n)$
11: saídas $\leftarrow Ativações_n$

O Código-fonte 3.2 apresenta o trecho de código para execução do *feedfoward* no Matlab.

Código 3.2: *Feedforward* no Matlab

```
1 % -- Feedforward
2     function outputs = simulate(inputs)
3 
4         layers(1).activation = inputs';
5         for l = 2:1:layerCount
6             if (l ~= layerCount)
7                 layers(l).potential = layers(l).weights * [layers(l-1).
      activation; layers(l).context; 1];
8             else
9                 layers(l).potential = layers(l).weights * [layers(l-1).
      activation; 1];
10            end
11
12            % Calculate action potential for hidden layer
13            layers(l).potential = layers(l).potential / layers(l).
      synapseCount; %simula o comportamento estocastico
14
15            % Calculate activation for hidden layer and copy context
16            layers(l).activation = activate(layers(l).potential);
17            if (l ~= layerCount)
18               layers(l).context = layers(l).activation;
19            end
20        end
21        outputs = layers(layerCount).activation';
22    end
```

É importante notar que o somatório envolvido no cálculo dos potenciais, presente na

linha 13, simula o comportamento estocástico, sendo efetuada a divisão pelo número de entradas. Necessita-se dessa simulação pois é o comportamento que será esperado em hardware, onde as operações serão implementadas de forma puramente estocástica.

O *backpropagation* é apresentado no Algoritmo 2 da mesma forma como foi apresentado o *feedfoward*.

Algoritmo 2 *Backpropagation* em uma rede L_1-L_2-$\cdots$-L_n

1: $erros \leftarrow$ saídas esperadas $-$ $Ativações_n$
2: **for** $i = n \rightarrow 2$ **do**
3: $ErrosLocais_i \leftarrow \dot{\varphi}(Potenciais_i) \circ erros$
4: $erros \leftarrow Pesos_i \times ErrosLocais_i$
5: $erros \leftarrow erros \div L_i$
6: **end for**

Calcula-se os erros locais através da derivada da função de ativação, aplicada aos potenciais de cada neurônio. Os erros são retropropagados às camadas anteriores e, nos casos das camadas ocultas, também retropropagam para as camadas de contexto. A entrada do algoritmo é o vetor de saída esperado de um determinado conjunto de treino.

O processo de atualização dos pesos sinápticos é mostrado no Algoritmo 3.

Algoritmo 3 Atualização dos pesos sinápticos de uma rede L_1-L_2-$\cdots$-L_n

1: **for** $i = 2 \rightarrow n - 1$ **do**
2: $Grad \leftarrow (ErrosLocais_i \times [Ativações_{i-1}\ \ Contexto_i])$
3: $\Delta Pesos_i \leftarrow (Grad \times \eta) + (\Delta Peso_i \times \alpha)$
4: $Pesos_i \leftarrow Pesos_i + \Delta Pesos_i$
5: $Pesos_i \leftarrow \max(\text{-}1, \min(Pesos_i, 1))$
6: $\Delta Peso_i \leftarrow \Delta Pesos_i$
7: **end for**
8: $Grad \leftarrow (ErrosLocais_n \times Ativações_{n-1})$
9: $\Delta Pesos_n \leftarrow (Grad \times \eta) + (\Delta Peso_n \times \alpha)$
10: $Pesos_n \leftarrow Pesos_n + \Delta Pesos_n$
11: $Pesos_n \leftarrow \max(\text{-}1, \min(Pesos_n, 1))$
12: $\Delta Peso_n \leftarrow \Delta Pesos_n$

Nota-se a imposição do intervalo [-1, 1] dos pesos, uma vez que tal imposição é feita com base nas limitações da computação estocástica, ou seja, tais pesos poderão ser utilizados diretamente na implementação de hardware, uma vez que todas as operações obedecem as características das operações estocásticas e os limites numéricos estão dentro do intervalo do domínio trabalhado.

O treinamento é realizado através da iteração da rede recebendo o conjunto de treino. Então o *feedfoward* é feito, seguido do *backpropagation* e, por fim, a atualização dos pesos.

Esse processo é repetido até que um critério de parada seja atingido. O critério de parada utilizado neste trabalho foi a quantidade de épocas, ou iterações, feita pela rede.

É interessante notar que há alguns termos não apresentados anteriormente na atualização de pesos e na própria estrutura da rede. O gradiente é multiplicado ao fator de aprendizado η e somado a uma multiplicação de termos. Tal multiplicação de termos envolve o *Momentum* (α) e a última atualização de pesos feita $\Delta Peso_i$. Essa multiplicação é uma adaptação do tradicional Gradiente Descendente, conhecida como Gradiente Descendente com Momentum. A utilização do *momentum* atenua oscilações no processo de iteração, resultando no processo de aprendizado mais rápido devido a otimização no processo de convergência [28].

A implementação no Matlab do algoritmo *backpropagation* é apresentado no Código-fonte 3.3.

Código 3.3: *Backpropagation* no Matlab

```
1 % -- Backpropagation
2     function perf = adapt(inputs, outputs)
3
4         o = simulate(inputs);
5         errors = (outputs - o) / 2;
6         perf = sum(errors .^ 2);
7         errors = errors';
8
9         for l = layerCount:-1:2
10           % Calculate local errors for hidden layer
11            layers(l).localError = derive(layers(l).potential) .* errors;
12
13            % Calculate delta weights for hidden layer
14            if(l ~= layerCount)
15                dw = layers(l).localError * [layers(l-1).activation'
      layers(l).context' 1];
16            else
17                dw = layers(l).localError * [layers(l-1).activation' 1];
18            end
19
20            dw =  (parameters.MomentumFactor * layers(l).dw) + (dw *
      parameters.LearningFactor);
21            dw = dw/2;
22
23            % Update weights for hidden layer
24            layers(l).weights = max(-1, min(1, layers(l).weights + dw));
25            layers(l).dw = dw;
26
```

```
27              % Back-propagate errors to hidden layer
28              errors = layers(l).weights(:,1:layers(l-1).size)' * layers(l)
      .localError;
29              errors = errors / layers(l).size;
30          end
31      end
```

A partir de um conjunto de entradas com sua saída esperada, é possível utilizar as funções e estrutura apresentadas para treinar uma rede neural recorrente simulando o computação estocástica.

3.2 Implementação em FPGA

Esta seção detalha a implementação de uma rede neural recorrente, a qual foi empregada uma arquitetura estocástica. A implementação dos circuitos foi desenvolvida no trabalho de *Carvalho* [29]. A seguir, serão apresentados detalhes do sistema.

O sistema implementado está organizado em dois módulos principais, que estão representados na Figura 3.1

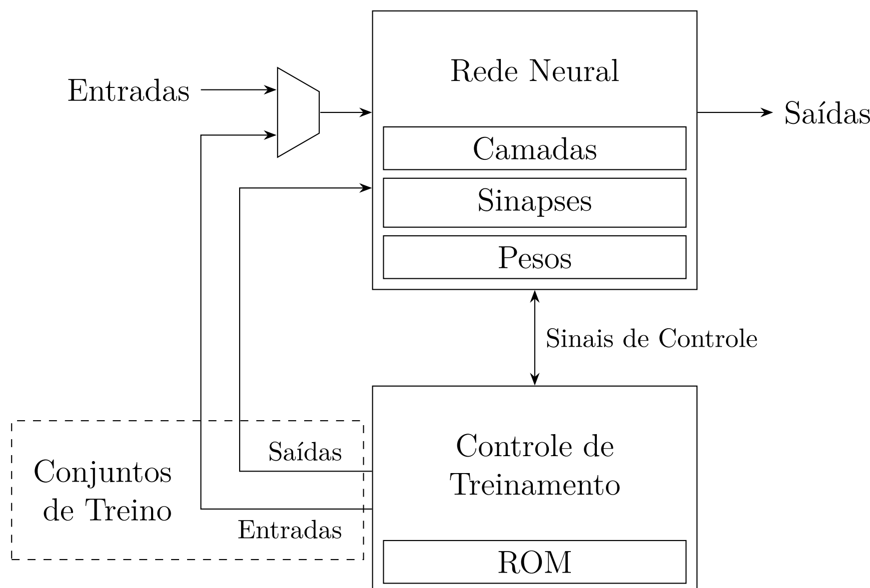


Figura 3.1: Módulos principais do sistema.

O módulo *Rede Neural* é responsável pelo processamento dos sinais de entrada e geração dos sinais de saída, onde tal processamento ocorre de forma puramente estocástica, ou seja, todas as sequências de *bits* são interpretadas como probabilidades e operadas utilizando os conceitos apresentados no Capítulo 2. É neste módulo que são implementados os neurônios estocásticos, as sinapses das redes, as camadas de entrada, ocultas e de saída.

Por outro lado, o módulo *Controle de Treinamento* é responsável pela apresentação de sinais de entrada e saída para a rede neural, estimulando assim a adaptação dos pesos da rede utilizando o algoritmo *Backpropagation*.

O treinamento de redes neurais recorrentes estocásticas em circuitos reconfiguráveis não faz parte do escopo deste trabalho.

3.2.1 Módulo Rede Neural

O Módulo de Rede Neural foi implementado de maneira modular, onde é possível organizar a estrutura da rede neural facilmente. A Figura 3.2 apresenta o diagrama em blocos do esquema de *feedforward* de uma rede neural recorrente.

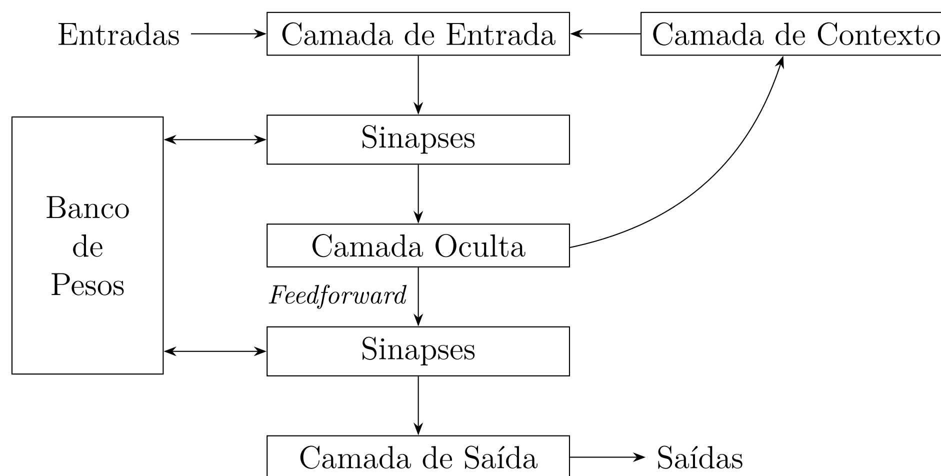


Figura 3.2: Esquema de *feedforward* de uma rede neural recorrente com uma camada oculta.

Os neurônios estocásticos fazem parte da Camada Oculta e também da Camada de Saída e são implementados de modo que os algoritmos de *feedforward* e retropropagação de erros ocorram conforme modo de operação da rede neural.

As "Sinapses" são estruturas que contém a lógica combinatória dos fluxos da rede e realizam conexões entre camadas anteriores e camadas posteriores, de modo que seja possível implementar redes com maior número de camadas ocultas apenas reestruturando o circuito. Operações como somas ponderadas para cálculo dos valores de ativação da camada anterior são realizadas por esse circuito.

O "Banco de Pesos" é um módulo de grande importância para o funcionamento da Rede Neural. É nele que são armazenados e manipulados os pesos sinápticos. Esse banco foi estruturado como um banco de registradores de *8 bits* e sua função é armazenar os pesos sinápticos na forma binária.

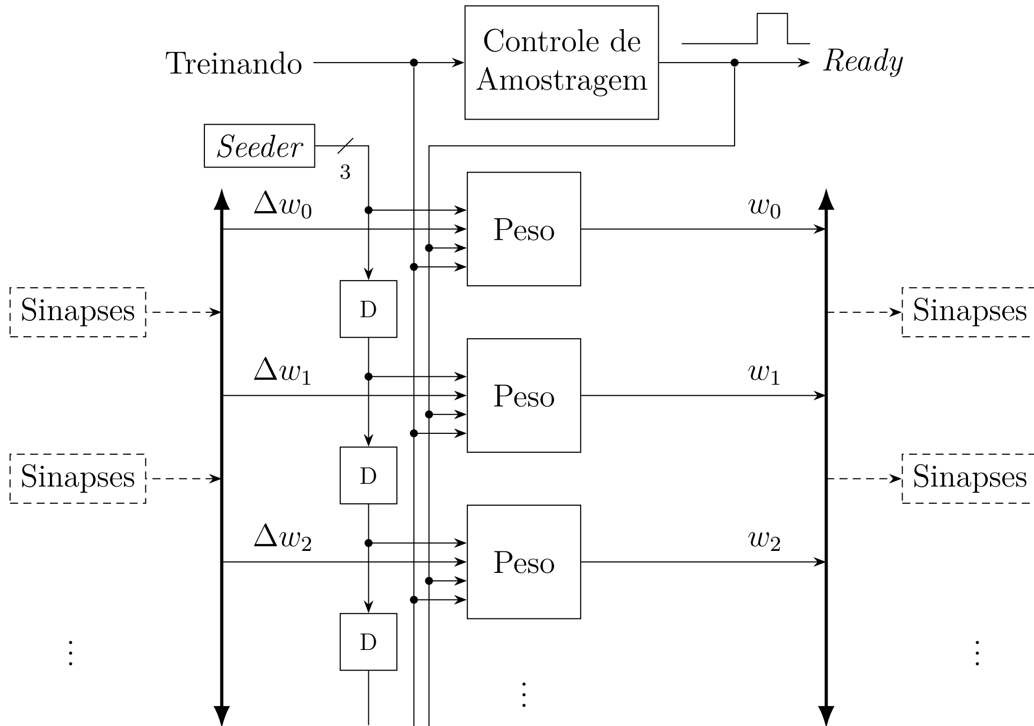


Figura 3.3: Funcionamento interno do Banco de Pesos.

A Figura 3.3 ilustra as conexões dos circuito implementado para o banco de pesos, o qual fornece às sinapses da rede neural o conjunto de pesos estocásticos, bem como recebe os valores Δw_n para ajuste dos pesos durante a execução do algoritmo de treinamento.

Os "Pesos", mostrados na Figura 3.3, representam instâncias de armazenamento de peso sináptico de forma binária. Possuem, também, pequenos circuitos que possibilitam a conversão de números binários para a forma de números estocásticos. A Figura 3.4 detalha a organização interna desta estrutura.

A conversão dos valores de cada peso binário para sua forma estocástica é feita pelo circuito "*Seeder*", mostrado na Figura 3.3 e detalhado na Figura 3.5, o qual é implementado através de um gerador por multiplexação. Esse gerador funciona através da multiplexações de cada *bit* que compõe uma quantidade binária, de modo que o valor final seja um *bit stream* estocástico com valor equivalente à quantidade de entrada. Para que tal processo aconteça, a raiz (*seed*) utilizado pelo multiplexador deve ser formada por sequências estocásticas de determinadas probabilidades que devem ser geradas externamente e então repassadas ao "*Seeder*".

Cada *seed* é gerada por este módulo do banco de pesos e cópias com atraso, ou seja, cópias que são efetivamente não correlacionadas, são redirecionadas a cada peso, o que elimina a necessidade de mais circuitos geradores de endereço. A Figura 3.5 ilustra o circuito interno deste módulo por diagrama de blocos.

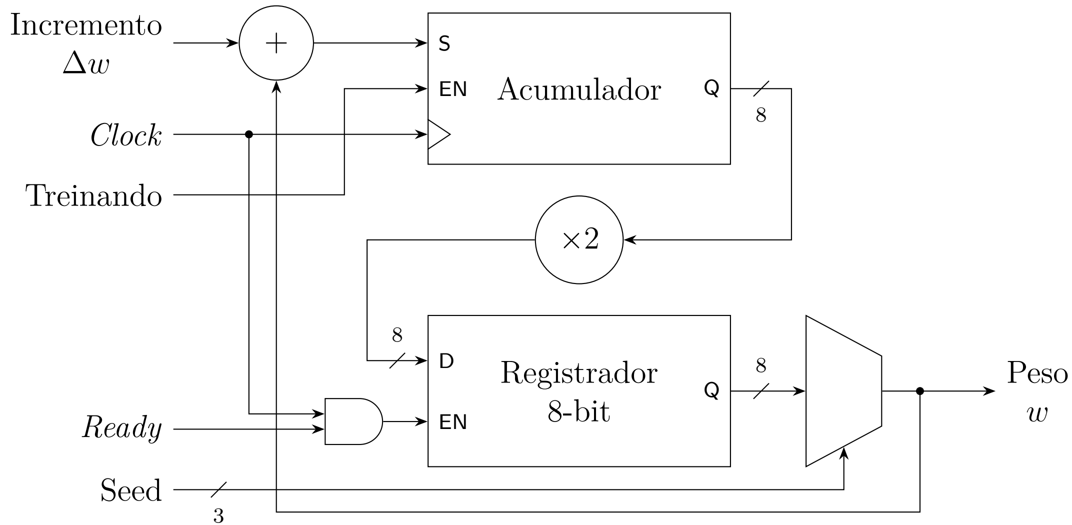


Figura 3.4: Estrutura interna do módulo "Pesos" do Banco, responsável pelo armazenamento e processamento de um peso sináptico individual.

De maneira detalhada, o *Seeder* é implementado utilizando Registrador de Deslocamento com realimentação Linear (do inglês *Linear-Feedback Shift Register*) [29] de 32 *bits* e três geradores estocásticos por modulação, os quais são responsáveis pela elaboração de sequências elementares que formam a *seed* de um gerador de 8 *bits* por multiplexação.

3.2.2 Neurônios estocásticos

O neurônio é a estrutura computacional responsável pelo processamento não-linear de uma rede neural artificial determinado por uma função, chamada função de ativação, conforme visto anteriormente no Capítulo 2. Esse módulo também implementa a derivada da função de ativação para realizar o algoritmo de aprendizado, porém essa derivada não será abordada neste trabalho.

O circuito do neurônio estocástico foi implementado de modo que seja possível calcular, de forma aproximada, a função *tansig*, ou tangente hiperbólica de um determinado valor de modo puramente estocástico. O esquemático ilustrado pela Figura 3.6 mostra o diagrama dessa estrutura.

Uma versão aproximada da função *tansig*, proposta por *Brown et al.* [30], foi utilizada como base para a implementação desta funcionalidade, a qual baseia-se em máquina de estados finitos.

Tomando-se como base um contador saturado, de n estados, capaz de armazenar números no intervalo fechado $[0, n-1]$, no qual, considerando uma sequência estocástica de controle S de modo que, a cada ciclo de *clock* um *bit* 1 incrementa o contador e um *bit* 0 realiza o decremento, o sinal de saída é adicionado ao contador, assumindo valor 1 se,

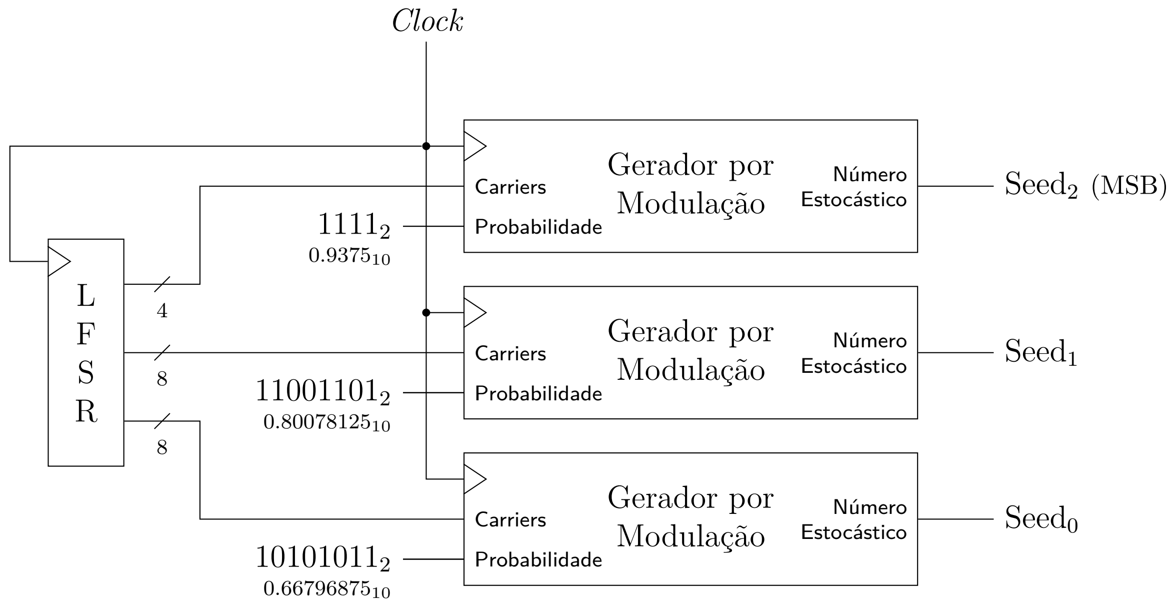


Figura 3.5: Circuito utilizado para conversão estocástica dos pesos sinápticos.

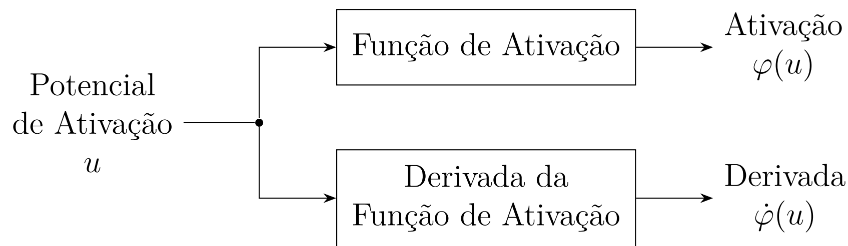


Figura 3.6: Visão geral de um neurônio estocástico.

e somente se, o contador encontrar-se em um estado maior ou igual a $\frac{n}{2}$. Ao interpretar tal saída como uma sequência estocástica, é possível representar tal valor como uma aproximação da tangente hiperbólica. A máquina de estados que modela esse contador é ilustrado pela Figura 3.7.

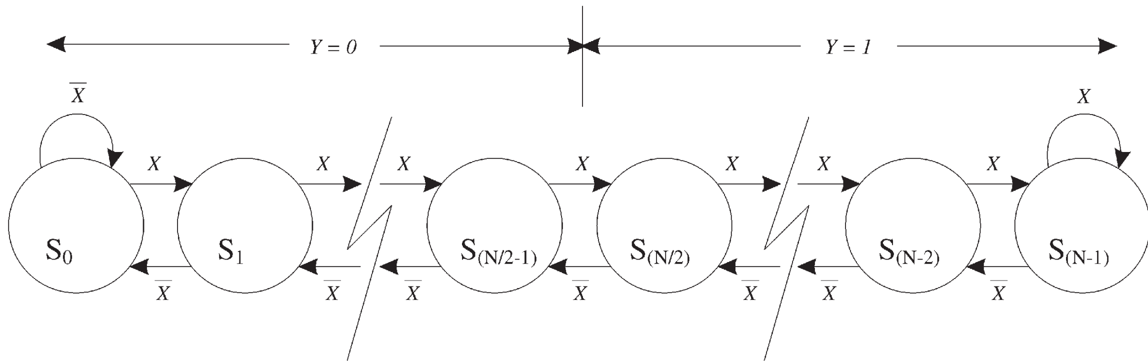


Figura 3.7: Diagrama da tangente hiperbólica estocástica (Fonte: [30]).

É possível perceber o comportamento de um contador saturado a partir da Figura 3.7, o qual opera dentro de uma faixa de valores limitado por um valor máximo e um valor mínimo. Por exemplo, utilizando um contador de 3 *bits*, seus valores máximo e mínimo, respectivamente, são 111 e 000. Assim, em uma tentativa de incremento caso o valor seja 111, a máquina de estados permanece no mesmo estado. O mesmo ocorre para o valor 000, no qual a máquina de estados mantém o estado na tentativa de decremento.

Essa implementação da tangente hiperbólica apresenta grande vantagem na implementação de uma estrutura como o neurônio estocástico utilizado nesse trabalho, uma vez que sua implementação resulta em um circuito compacto e altamente paralelizado. O circuito utilizado neste trabalho utiliza uma máquina de estados com $N = 6$ estados, ou seja, requer um contador de 3 *bits* e um circuito combinacional.

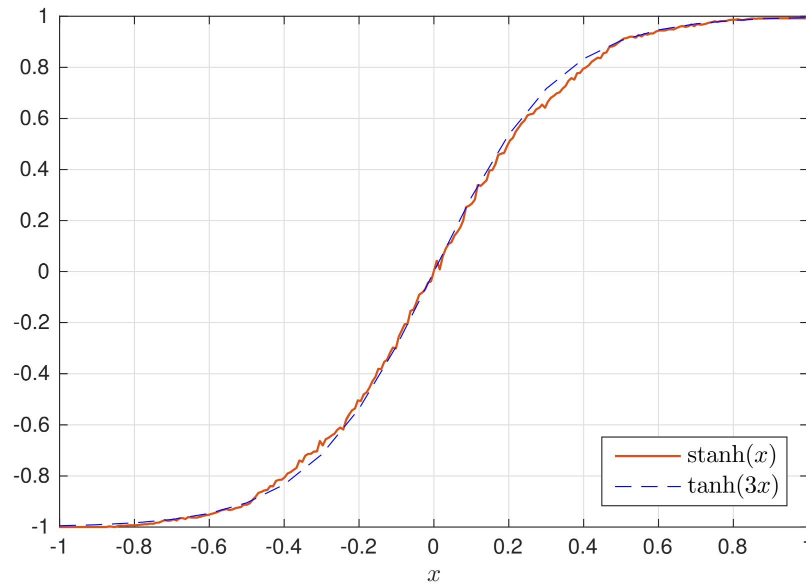


Figura 3.8: Tangente hiperbólica estocástica implementada.

A Figura 3.8 apresenta o comportamento da função tangente hiperbólica estocástica ($stanh(x)$, em vermelho) no FPGA *DE2-115* da *Altera*, a qual utiliza 3 elementos lógicos para tal estrutura, em comparação com a tangente hiperbólica real ($tanh(3x)$, em pontilhado azul).

3.2.3 Camadas Neurais

As estruturas de camadas são responsáveis pelo real processamento neural da rede, enquanto que as estruturas de sinapses encarregam-se do controle de fluxo e execução dos algoritmos de aprendizado e *feedforward*.

Conforme já visto anteriormente, uma rede neural recorrente de *Elman* é organizada de forma semelhante a uma Perceptron Multicamadas (do inglês *Multilayer Perceptron*) (MLP), ou seja, possui três tipos camada: camada de entrada; camada escondida; camada de saída. Além destas camadas, uma rede de *Elman* possui uma camada denominada *Camada de Contexto*, a qual armazena os valores de saída da camada escondida quando estes são propagados para a camada de saída.

A camada de entrada encarrega-se de fornecer à rede os sinais de entrada do sistema. Implementa-se esta camada apenas roteando-se as entradas às ativações do primeiro módulo sináptico, ou seja, essa camada é composta pelo conjunto de sinais de entrada, os valores da ativação da camada escondida da iteração anterior (sinais de realimentação da camada escondida) e o *viés*.

Os sinais da camada de entrada são propagados através das Sinapses para a camada escondida, que são sintetizadas a partir de uma quantidade pré-definida de neurônios, responsáveis pela ativação dos sinais roteados conforme a Figura 3.9. Os neurônios desta camada operam a função tangente hiperbólica para ativação das entradas.

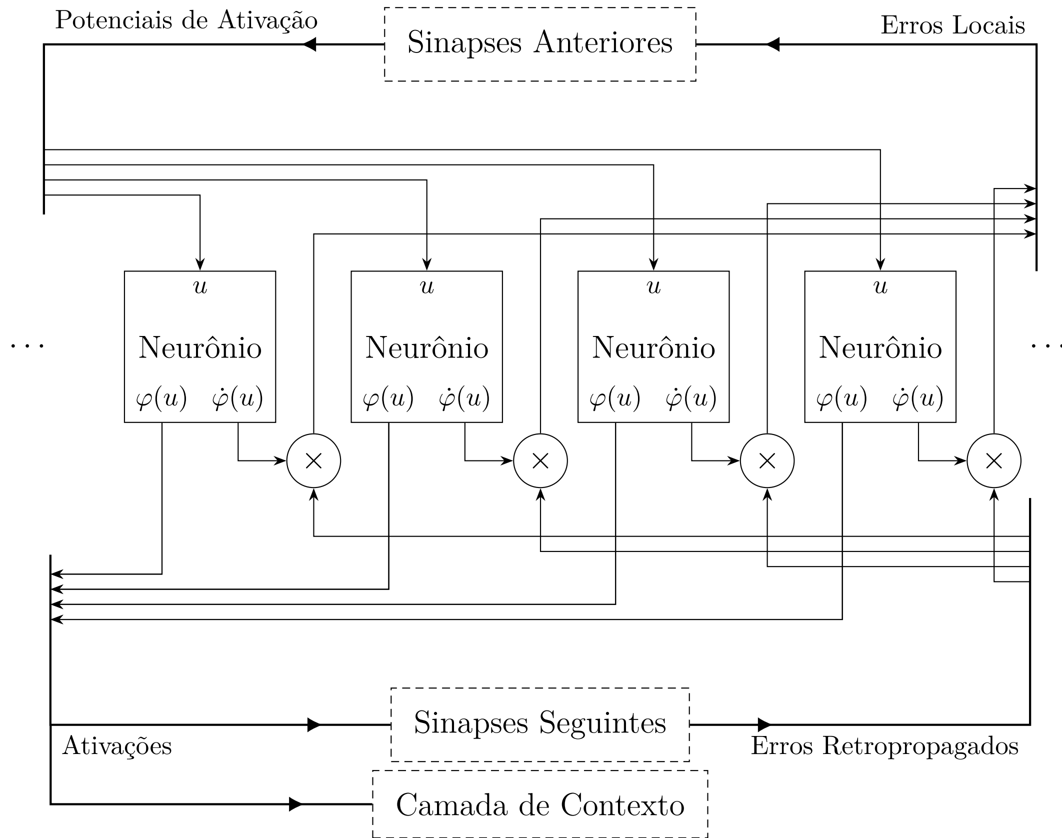


Figura 3.9: Diagrama que descreve uma camada oculta da rede.

O *feedforward* ocorre quando as "Sinapses Anteriores" fornecem à camada escondida os potenciais de ativação, então os neurônios ativam esses potenciais, que são sinais de entrada das "Sinapses Seguintes", de modo que seja continuado o *feedforward* até a camada de saída.

A implementação da camada de saída é similar à camada escondida, entretanto é nesta estrutura que ocorre o cálculo dos erros de saída para a execução da *retropropagação de erros*. A Figura 3.10 ilustra o funcionamento interno da camada de saída. Tal camada é implementada com neurônios estocásticos e sinapses anteriores. Não é necessário a adição de sinapses seguintes, uma vez que tal estrutura é a última estrutura da rede, sendo necessário apenas apresentar os valores de saída e calcular o erro para a *retropropagação*.

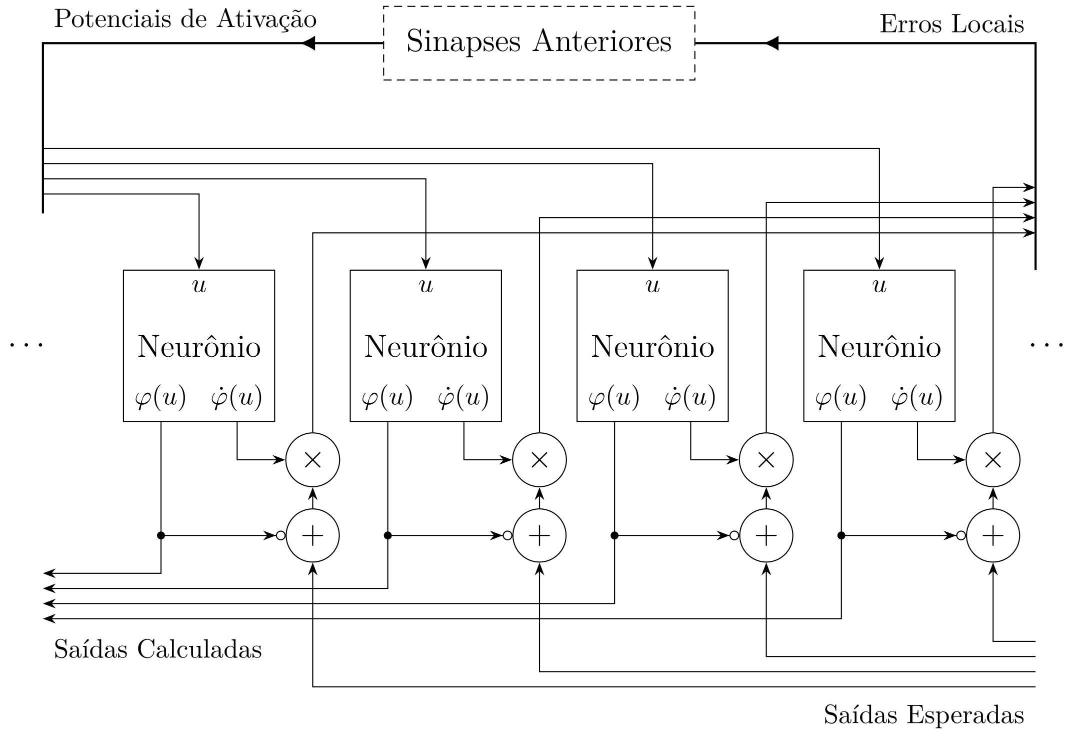


Figura 3.10: Implementação da camada de saída.

3.2.4 Sinapses

Conforme apresentado nas subseções anteriores, os neurônios estocásticos são organizados internamente na camada escondida e na camada de saída e são responsáveis pelo processamento e ativação dos potenciais de ativação das camadas anteriores. A camada escondida e de saída encapsulam conjuntos de neurônios e conectam-se às sinapses. Por fim, as sinapses são responsáveis pelo direcionamento de fluxo dos sinais, sendo eles o fluxo direto de execução da rede, ou *feedforward*, e fluxo inverso de correção dos pesos sinápticos, através da *retropropagação de erros*. A estrutura interna das sinapses pode ser visualizada através da Figura 3.11.

Pelo diagrama de blocos apresentado pela Figura 3.11 é possível identificar três grandes módulos internos dessa estrutura: "Propagação", "Adaptação" e "Retropropagação". As duas últimas macroestruturas internas do módulo Sinapses não serão abordadas por fugirem o escopo deste trabalho pois tratam, respectivamente, do cálculo dos incrementos dos pesos sinápticos e controle de fluxo dos sinais de erro do *backpropagation*.

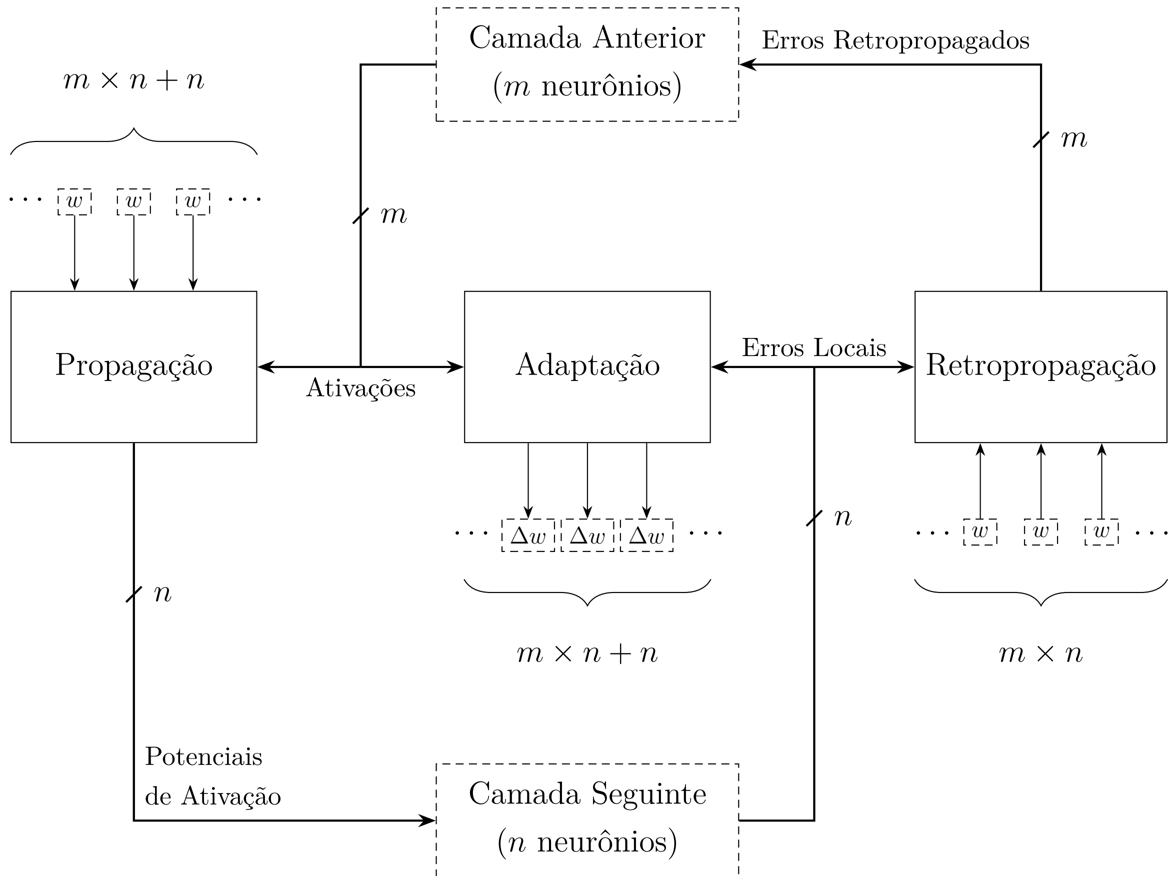


Figura 3.11: Blocos de circuitos responsáveis pela lógica sináptica.

Os sinais *ativações*, *potenciais*, *erros locais* e *erros retropropagados* realizam as conexões entre os grandes módulos que compõem o circuito sináptico. Os sinais de *ativação* são a saídas fornecidas pelos neurônios da camada anterior. *Potenciais de ativação* representam as entradas da camada seguinte, ou seja, os sinais de saída dos neurônios dessa camada multiplicados pelos pesos sinápticos das conexões com a próxima camada. Os *erros locais* e *erros retropropagados* são, respectivamente, os erros locais dos neurônios da camada seguinte e os erros fornecidos pelos neurônios da camada anterior, no qual ocorre a soma ponderado dos sinais.

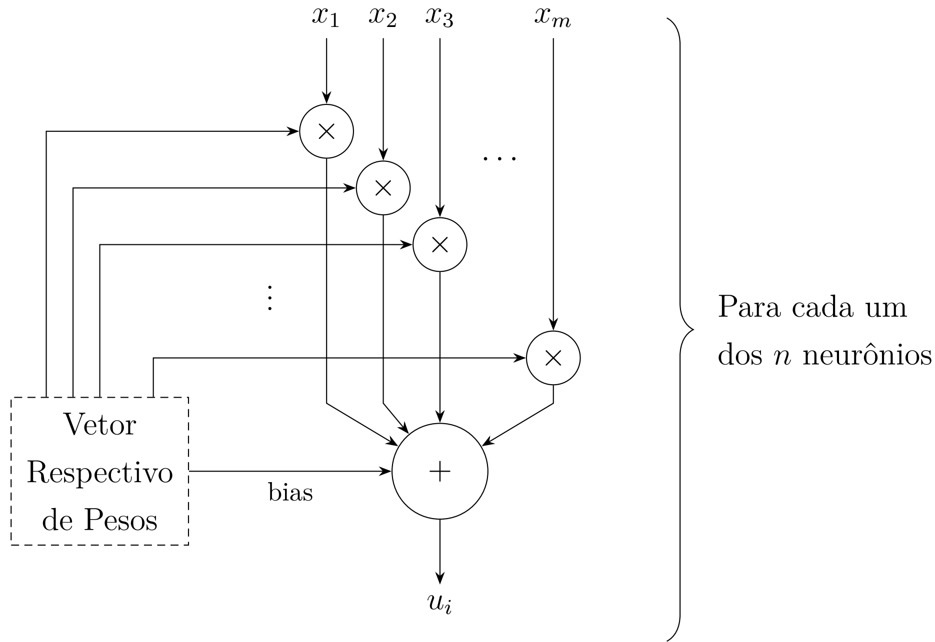


Figura 3.12: Circuito sináptico.

É interessante notar a similaridade entre os circuitos dos algoritmos de *feedforward*, ilustrado pela Figura 3.12, e *backpropagation*, que pode ser visto na Figura 3.13, uma vez que ambos realizam operações de produto escalar de um conjunto de sinais com um conjunto de pesos, de modo que sejam gerados os sinais de *potenciais de ativação* e *erros retropropagados*. Tais algoritmos diferem-se na presença do *viés*, presente na propagação de sinais de entrada, o qual torna-se necessário sinais de pesos adicionais.

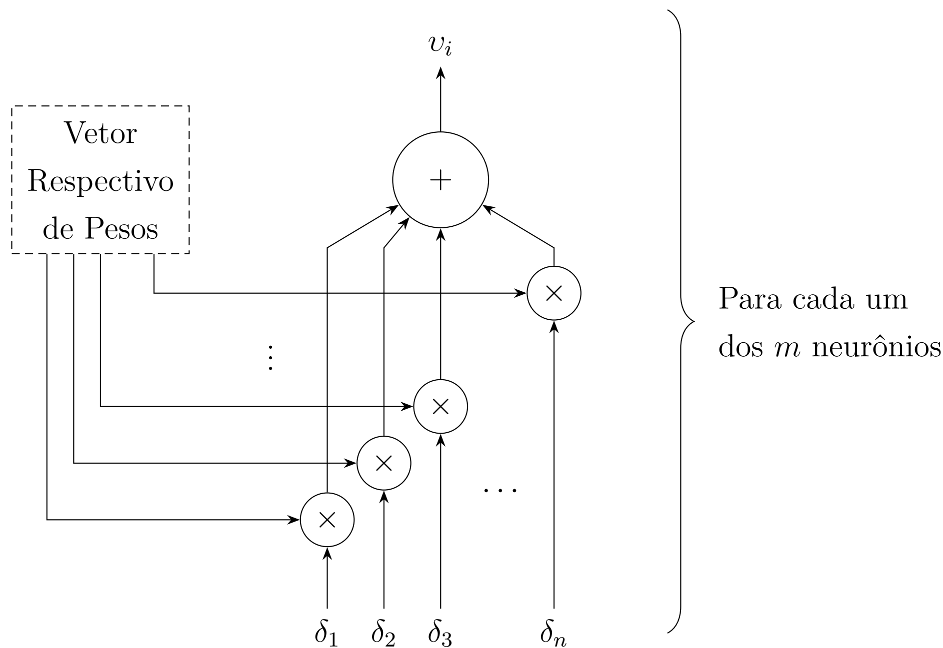


Figura 3.13: Circuito sináptico reverso, usado na retropropagação de erros.

Nos circuitos apresentados para o *feedforward* e *backpropagation*, os quais são esquemas detalhados dos módulos "Propagação" e "Retropropagação", é possível observar diversos multiplicadores e somadores estocásticos, os quais são implementados, respectivamente, por portas XNOR e somadores de Markov.

3.2.5 Controle da Rede Neural

Através de chaves seletoras na placa de desenvolvimento, é possível determinar o modo de operação da rede neural recorrente estocástica. Foram estabelecidos códigos de operação para a rede neural, de modo que seja possível realizar as seguintes operações: processamento da entrada a fim de gerar uma saída; uma iteração do algoritmo de aprendizado; estabelecer pesos sinápticos aleatórios, reiniciando a rede; transferir pesos da memória RAM para o Banco de Pesos; transferir os pesos do Banco de Pesos para a memória RAM; executar um ciclo de avaliação da rede neural recorrente, copiando os valores da camada escondida para a camada de contexto; zerar os valores da camada de contexto.

```
// -- Operations.
typedef enum logic [2:0]
{
   NORMAL,      // Normal forward processing. 000
   LEARN,       // Executes one iteration of the Backpropagation algorithm 001
   RESET,       // Resets everything, initialize weights to random values 010
   RAM2BANK,    // Transfers weights from the RAM to the bank 011
   BANK2RAM,    // Transfers weights form the bank to the RAM 100
   CYCLE,       // Executes a cycle, updating context layer delays 101
   CLEAR        // Clears context layers 110

} Code;

endpackage
```

Figura 3.14: Códigos de operação da rede neural no *FPGA*

A Figura 3.14 mostra o nome das operações e comentários resumindo a operação e o respectivo código.

3.3 Treinamento da Rede Neural Recorrente Estocástica

A proposta deste trabalho é implementar redes neurais recorrentes estocásticas que aprendam problemas simples de classificação e séries temporais, de modo que seja possível utilizar os pesos encontrados no treinamento para utilização das redes através de dispositivos reconfiguráveis FPGA.

Para o treinamento, foram utilizados funções escritas em *software* através do Matlab, conforme apresentado anteriormente neste capítulo.

Após o treinamento e obtenção de redes satisfatórias, foram gerados arquivos para carregamento em memória do FPGA com os pesos dessas redes, os quais foram posteriormente carregados da memória para os registradores do "Banco de Pesos". Desse modo, é possível reproduzir em *hardware* puramente estocástico redes treinadas por *software* no qual foi respeitado o comportamento estocástico esperado das operações realizadas.

Este capítulo apresentou a metodologia e processo utilizado na implementação da solução proposta, de modo que algoritmos e trechos de códigos são explicados e traduzem os conceitos do Capítulo 2 para linguagem de programação. O Capítulo seguinte abordará experimentos e seus resultados com a finalidade de verificar o funcionamento e eficácia da solução.

Capítulo 4

Resultados Obtidos

Neste capítulo serão apresentados os resultados de treinamentos de diversas redes neurais recorrentes estocásticas, a fim de avaliar taxas de acerto, quantidade de neurônios necessários para aprendizado dos problemas apresentados e recursos físicos exigidos para a síntese do circuito em *FPGA*. Também será comparada, em termos de requisitos físicos, a implementação de redes neurais recorrentes com computação estocástica, ponto fixo e ponto flutuante.

Para o desenvolvimento deste trabalho, foi adotado um desenvolvimento modular, de modo que primeiramente fosse implementados os algoritmos de *feedforward* e *back-propagation*. Dessa forma, tornou-se necessária a validação do que foi implementado no Matlab, a fim de que o comportamento estocástico do sistema pudesse ser visualizado graficamente em comparação com o comportamento evidenciado na implementação dos módulos estocásticos do circuito escrito em *SystemVerilog*.

Comprovando-se a corretude dos *scripts* Matlab, iniciou-se o processo de treinamento com problemas simples e de rápida convergência. Para tal, foi gerado um conjunto de treinamento com entradas e saídas para realização de treinamento supervisionado para que a rede neural recorrente estocástica aprendesse a resolver o problema XOR. Esse problema, clássico da literatura de redes neurais [24], foi adaptado para torná-lo um problema temporal, conforme é proposto por Elman [8].

Após a comprovação da corretude dos algoritmos implementados em *software*, foi escolhido um problema mais sofisticado, o qual consiste numa sequência de entradas no tempo para reconhecimento de fonema. Tal problema pode ser encontrado no *Matlab*. Maiores detalhes quanto a esse conjunto de dados pode ser encontrado nas próximas subseções.

De posse de redes neurais cuja convergência foi alcançada para ambos os problemas, o próximo e último passo foi gerar arquivos Arquivo de Inicialização de Memória (do inglês *Memory Initialization File*) (MIF) com os pesos sinápticos das redes e carregá-los

na memória RAM do circuito. Para tal processo, foi preciso sintetizar o circuito conforme a configuração arquitetural das redes a serem testadas no *FPGA*, ou seja, sintetizar a rede com a quantidade de neurônios na camada escondida, bem como na camada de contexto para recorrência. As redes, então, foram simuladas no próprio *hardware* e os dados de saída foram extraídos para visualização da saída esperado mediante apresentação de um conjunto de sinais de entrada.

A partir do processo descrito acima e dos resultados obtidos, foi possível analisar os resultados da rede neural recorrente estocástica e compará-los com outros resultados provenientes de treinamento da rede neural recorrente de *Elman* do Matlab, a qual foi configurada com os mesmos algoritmos utilizados pela rede neural deste trabalho, entretanto operando no domínio real. Todos os passos descritos acima são apresentados em mais detalhes nas subseções abaixo.

4.1 Estudos de Caso

Esta subseção tem como objetivo apresentar os problemas XOR Temporal e Reconhecimento de Fonema. Ambos os problemas foram utilizados para treinamento de redes neurais recorrentes de Elman tradicionais e estocásticas. A partir do treinamento e seleção das redes com melhores taxas de acerto, foi possível observar a saída destas redes para os cenários onde treinaram-se redes neurais tradicionais e redes neurais estocásticas com a finalidade de compará-las e verificar a convergência das redes estocásticas, além da precisão da saída. Por fim, foram sintetizadas redes neurais estocásticas para o *FPGA*, de modo que fossem analisadas questões como quantidade de elementos lógicos necessários para síntese, bem como quantidade de registradores e memória.

4.1.1 Problema do XOR Temporal

A função "OU-EXCLUSIVO" (*exclusive-or*, ou apenas XOR) torna-se interessante de ser utilizada como validação para funcionamento da implementação de redes neurais, uma vez que trata-se de um problema não linearmente separável, isto é, tal problema não pode ser aprendido por uma simples rede neural de duas camadas (camada de entrada e camada de saída) [8]. Essa função é apresentada à rede como uma entrada de dois *bits* e um *bit* de saída, sendo o *bit* de saída equivalente a operação "OU-EXCLUSIVO" dos *bits* de entrada.

A Tabela 4.1 apresenta a tabela verdade da função XOR.

É possível adaptar esse problema para o domínio do tempo, de forma que sejam apresentados os sinais de entrada x_t e x_{t+1} e espera-se na saída, no tempo $t+1$, o resultado da operação XOR entre x_t e x_{t+1}, em seguida apresenta-se x_{t+2} e a saída da rede é o resultado da operação XOR entre x_{t+1} e x_{t+2} e assim por diante.

Tabela 4.1: Tabela Verdade da função XOR.

Entradas	Saída
00	0
01	1
10	1
11	0

Por exemplo, uma possível sequência de entrada seria $x = \{1, 0, 1, 0, 0, 1, 1, 0\}$ e sua saída correspondente $y = \{0, 1, 1, 1, 0, 1, 0, 1\}$. É importante perceber que o primeiro *bit* de saída após a apresentação da entrada x_0 pode ser 0 ou 1, pois é necessário mais um instante de tempo para dar início à operação.

Treinamento em *Software*

Por se tratar de um problema simples, foram treinadas 5 redes neurais recorrentes estocásticas para cada variação de quantidade de neurônio na camada escondida, sendo essas quantidades iguais a 2, 3 e 4.

As melhores redes para cada quantidade de neurônio apresentam os seguintes resultados de acordo com a seguintes entradas nos tempos t_0 e t_1.

Tabela 4.2: Resposta da rede com 2, 3 e 4 neurônios na camada escondida.

Entradas	Saída com 2 neurônios	Saída com 3 neurônios	Saída com 4 neurônios
$x_{t_0} = -1$ $x_{t_1} = -1$	$y_{t_1} = -0.5119$	$y_{t_1} = -0.4480$	$y_{t_1} = -0.9111$
$x_{t_0} = -1$ $x_{t_1} = 1$	$y_{t_1} = 0.9335$	$y_{t_1} = 0.8865$	$y_{t_1} = 0.4180$
$x_{t_0} = 1$ $x_{t_1} = -1$	$y_{t_1} = 0.9295$	$y_{t_1} = 0.7410$	$y_{t_1} = 0.7956$
$x_{t_0} = 1$ $x_{t_1} = 1$	$y_{t_1} = -0.8503$	$y_{t_1} = -0.8258$	$y_{t_1} = -0.7443$

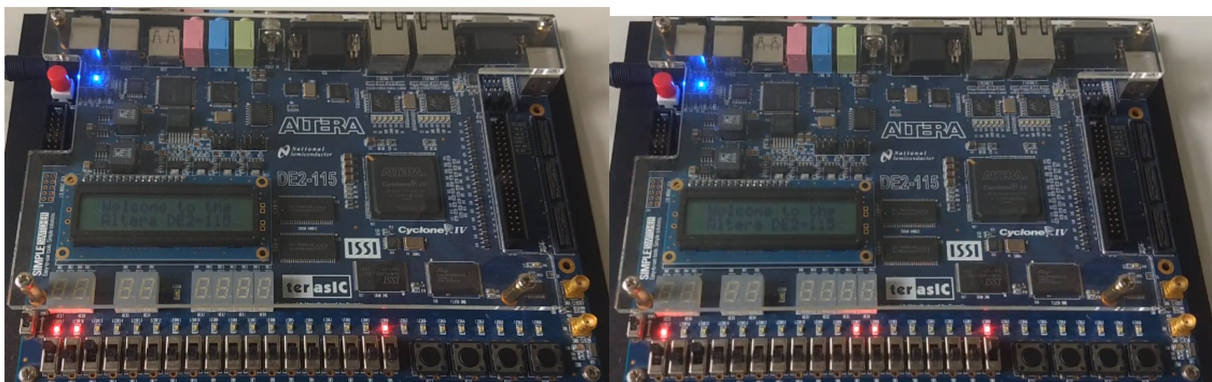
Pelas três redes apresentadas na Tabela 4.2, é possível perceber que todas as redes, após o treinamento, foram capazes de realizar a operação XOR no tempo, onde recebe-se um primeiro *bit* como entrada e, no *bit* seguinte, realiza-se a operação XOR destes termos.

É importante ressaltar que a saída y_{t_0} não é confiável, uma vez que tal saída corresponde à resposta da rede à entrada x_{t_0}, ou seja, não há valores anteriores para realização da operação XOR, sendo possível observar o comportamento desta operação a partir do segundo *bit* de entrada, no tempo t_1.

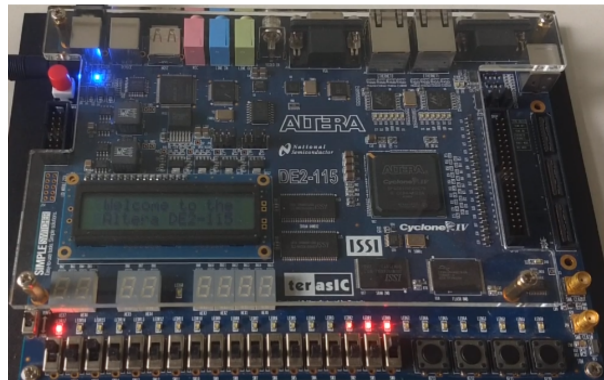
Resposta da Rede no *FPGA*

A rede com 2 neurônios da Tabela 4.2 foi escolhida para este procedimento, o qual envolve a geração de um arquivo MIF com os pesos obtidos no treinamento dessa rede, sendo possível a importação dos pesos para a RAM do *FPGA*.

Para interface com o circuito com o kit de desenvolvimento DE2-115, foram utilizados as chaves seletoras (*switches*) 17, 16 e 15 para os códigos de operação, ilustrados pela Figura 3.14, a chave seletora 0 para entrada da rede neural, o botão (*key*) 0 para executar a operação selecionada pelo código de operação escolhido nas chaves 17, 16 e 15. Os *leds* vermelhos apresentam a distribuição de probabilidade da saída da rede, a qual representa a quantidade -1 caso os *leds* vermelhos concentrem-se à esquerda do barramento de *leds*. A quantidade 0 apresenta os *leds* do centro acesos e, à direita, representa a quantidade 1.



(a) Saída -1 (b) Saída 0



(c) Saída 1

Figura 4.1: Representações dos valores de saída -1, 0 e 1 no FPGA

As Figura 4.1a, Figura 4.1b e Figura 4.1c apresentam os *leds* acesos para as respectivas saídas. É importante notar que a saída não está exatamente no valor 0, mas sim um valor aproximado devido a natureza do comportamento estocástico do sistema.

As chaves seletoras encontram-se abaixo dos *leds* vermelhos à esquerda dos botões. Utilizando-se das chaves 17, 16 e 15, as três últimas da esquerda para a direita, é possível

escolher o código de operação do sistema. O botão 0 é o primeiro da direita para a esquerda e é responsável por acionar o sistema de acordo com a operação escolhida.

Os códigos de operação da rede neural utilizados nesse procedimento foram: 011 para transferência dos pesos da *RAM* para o banco de pesos da rede, 101 para *feedforward* da rede neural recorrente e 110 para limpar a camada de contexto.

Após a importação dos pesos para a *RAM*, estes são copiados para o banco de pesos pela operação 011. Então, a camada de contexto é zerada para primeira iteração da rede através do comando 110. As iterações *feedforward* ocorrem com a operação 101.

Um vídeo demonstrativo está disponível em: `https://youtu.be/JHwlMopM6W8`

Requisitos Físicos

Para sintetizar uma rede neural recorrente estocástica em FPGA, utilizando 2 neurônios na camada escondida e de contexto, são necessários os requisitos físicos apresentados na Tabela 4.3.

Tabela 4.3: Requisitos Físicos para Sintetizar Rede com 2 neurônios.

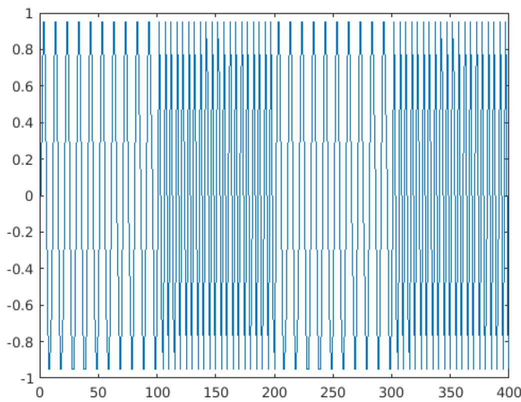
Elementos Lógicos	1,927 / 114,480 (2%)
Registradores	1079
Bits de Memória	73,728 / 3,981,312 (2%)
Multiplicador de 9-*bits*	0 / 532 (0%)

Dessa forma, a síntese da rede neural recorrente estocástica, para a resolução do problema XOR temporal, necessita de 1,927 elementos lógicos, ou seja, apenas 2% do total de elementos lógicos disponíveis no *FPGA*. A quantidade de pinos e *bits* de memória, como será visto no problema seguinte, não varia com aumento da quantidade de neurônios.

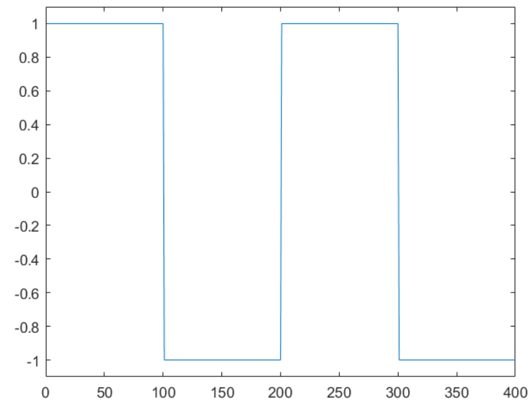
4.1.2 Reconhecimento de Fonema

Outra aplicação de redes neurais temporais envolve a detecção de fonemas. Tal problema foi investigado por *Waibel et al.* [31] no estudo de Redes Neurais com Atraso Temporal (do inglês *Time-Delay Neural Networks*) (TDNN), uma arquitetura de redes neurais que processa sinais de entrada com atraso no tempo, a qual apresentou-se bastante especializada para o reconhecimento de fonemas.

O problema resolvido nesta seção envolve um processo similar ao reconhecimento de fonema, no qual a rede neural busca reconhecer o conteúdo da frequência de um dado sinal de entrada. A Figura 4.2a ilustra o sinal de entrada do conjunto de treino.



(a) Sinal de entrada



(b) Sinal de saída

Figura 4.2: Sinais do problema *phoneme*

A saída esperada da rede para apresentação do sinal de entrada pode ser vista na figura Figura 4.2b. É importante notar que a saída esperada da rede deve ser 1 quando baixas frequências são identificadas no sinal de entrada. Para o reconhecimento de altas frequências, espera-se que a rede tenha -1 como valor de saída.

Esse problema pode ser gerado para utilização no Matlab através do Código-fonte 4.1.

Código 4.1: Script para geração do problema *phoneme*

```
1 time = 0:99;
2 y1 = sin(2*pi*time/10);
3 y2 = sin(2*pi*time/5);
4 y = [y1 y2 y1 y2];
5 t1 = ones(1,100);
6 t2 = -ones(1,100);
7 t = [t1 t2 t1 t2];
```

onde as variáveis y e t são, respectivamente, o sinal de entrada e a saída esperada.

Treinamento via *Software*

Utilizando uma rede neural recorrente de *Elman* nativa do Matlab, com as configurações apresentadas na Figura 4.3. Na imagem, foram utilizados 5 neurônios na camada escondida, entretanto acrescentou-se 5 neurônios a cada bateria de treinamento até atingir 30 neurônios. Dessa forma, foram treinadas redes com 5, 10, 15, 20, 25 e 30 neurônios por 5 mil épocas.

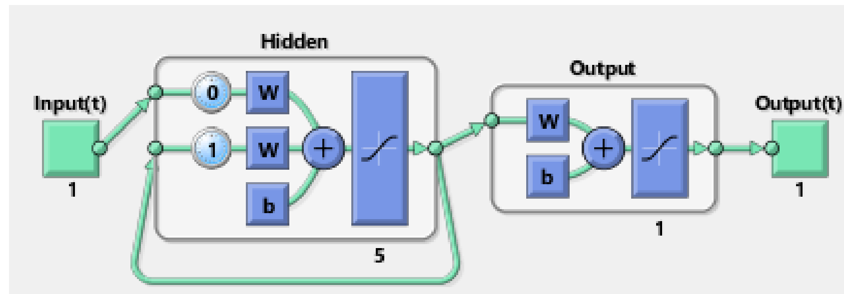


Figura 4.3: Estrutura da rede de Elman no Matlab com 5 neurônios na camada escondida.

A Figura 4.4 apresenta o gráfico de média de porcentagem de acerto das redes treinadas utilizando a *toolbox* do Matlab.

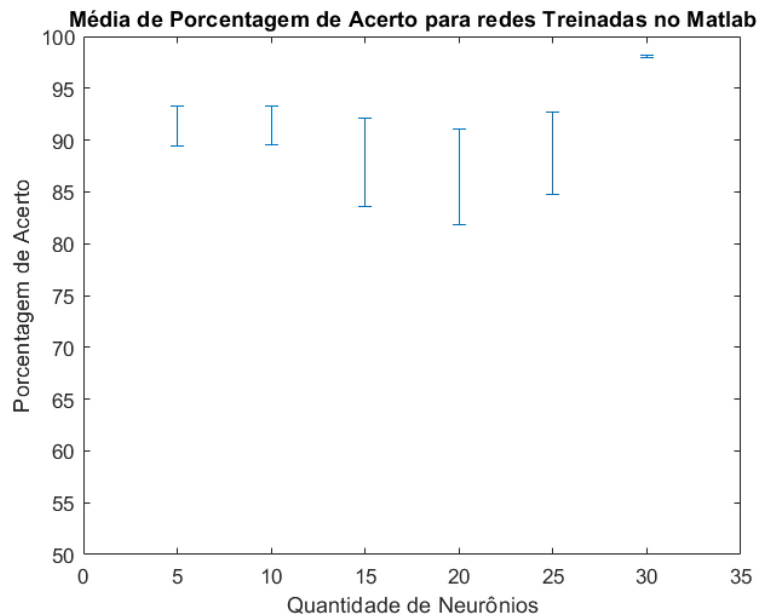


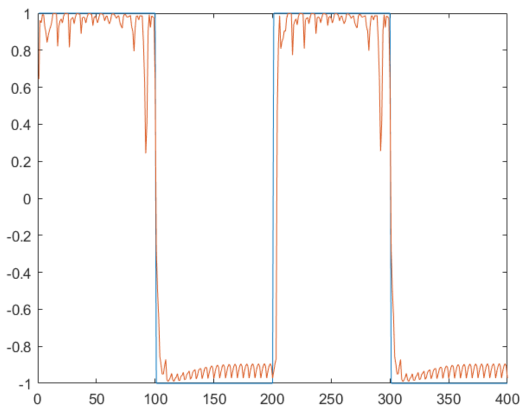
Figura 4.4: Média de porcentagem de acerto.

A Tabela 4.4 apresenta a taxa de acerto para as melhores redes obtidas para cada número de neurônios na camada escondida treinadas no *software* Matlab.

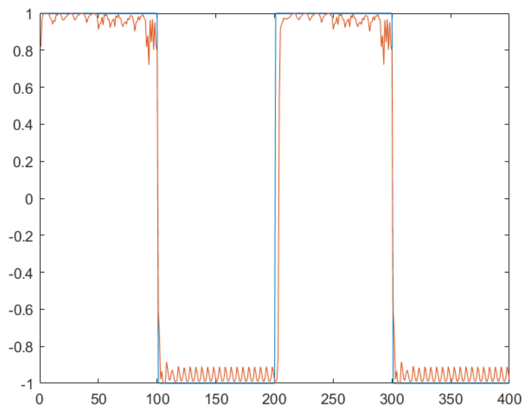
As melhores redes com 15, 20 e 25 neurônios apresentaram a mesma taxa de acerto, desse modo está apresentado na Figura 4.5 as comparações da saída dessas 3 redes com a resposta esperada de acordo com o sinal de entrada.

Tabela 4.4: Melhores Redes treinadas pelo Matlab.

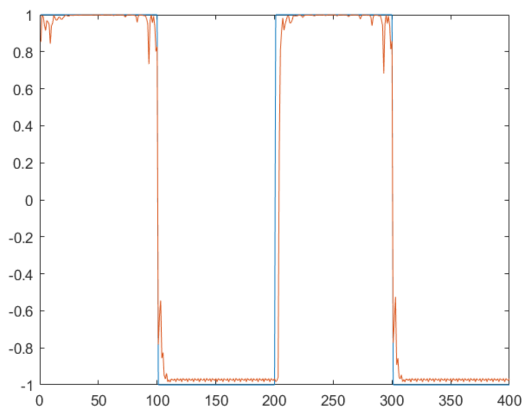
Qtd. Neurônios	Taxa de Acerto
5	98.50%
10	98.50%
15	99.25%
20	99.25%
25	99.25%
30	98.75%



(a) 15 neurônios.



(b) 20 neurônios.



(c) 25 neurônios.

Figura 4.5: Respostas das redes

Nota-se que embora todas acertem a classificação do sinal de entrada, a rede com 25 neurônios apresenta a saída mais próxima ao sinal desejado, conforme esperado.

Todas as redes utilizaram como parâmetros o fator de aprendizado $\mu = 0.01$ e o *Momentum* $\alpha = 0.9$. O algoritmo utilizado na configuração da rede de Elman foi o

Gradiente Descendente com Momentum (`traingdm`).

Para a rede neural recorrente estocástica, com 5, 10, 15, 20, 25 e 30 neurônios na camada escondida, bem como na camada de contexto. Foram treinadas um total de 30 redes para cada quantidade de neurônio, totalizando 180 redes.

Além de variar a quantidade de neurônios, variou-se também a quantidade de épocas totais. Cada conjunto de 180 redes foi treinado por 10 mil épocas, 20 mil épocas, 40 mil épocas e 50 mil épocas, resultando em 720 redes no total.

A Figura 4.6 apresenta a média de porcentagem de acerto das 30 redes neurais treinadas para cada quantidade de neurônio na camada escondida após serem treinadas por 10 mil épocas.

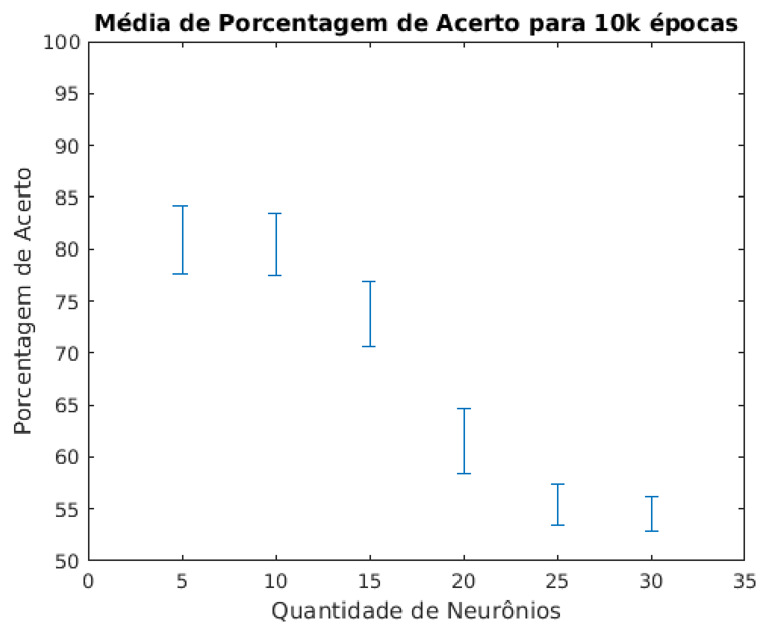


Figura 4.6: Média de porcentagem de acerto para 10 mil épocas.

A Tabela 4.5 apresenta as relações das redes que obtiveram maior percentual de acerto para cada quantidade de neurônio treinada.

Tabela 4.5: Melhores Redes para 10 mil épocas.

Qtd. Neurônios	Taxa de Acerto
5	97.00%
10	96.00%
15	96.50%
20	96.00%
25	87.25%
30	84.50%

A rede com maior porcentagem de acerto foi de 97%, com 5 neurônios na camada escondida e de contexto. A Figura 4.7 mostra a resposta da rede ao conjunto de sinais de entrada, em comparação com a saída esperada, apresentada à rede durante o período de treinamento.

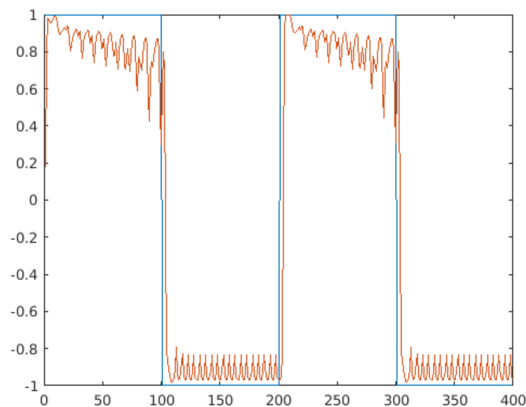


Figura 4.7: Resposta da rede com 5 neurônios treinada com 10 mil épocas.

Pelos dados apresentados, é possível observar que redes com quantidades de neurônio menores que 30 e 25 têm desempenho melhor para a solução deste problema.

Aumentando-se a quantidade de épocas para 20 mil, foi possível obter resultados melhores em comparação às redes treinadas por apenas 10 mil épocas. A Figura 4.8 apresenta a média de porcentagem de acerto das 30 redes neurais treinadas para cada quantidade de neurônio na camada escondida.

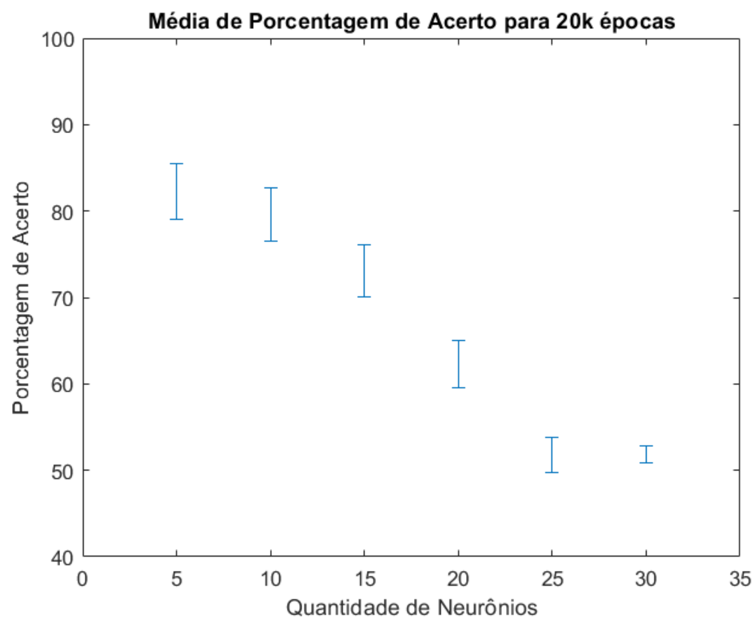


Figura 4.8: Média de porcentagem de acerto para 20 mil épocas.

A Tabela 4.6 apresenta as relações das redes que obtiveram maior percentual de acerto para cada quantidade de neurônio treinada.

Tabela 4.6: Melhores Redes para 20 mil épocas.

Qtd. Neurônios	Taxa de Acerto
5	96.50%
10	97.00%
15	98.00%
20	90.50%
25	94.50%
30	73.25%

Com 15 neurônios na camada escondida foi possível atingir uma porcentagem de acerto de 98%. A resposta da rede aos sinais de entrada é apresentada na Figura 4.9.

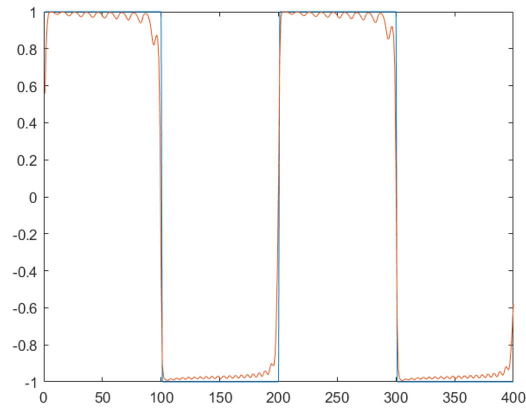


Figura 4.9: Resposta da rede com 15 neurônios treinada com 20 mil épocas.

O treinamento por 40 mil épocas apresenta resultados semelhantes ou inferiores ao treinamento com 20 mil e 10 mil épocas. Aumentar a quantidade de épocas não melhorou a convergência da redes.

A Figura 4.10 apresenta a média de porcentagem de acerto das 30 redes neurais treinadas para cada quantidade de neurônio na camada escondida.

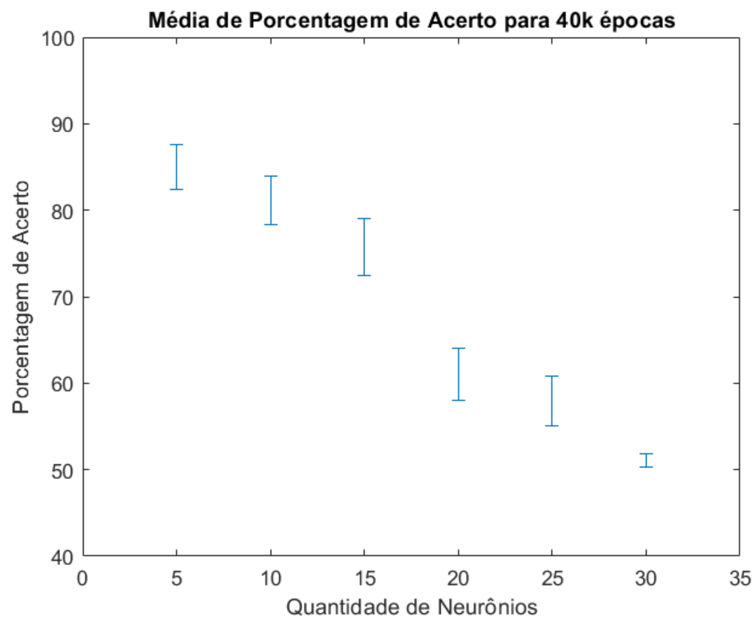


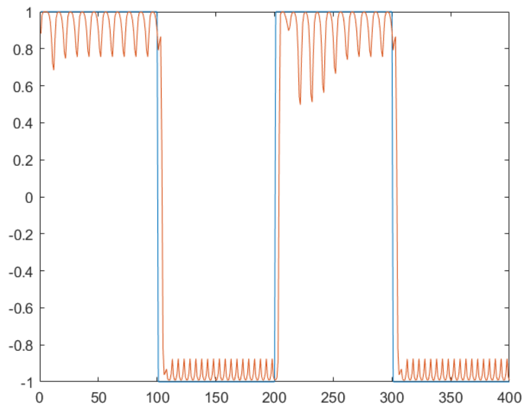
Figura 4.10: Média de porcentagem de acerto para 40 mil épocas.

A Tabela 4.7 apresenta as relações das redes que obtiveram maior percentual de acerto para cada quantidade de neurônio treinada.

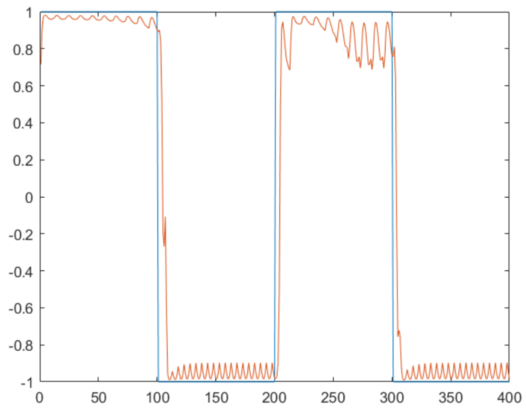
Tabela 4.7: Melhores Redes para 40 mil épocas.

Qtd. Neurônios	Taxa de Acerto
5	96.50%
10	96.50%
15	96.50%
20	96.00%
25	90.50%
30	74.50%

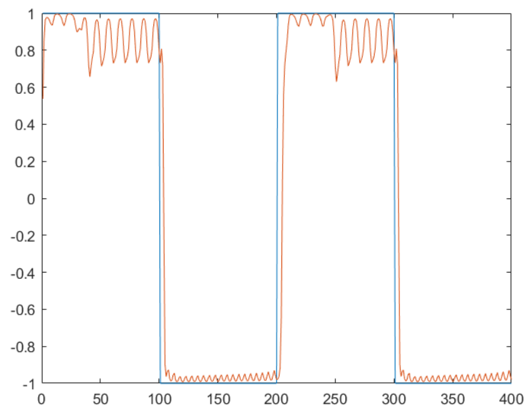
É possível perceber que três redes atingiram 96.50% de taxa de acerto. Respectivamente, as redes com 5, 10 e 15 neurônios alcançaram a maior porcentagem de acerto deste grupo. A Figura 4.11a mostra a resposta da rede com 5 neurônios ao conjunto de sinais de entrada, a Figura 4.11b apresenta a saída da rede com 10 neurônios e a Figura 4.11c a saída da rede com 15 neurônios na camada escondida.



(a) 5 neurônios.



(b) 10 neurônios.



(c) 15 neurônios.

Figura 4.11: Respostas das redes treinadas com 40 mil épocas.

Por último, o treinamento com 50 mil iterações, o qual mostrou resultados igualmente semelhantes aos treinamentos anteriores. Dessa forma, é possível concluir que aumentar a quantidade de iterações não melhorou a convergência das redes para este problema, sendo possível uma rede aprender a classificar sinais de alta e baixa frequências com apenas 10 mil ou 20 mil épocas.

Seria possível melhorar a convergência das redes variando-se levemente os parâmetros de *momentum* e fator de aprendizado. A Figura 4.12 apresenta a média de porcentagem de acerto das 30 redes neurais treinadas para cada quantidade de neurônio na camada escondida.

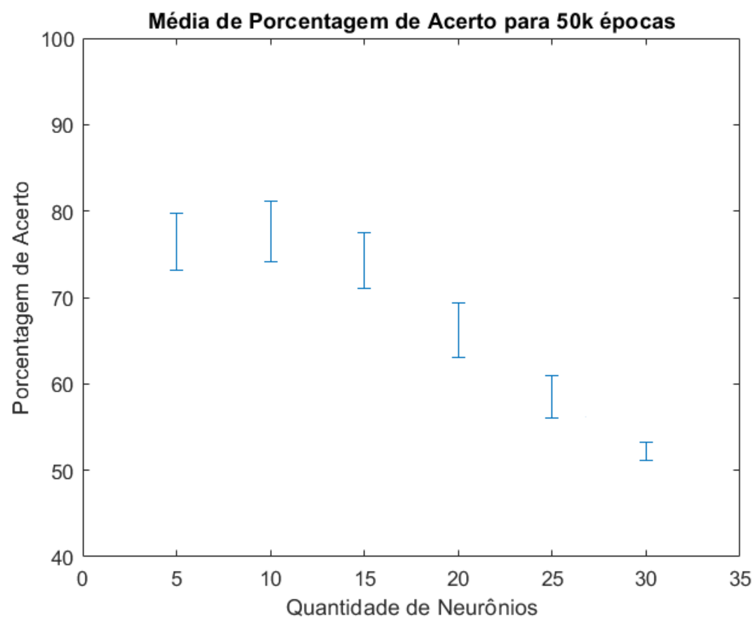


Figura 4.12: Média de porcentagem de acerto para 50 mil épocas.

A Tabela 4.8 apresenta as relações das redes que obtiveram maior percentual de acerto para cada quantidade de neurônio treinada.

Tabela 4.8: Melhores Redes para 50 mil épocas.

Qtd. Neurônios	Taxa de Acerto
5	97.00%
10	97.50%
15	96.00%
20	96.50%
25	95.50%
30	73.00%

A rede 10 neurônios apresentou maior porcentagem de acerto, 97.50%, quase alcançando o valor de 98.00% da rede com 15 neurônios do treinamento com 20 mil épocas. A Figura 4.13 mostra a resposta da melhor rede de treinada com 50 mil épocas.

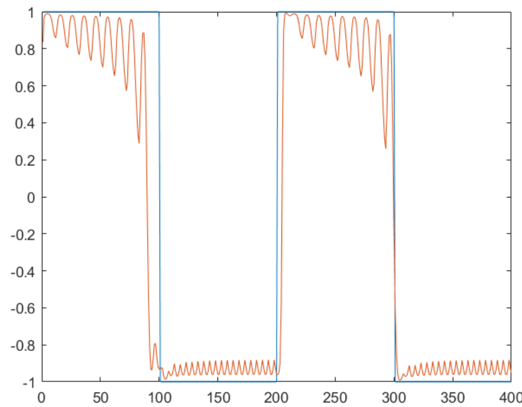


Figura 4.13: Resposta rede com 10 neurônios treinada com 50 mil épocas.

No entanto, é possível perceber que a resposta da melhor rede de 20 mil épocas, Figura 4.9, apresenta uma resposta mais próxima do sinal de saída esperado.

Requisitos Físicos

Para sintetizar uma rede neural recorrente estocástica em FPGA, utilizando 15 neurônios na camada escondida e de contexto, seria necessário os requisitos físicos apresentados na Tabela 4.9.

Tabela 4.9: Requisitos Físicos para Sintetizar Rede com 15 neurônios.

Elementos Lógicos	19,564 / 114,480 (17%)
Registradores	8221
Bits de Memória	73,728 / 3,981,312 (2%)
Multiplicador de 9-*bits*	0 / 532 (0%)

Para esta rede, apenas 17% de elementos lógicos disponíveis seriam utilizados.

4.2 Comparação com a Literatura

Outros autores realizaram trabalhos na área de redes neurais recorrentes implementadas em *hardware*, utilizando convenções binárias alternativas à computação estocástica, como ponto fixo e ponto flutuante.

Uma implementação de rede neural recorrente em FPGA, utilizando aritmética de ponto flutuante foi feita por *Silva Neto* [2], o qual também utilizou o problema do XOR temporal como validação do funcionamento da rede neural implementada.

Para sintetizar uma rede neural recorrente operando com representação em ponto flutuante, com 4 neurônios na camada escondida, foi necessário utilizar os recursos físicos presentes na Tabela 4.10

Tabela 4.10: Requisitos Físicos de síntese da rede neural recorrente em ponto flutuante para problema XOR temporal [2]

		Elementos Lógicos	Registradores	Bits de Memória	Multiplicadores de 9 bits
Aproximação polinomial	Aproximação de ordem 1	28.627(25%)	14.376	1.548(<1%)	196(37%)
	Aproximação de ordem 2	35.359(31%)	17.648	1.940(<1%)	224(42%)
	Aproximação de ordem 3	39.346(34%)	19.643	2.200(<1%)	252(47%)
Aproximação por Intervalos fixos	Aproximação 256 amostras	30.280(26%)	14.142	1.448(<1%)	168(32%)
Implementação exata	Implementação exata	31.316(27%)	15.462	21.152(<1%)	356(67%)
Número total de elementos no kit de desenvolvimento DE2-115	Total	/114.480		/3.981. 312	/532

Para resolver o mesmo problema do XOR temporal, utilizando computação estocástica com uma rede neural de mesma arquitetura, 4 neurônios na camada escondida, são necessário os recursos físicos apresentados na Tabela 4.11

Tabela 4.11: Requisitos físicos para sintetizar uma rede neural recorrente estocástica com 4 neurônios.

Elementos Lógicos	3,178 / 114,480 (3%)
Registradores	1606
Bits de Memória	73,728 / 3,981,312 (2%)
Multiplicador de 9-*bits*	0 / 532 (0%)

Pelas Tabelas 4.10 e 4.11, é possível notar que, para a mesma estrutura de rede neural, uma implementação utilizando computação estocástica reduz em aproximadamente dez vezes o uso de registradores, além de reduzir, também, em aproximadamente dez vezes a quantidade de elementos lógicos. Em contrapartida, a computação estocástica utiliza-se de maior quantidade de *bits* de memória em relação a implementação de ponto flutuante. A quantidade de multiplicadores de 9 *bits* é nula para a implementação estocástica, por outro lado, a implementação de ponto flutuante varia de 32% a 67% do uso total.

Outra implementação de rede neural recorrente feita em FPGA foi desenvolvida por *Makiuchi* [3]. Tal implementação foi feita utilizando aritmética de ponto fixo.

A síntese de uma rede neural recorrente em ponto fixo de 12 neurônios na camada escondida e 2 neurônios na camada de saída é apresentada na Tabela 4.12.

Tabela 4.12: Requisitos físicos da rede neural recorrente em ponto fixo com 12 neurônios na camada escondida e 2 neurônios na camada de saída [3]

Recurso	Quantidade utilizada	Ocupação do FPGA (%)
Elementos lógicos	100.922	88
Registradores	6.460	-
Pinos de entrada/saída	108	20
Multiplicadores de 9 *bits*	532	100

A Tabela 4.13 mostra os requisitos físicos necessários para sintetizar uma rede neural estocástica com 12 neurônios na camada escondida e 2 neurônios na camada de saída.

Tabela 4.13: Requisitos físicos da rede neural estocástica com 12 neurônios na camada escondida e 2 neurônios na camada de saída.

Elementos Lógicos	14,397 / 114,480 (13%)
Registradores	6154
Bits de Memória	73,728 / 3,981,312 (2%)
Multiplicador de 9-*bits*	0 / 532 (0%)

Para a mesma estrutura de rede, novamente a implementação estocástica apresenta vantagens em termos de requisitos físicos para síntese em FPGA. Apenas 13% de elementos lógicos são necessários, valor bem inferior aos 88% da implementação por ponto fixo. Todos os multiplicadores de 9 *bits* são utilizados, enquanto nenhum é utilizado na implementação

estocástica. A quantidade de registradores utilizados pela implementação estocástica também é inferior em relação a implementação de ponto fixo, apresentando apenas 306 registradores a menos.

De acordo com os valores apresentados, é possível concluir que a implementação estocástica de rede neural recorrente requer uma quantidade menor de recursos físicos para síntese em FPGA, sendo possível a síntese de redes com maiores quantidades de neurônios e resolução de problemas mais complexos em *hardware*, além de apresentar as vantagens que a computação estocástica traz ao sistema, como tolerância a erros, baixo consumo de energia e alta confiabilidade.

Capítulo 5

Conclusão

Este trabalho teve como objetivo a implementação e treinamento de redes neurais artificiais recorrentes utilizando computação estocástica como alternativa às convenções binárias de ponto flutuante e ponto fixo. Dessa forma, operações aritméticas necessárias para cálculos dos algoritmos de *feedforward* e *backpropagation* são realizadas no domínio estocástico, de modo que menos recursos sejam utilizados para síntese do circuito a ser implantando em FPGA.

Foi implementado, utilizando o *software Matlab*, uma rede neural recorrente simulando o comportamento das operações aritméticas estocásticas A partir desta implementação, foram treinadas diversas estruturas de redes neurais recorrentes estocásticas utilizando o problema clássico da literatura de redes neurais, o problema da operação XOR adaptada para o domínio do tempo. Comprovada a corretude e convergência da implementação, um problema simples de identificação de fonema foi utilizado no treinamento de diversas redes com quantidade de neurônios nas camadas escondida e de contexto. Nesta etapa, foi verificado que as redes obtiveram boas taxa de acerto sobre a base de treinamento, alcançando até 98% de acerto.

Os resultados obtidos comprovam o funcionamento da implementação da rede neural recorrente estocástica, bem como consumo de recursos do *FPGA* inferiores a implementações da mesma arquitetura de rede utilizando as convenções binárias de ponto fixo e ponto flutuante, conforme apresentado no Capítulo 4. A configuração 1-4-1 para ponto flutuante apresenta de 25% a 34% de consumo dos elementos lógicos do *FPGA*, além de consumir pelo menos 14.142 registradores. Tais valores apresentam-se superiores em relação a proposta deste trabalho, o qual necessita de apenas 3% do uso de elementos lógicos et 1.606 registradores. Comparando-se a implementação de ponto fixo com configuração 1-12-2, 88% dos elementos lógicos são consumidos pela síntese do circuito e 6.460 registradores são necessários. A rede neural recorrente estocástica necessita apenas de 13% de elementos lógicos e 6.154 registradores. Dessa forma, demonstra-se a possibilidade de implementação

de grandes redes neurais para a solução de problemas complexos que não seriam possíveis resolver com ponto flutuante e ponto fixo. Uma demonstração em vídeo do problema XOR temporal no *FPGA* foi apresentada neste trabalho, de modo que fosse possível comprovar o funcionamento do circuito no dispositivo reconfigurável.

Neste trabalho não foi abordado o treinamento da rede neural recorrente estocástica direto no dispositivo FPGA, uma vez que o *backpropagation* para uma rede neural organizada com camada de contexto, seguindo a arquitetura de *Elman*, não converge adequadamente nesta implementação utilizando computação estocástica, necessitando mais estudos. Desse modo, sugere-se como trabalho futuro uma revisão dos circuitos resultantes da configuração recorrente da rede estocástica para depuração do algoritmo e alterações para que seu treinamento venha a convergir. A seguir, sugere-se ainda um estudo para a realização de treinamento *online*, para que seja possível utilizar tal sistema adaptativo em aplicações de problemas temporais, com baixo custo de hardware.

Referências

[1] D. Zhang e H. Li. A stochastic-based FPGA controller for an induction motor drive with integrated neural network algorithms. *IEEE Trans. Industrial Electronics*, 55:551–561, 2008. viii, 11

[2] José R. C. S. A. V. Silva Neto. Implementação de Redes Neurais Artificiais Perceptron e Recorrentes em FPGA Utilizando Aritmética em Ponto Flutuante. Trabalho de Conclusão de Curso (Bacharelado em Engenharia Mecatrônica), Universidade de Brasília, 2019. x, 61, 62

[3] Mariana R. Makiuchi. Desenvolvimento de Rede Neural Artificial Recorrente em FPGA para Previsão online de Oportunidades em Transmissões Oportunísticas em Redes de Comunicação Wireless. Trabalho de Conclusão de Curso (Bacharelado em Engenharia Mecatrônica), Universidade de Brasília, 2018. Disponível em `http://bdm.unb.br/handle/10483/22068`. x, 63

[4] R. Haycock e T. York. Hardware implementation of Boolean neural networks. In *IEE Colloquium on Hardware Implementation of Neural Networks and Fuzzy Logic*, pages 3/1–3/4, 1994. 1

[5] Armin Alaghi e John P. Hayes. Survey of Stochastic Computing. *ACM Transactions on Embedded computing systems (TECS)*, 12(2s), 2013. 3, 4, 5, 7, 9

[6] Marc D. Riedel Kia Bazargan Xin Li, Weikang Qian e David J. Lilja. A Reconfigurable Stochastic Architecture for Highly Reliable Computing. In *Proceedings of the Great Lakes Symposium on VLSI*, pages 315–320, 2009. 8

[7] Cohen D. A. Jeavons, P. e J. Shawe-Taylor. Generating binary sequences for stochastic computing. In *Proceedings of the Great Lakes Symposium on VLSI*, page 716–720, 1994. 10

[8] Jeffrey L. Elman. Finding Structure in Time. *Cognitive Science*, 14:179–211, 1990. 10, 21, 22, 46, 47

[9] Quero J. M. Toral, S. L. e L. G. Franquelo. Stochastic pulse coded arithmetic. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, page 599–602, 1999. 11

[10] P. Mars e H. R. McLean. High-speed matrix inversion by stochastic computer. *Electronics Letters*, 18:457–459, 1976. 11

[11] W. Qian e M. D. Riedel. Stochastic pulse coded arithmetic. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, page 648–653, 2008. 11

[12] Li X. Riedel M. D. Bazargan K. Qian, W. e D. J. Lilja. An architecture for fault-tolerant computation with stochastic logic. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, page 93–105, 2011. 11

[13] H. Aliee e H. R. Zarandi. Fault tree analysis using stochastic logic: A reliable and high speed computing. In *Proceedings of the Reliability and Maintainability Symposium*, page 1–6, 2011. 11

[14] H. Chen e J. Han. Stochastic computational models for accurate reliability evaluation of logic circuits. In *Proceedings of the Great Lakes Symposium on VLSI*, pages 61–66, 2010. 11

[15] J. F. Keane e L. E. Atlas. Impulses and stochastic arithmetic for signal processing. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1257–1260, 2001. 11

[16] D. Pejic e V. Vujicic. Accuracy limit of high precision stochastic watt-hour meter. *IEEE Trans. Instrumentation and Measurements*, page 617–620, 1999. 11

[17] B. D. Brown e H. C Card. Stochastic Neural Computation I: Computational Elements. *IEEE Trans. Comput.*, 50:891–905, 2001. 11

[18] McLeod R. D. Dickson, J. A. e H. C Card. Stochastic arithmetic implementations of neural networks with in situ learning. In *Proceedings of the International Conference on Neural Networks*, pages 711–716, 1993. 11

[19] Y-C. Kim e M. A. Shanblatt. Architecture and statistical model of a pulse-mode digital multilayer neural network. *IEEE Trans. Neural Networks*, 6:1109–1118, 1995. 11

[20] Zhao L. Das S. R. Groza V. Z. Petriu, E. M. e A. Cornell. Instrumentation applications of multibit random-data representation. *IEEE Trans. Instrumentation and Measurement.*, 52:175–181, 2003. 11

[21] Cirstea M. N. Dinu, A. e M. McCormick. Stochastic implementation of motor controllers. In *Proceedings of the IEEE Symposium on Industrial Electronics*, pages 639–644, 2002. 11

[22] Reboul J. M. Q. Marin, S. L. T. e L. G. Franquelo. Digital stochastic realization of complex analog controllers. *IEEE Trans. Industrial Electronics*, 49:1101–1109, 2002. 11

[23] Nilson M. Bermak A. Hammadou, T. e P. Ogunbona. A 96 x 64 intelligent digital pixel array with extended binary stochastic arithmetic. In *Proceedings of the International Symposium on Circuits and Systems*, volume IV, pages 772–775, 2003. 12

[24] Simon Haykin. *Neural Networks: A Comprehensive Foundation.* Pearson Education, 2nd edition, 2005. 12, 14, 15, 46

[25] M.I. Jordan. Serial order: A parallel distributed processing approach. Technical Report No.8604, San Diego: University of California, Institute for Cognitive Science, 1986. 22

[26] A. Cochocki e Rolf Unbehauen. *Neural networks for optimization and signal processing.* John Wiley & Sons, Inc., 1993. 23, 24

[27] Altera Corporation. *Cyclone IV Device Handbook*, 2016. 25, 26

[28] N. Mohana Sundaram e P. N. Ramesh. Optimization of training phase of Elman neural networks by suitable adjustments on the network parameters. In *Proceedings of International Conference on Systems, Science, Control, Communication, Engineering and Technology*, 2015. 32

[29] Lucas N. Carvalho. Projeto e Treinamento de Redes Neurais em Hardware FPGA usando Computação Estocástica. Trabalho de Conclusão de Curso (Bacharelado em Engenharia de Computação), Universidade de Brasília, 2016. Disponível em `http://bdm.unb.br/handle/10483/22823`. 33, 36

[30] Bradley D. Brown e Howard C. Card. Stochastic neural computation I: Computational elements. *IEEE Transactions on Computers*, 50(9):891–905, 2001. 36, 38

[31] T. Hanazawa G. Hinton K. Shikano Waibel, A. e K. J. Lang. Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37:328–339, 1989. 50