TRABALHO DE GRADUAÇÃO

Implementação de Redes Neurais Artificiais Perceptron e Recorrentes em FPGA Utilizando Aritmética em Ponto Flutuante

José Reinaldo da C.S.A.V da Silva Neto

Brasília, Julho de 2019

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia
Curso de Graduação em Engenharia de Controle e Automação

TRABALHO DE GRADUAÇÃO

Implementação de Redes Neurais Artificiais Perceptron e Recorrentes em FPGA Utilizando Aritmética em Ponto Flutuante

José Reinaldo da C.S.A.V da Silva Neto

Relatório submetido como requisito parcial de obtenção

de grau de Engenheiro de Controle e Automação

Banca Examinadora

Prof. Dr. Marcus Vinicius Lamar, CIC/UnB ____________________
Orientador

Prof. Dr. Ricardo Pezzuol Jacobi, CIC/UnB ____________________
Examinador interno

Prof. Dr. Marcelo Grandi Mandelli, CIC/UnB ____________________
Examinador interno

Brasília, Julho de 2019

FICHA CATALOGRÁFICA

DA C.S.A.V. DA SILVA NETO, JOSÉ REINALDO
Implementação de Redes Neurais Artificiais Perceptron e Recorrentes em FPGA Utilizando Aritmética em Ponto Flutuante

[Distrito Federal] 2019.

viii, 63p., 297 mm (FT/UnB, Engenheiro, Controle e Automação, 2019). Trabalho de Graduação – Universidade de Brasília.Faculdade de Tecnologia.

1. Redes Neurais Perceptron 2.Redes Neurais de Elman
3. FPGA

I. Mecatrônica/FT/UnB II. Título (Série)

REFERÊNCIA BIBLIOGRÁFICA

DA C.S.A.V. DA SILVA NETO, JOSÉ REINALDO, (2019). Implementação de Redes Neurais Artificiais Perceptron e Recorrentes em FPGA Utilizando Aritmética em Ponto Flutuante. Trabalho de Graduação em Engenharia de Controle e Automação, Publicação FT.TG-n°006, Faculdade de Tecnologia, Universidade de Brasília, Brasília, DF, 63p.

CESSÃO DE DIREITOS

AUTOR: José Reinaldo da Cunha Santos Aroso Vieira da Silva Neto

TÍTULO DO TRABALHO DE GRADUAÇÃO: Implementação de Redes Neurais Artificiais Perceptron e Recorrentes em FPGA Utilizando Aritmética em Ponto Flutuante.

GRAU: Engenheiro ANO: 2019

É permitida à Universidade de Brasília permissão para reproduzir cópias deste Trabalho de Graduação e para emprestar ou vender tais cópias para propósitos acadêmicos e científicos. O autor reserva outros direitos de publicação e nenhuma parte deste Trabalho de Graduação pode ser reproduzida sem autorização por escrito do autor.

José Reinaldo da Cunha Santos Aroso Vieira da Silva Neto

SMPW quadra 09 conjunto 01 lote 05 - Park Way

71741-001 Brasília– DF – Brasil.

Dedicatória

Dedico este trabalho a todos aqueles que me acompanharam e me deram suporte ao longo desta extensa e nada trivial jornada universitária. Em especial aos meus pais, Joana e Marcos e à minha irmã Marina por sempre estarem ao meu lado mesmo nos momentos mais difíceis.

José Reinaldo da C.S.A.V da Silva Neto

Agradecimentos

Primeiramente agradeço aos meus Pais Joana e Marcos e minha irmã Marina por sempre estarem ao meu lado e por vezes sofrerem mais do que eu mesmo pelas adversidades em minha vida.
Tenho de agradecer, também, aos meus amigos de faculdade por me aguentarem por 5 anos completos que imagino não ser uma tarefa trivial. Reitero aqui que o ocorrido com o baralho não foi culpa minha.
Aos meus avós José Reynaldo e Evani que apesar de todos os impasses políticos sempre se preocuparam, por vezes até demais, com a minha pessoa.
Agradeço também ao meu orientador Prof. Marcus Vinicius Lamar por ter aceitado me orientar e principalmente pela paciência divina com a qual sempre me ajudou.
Tecidos os agradecimentos individuais deixo aqui meus agradecimentos a todos que me apoiaram nesta tragicomédia que chamei de vida universitária. Se pudesse voltar no tempo faria tudo novamente.

The bliss of comprehension!

José Reinaldo da C.S.A.V da Silva Neto

RESUMO

Para a implementação de redes neurais artificiais perceptron e recorrentes em chips FPGA foram desenvolvidos módulos de funções de ativação sigmoide logística e tangente hiperbólica utilizando técnicas de aproximações polinomiais, aproximação em amostras de intervalos fixos e implementação exata utilizando blocos IP de exponenciação. Foi observado que as funções com maiores utilizações de recursos físicos também apresentavam os menores erros de aproximação. Utilizando as funções de ativação desenvolvidas e módulos IP de operações aritméticas em ponto flutuante propomos uma ferramenta que fosse capaz de converter automaticamente redes definidas em software para linguagem de descrição de hardware. Os códigos resultantes da conversão poderiam, então, ser diretamente embarcados em chip FPGA. De forma a validar o funcionamento da ferramenta quatro casos de teste foram propostos, sendo dois deles relativos a redes *feedforward* e dois a redes recorrentes. O menor erro obtido entre as redes embarcadas e as em software foi de $1,1968 \times 10^{-13}$ enquanto que o maior foi de $0,0366$.

Palavras Chave: FPGA, rede neural perceptron, rede neural recorrente, aritmética de ponto flutuante.

ABSTRACT

For the implementation of perceptron and recurrent artificial neural networks in FPGA chips, we developed logistic sigmoid and hyperbolic tangent activation function modules by polynomial and fixed-interval samples approximations and exact implementation using exponentiation IP blocks. It was observed that those that required more hardware resources were also the ones with the smaller approximation errors. By using the developed activation functions and floating-point arithmetic operations IP blocks we propose a tool that were able to automatically convert software defined neural networks into hardware description language. The resulting codes could then be directly embarked into an FPGA chip. As a means to validate the tool's correct operation, four test cases were proposed, two of them being related to feedforward networks while the other two were related to recurrent networks. The smallest obtained error between embarked and software networks was $1,1968 \times 10^{-13}$ while the biggest was $0,0366$.

Keywords: FPGA, perceptron neural network, recurrent neural network, floating-point arithmetic.

SUMÁRIO

1 Introdução ... **1**
1.1 Motivação ... 1
1.2 Objetivos ... 2
1.3 Estrutura do documento ... 2

2 Fundamentação Teórica ... **3**
2.1 Representações Numéricas Binárias ... 3
2.1.1 Representação em Ponto Fixo ... 3
2.1.2 Representação em Ponto Flutuante ... 4
2.1.3 Padrão IEEE 754 ... 4
2.2 Redes neurais ... 5
2.2.1 Neurônios artificiais ... 6
2.2.2 Funções de ativação ... 7
2.2.3 Redes Perceptron ... 8
2.2.4 Redes recorrentes ... 8
2.3 Algoritmos de treinamento ... 9
2.4 FPGA ... 10
2.4.1 Implementação das células lógicas ... 10
2.4.2 Placa Altera DE2-115 ... 11
2.5 Protocolo de comunicação ... 11
2.5.1 Recommended Standard 232 - RS232 ... 11
2.5.2 Universal Asynchronous Receiver-Transmitter - UART ... 13

3 Sistema Proposto ... **15**
3.1 Módulos de propriedade intelectual utilizados ... 15
3.2 Função de ativação ... 16
3.2.1 Implementação por aproximação polinomial ... 16
3.2.2 Implementação utilizando módulo exponencial ... 21
3.2.3 Implementação por aproximação em intervalos fixos ... 22
3.3 Neurônio ... 25
3.4 Realimentações ... 27
3.5 Rede Neural ... 28

3.6 Ferramenta para geração automática do código Verilog a partir da rede em MATLAB 29
3.7 Comunicação de dados entre DE2-115 e MATLAB..... 32

4 Resultados Obtidos..... 35
4.1 Comunicação entre FPGA e MATLAB..... 35
4.2 Implementações da função de ativação 35
4.2.1 Implementação da função sigmoide logística 36
4.2.2 Implementação da função tangente hiperbólica..... 40
4.3 Casos de teste..... 44
4.3.1 Rede neural perceptron para implementação de porta lógica XOR. 44
4.3.2 Rede perceptron para classificação de espécies da flor de íris 46
4.3.3 Rede recorrente para realização de operação lógica XOR temporal 48
4.3.4 Rede de Elman para previsão *online* de oportunidades em transmissões oportunísticas em redes de comunicação *wireless* 51
4.3.5 Limite físico para implementação de redes neurais no kit de desenvolvimento DE2-115..... 52

5 Conclusões 53
5.1 Trabalhos Futuros 54

REFERÊNCIAS BIBLIOGRÁFICAS 55

Anexos..... 57

I Códigos utilizados no treinamento das redes neurais..... 58

II Formas de onda 61

LISTA DE FIGURAS

2.1 Representação de ponto fixo ... 4
2.2 Cadeia de bits para representação em ponto flutuante IEEE 754 de precisão simples e dupla ... 5
2.3 Representação de um neurônio artificial[1] ... 6
2.4 Representação de funções de ativação ... 7
2.5 Representação de uma rede neural artificial perceptron com uma camada escondida.. 8
2.6 Representação de uma rede de Elman[2] ... 9
2.7 Representação simplificada de uma FPGA ... 10
2.8 Representação simplificada de uma célula lógica[3] ... 11
2.9 Diagrama da placa Terasic DE2-115[4] ... 12
2.10 Conectores com 9 e 25 pinos para comunicação serial RS232[5] ... 14
2.11 Sequência de bits para transmissão de dados UART ... 14

3.1 Diagrama para implementação de função de ativação sigmoide logística aproximada por polinômio de grau 2 ... 19
3.2 Diagrama para implementação de função de ativação tangente hiperbólica aproximada por polinômio de grau 2 ... 20
3.3 Módulo para implementação exata da função de ativação sigmoide logística ... 21
3.4 Módulo para implementação exata da função de ativação tangente hiperbólica ... 21
3.5 Diagrama para implementação da função de ativação sigmoide logística por amostragem em intervalos fixos ... 24
3.6 Diagrama para implementação da função de ativação tangente hiperbólica por amostragem em intervalos fixos ... 24
3.7 Diagrama de um neurônio com 3 entradas ... 26
3.8 Máquina de estados para o recebimento, processamento e envio dos dados entre MATLAB e DE2-115 ... 32

4.1 Resultados das implementações por aproximação polinomial de grau 1, 2 e 3 para a função sigmoide logística ... 38
4.2 Resultados das implementações de aproximação da função sigmoide logística por amostragem ... 39
4.3 Resultado da implementação exata da função sigmoide logística ... 39
4.4 Resultados das implementações por aproximação polinomial de grau 1, 2 e 3 para a função tangente hiperbólica ... 42

4.5 Resultados das implementações da função tangente hiperbólica por amostragem em intervalos fixos 43
4.6 Resultado da implementação exata da função tangente hiperbólica 43
4.7 Estrutura da rede perceptron para resolução do problema XOR 44
4.8 Estrutura da rede perceptron para classificação da flor de íris 47
4.9 Estrutura da rede neural recorrente para implementação da lógica XOR temporal 49
4.10 Estrutura da rede para previsão de oportunidades em transmissões oportunísticas em redes de comunicação *wireless* 51

II.1 Forma de onda das implementações das funções de ativação sigmoide logísticas 62
II.2 Forma de onda das implementações das funções de ativação tangente hiperbólica 63

LISTA DE TABELAS

2.1 Quantidade de bits para as precisões definidas pelo IEEE 754 5
2.2 Sinais definidos pelo protocolo RS232 13

3.1 Módulos de IP para operações de ponto flutuante. 16
3.2 Intervalos e polinômios para as aproximações polinomiais das funções sigmoide logística e tangente hiperbólica. 19
3.3 Tempo de execução em ciclos de *clock* para as funções de ativação implementadas com aproximação polinomial 20
3.4 Tempo de execução em ciclos de *clock* para as funções de ativação implementadas com módulos IP de exponenciação 22
3.5 Tempo de execução em ciclos de *clock* para as funções de ativação com aproximação por amostragem em intervalos fixos 25

4.1 Recursos de hardware para a comunicação entre MATLAB e FPGA 35
4.2 Custo de hardware para as implementações da função sigmoide logística 37
4.3 Erros das implementações da função de ativação sigmoide logística. 37
4.4 Custo de hardware para as implementações da função tangente hiperbólica 40
4.5 Erros das implementações da função de ativação tangente hiperbólica 41
4.6 Custo da implementação da rede neural perceptron para implementação de porta lógica XOR 45
4.7 Resultados obtidos pelas diferentes implementações da rede perceptron para problema XOR 45
4.8 Latência em segundos para as redes neurais para o problema da porta lógica XOR ... 46
4.9 Custo da implementação da rede neural perceptron para classificação da flor de íris .. 47
4.10 Erros associados a cada implementação da rede neural para classificação da flor de íris quando comparados aos resultados da rede em MATLAB 48
4.11 Latência em segundos para as redes neurais de classificação da flor de íris. 48
4.12 Custo da implementação da rede neural recorrente para resolver o problema XOR temporal 49
4.13 Erros associados a cada implementação da rede neural para resolução do problema XOR temporal 50
4.14 Latência em segundos para as redes neurais para o problema do xor temporal. 50
4.15 Tabela de utilização de recursos de hardware para a implementação da rede neural para previsões de oportunidades de transmissão em redes sem fio 51

4.16 Recursos físicos utilizados para a implementação das redes neurais para o problema do XOR para teste de limite físico da FPGA 52

LISTA DE SÍMBOLOS

Símbolos Gregos

ω	Peso sináptico do neurônio
ϕ	Função de ativação

Siglas

ASIC	*Application-Specific Integrated Circuit*
FPGA	*Field-Programmable Gate Array*
IEEE	*Institute of Electrical and Electronics Engineers*
MLP	Perceptron Multicamadas do inglês *MultiLayer Perceptron*
HDL	Linguagem de descrição de hardware do inglês *Hardware description language*
RNA	Rede Neural Artificial
IP	Propriedade Intelectual do inglês *Intellectual property*

Capítulo 1

Introdução

O desejo por implementações de redes neurais em hardware não é recente[6]. Uma das grandes motivações para tal é a expectativa de aceleração do processamento em hardware quando comparado a implementações em software. A fabricação de chips ASIC (*Application Specific Integrated Circuit*) para a implementação de redes neurais traz a vantagem de ser realizada de modo eficiente em hardware, mas limita a rede a uma única topologia e demanda esforço para reconfigurar sua estrutura. Para cobrir esta deficiência, pode-se utilizar um chip FPGA (*Field Programable Gate Array*) que têm por característica ser facilmente reconfigurável.

A facilidade com que se pode reprogramar um chip FPGA traz grandes vantagens para a realização de testes de implementações em hardware. No contexto de redes neurais, o conjunto de pesos e a topologia da rede podem ser alterados com maior facilidade, quando comparado a sistemas ASIC.

1.1 Motivação

A criação e o treinamento de redes neurais artificiais perceptron e recorrentes no software MATLAB (ou no seu equivalente gratuito Octave) pode ser realizada de maneira simples e rápida, uma vez que diversas funções de preprocessamento, definições de topologias da rede e algoritmos de treinamento já estão implementadas para esta ferramenta. Implementações em hardware, por outro lado, são mais complexas de serem realizadas e são utilizadas para diminuir o tempo de processamento e gasto de energia. Propõe-se, neste trabalho, juntar a facilidade da criação e treinamento da rede em software com as vantagens obtidas por implementação em hardware através de uma ferramenta de conversão automática de redes definidas em software para representações em hardware.

1.2 Objetivos

De modo a possibilitar a criação e treinamento de redes neurais em software e facilitar a implementação destas em hardware, propomos neste trabalho uma ferramenta capaz de realizar a conversão automática de redes em linguagem MATLAB para linguagem de descrição de hardware Verilog, este é nosso principal objetivo. Para a realização desta ferramenta iremos projetar módulos de funções de ativação capazes de aproximar aquelas implementadas em software. Para a validação estudaremos 4 casos de teste sendo eles:

- Implementação de porta lógica XOR por redes neurais perceptron.
- Classificação das espécies de flor de íris por redes neurais perceptron.
- Realização de lógica XOR temporal por redes neurais recorrentes.
- Previsão *online* de oportunidades de transmissões em redes de comunicação *wireless* por redes neurais recorrentes.

As redes treinadas em MATLAB para a resolução dos casos de teste serão convertidas para Verilog e embarcadas no chip FPGA de forma a validar o correto funcionamento da ferramenta criada.

1.3 Estrutura do documento

Este trabalho é composto por 5 capítulos:

- Este capítulo apresenta uma breve introdução ao assunto proposto, bem como a motivação por trás da síntese de redes neurais em hardware e a apresentação dos objetivos a serem atingidos neste trabalho..
- O segundo capítulo detalha os fundamentos teóricos necessários para a realização do trabalho proposto por este documento, explicando com mais detalhes o funcionamento geral das redes neurais multi-camadas e recorrentes, modelo dos neurônios utilizados, formas de funções de ativação e uma introdução aos dispositivos e protocolos de comunicação utilizados.
- O terceiro capítulo descreve como foram realizadas as implementações em hardware dos elementos da rede como neurônios, função de ativação, realimentação e a própria rede neural. Também apresenta a ferramenta criada para conversão automática da estrutura da rede em MATLAB para código Verilog, e introduz os casos de teste que serão utilizados para comparação e validação das redes neurais implementadas em hardware.
- O quarto capítulo mostra os resultados obtidos para as implementações propostas das funções de ativação e para os casos de teste.
- O quinto capítulo conclui o trabalho resumindo os resultados obtidos no capítulo 4 através das implementações propostas no capítulo 3.

Capítulo 2

Fundamentação Teórica

Neste capítulo são apresentadas as bases de conhecimento necessárias para o entendimento e execução do projeto de implementação de redes neurais artificiais em FPGA. Para isto, apresentam-se os seguintes tópicos nesta seção: Representação de um valor numérico em ponto flutuante padrão IEEE 754, informações básicas sobre redes neurais artificiais perceptron e recorrentes e os elementos que as compõem e a estrutura básica de um chip FPGA juntamente com protocolos capazes de realizar a comunicação serial entre um computador pessoal e o kit de desenvolvimento com o chip FPGA.

2.1 Representações Numéricas Binárias

A representação de números reais usando apenas dígitos binários envolve a definição de padrões que indiquem o significado de cada bit. As formas mais comuns utilizadas hoje em dia são as representações em ponto fixo e em ponto flutuante.

2.1.1 Representação em Ponto Fixo

Como podemos representar um número real utilizando um número finito de bits em um computador? As duas soluções mais conhecidas são a utilização de notação de ponto fixo e de ponto flutuante. A notação de ponto fixo é aquela que mais se assemelha à notação tradicional de complemento de dois com a diferença que agora uma quantidade dos bits menos significativos representa potências negativas do número dois. Ou seja, como exemplificado na Figura 2.1, o bit que representa a potência 2^0 não será mais obrigatoriamente o menos significativo da cadeia, a posição deste bit será agora definida em projeto e os bits à direita deste ponto serão potências negativas, representando números fracionários, enquanto os bits à esquerda representarão as potências positivas de dois.

Note que na Figura 2.1 foi escolhido três dígitos fracionários. Desta forma se quisermos calcular o valor decimal x que esta cadeia de bits representa basta realizar

$$x = -b_7 \times 2^4 + b_6 \times 2^3 + b_5 \times 2^2 + b_4 \times 2^1 + b_3 \times 2^0 + b_2 \times 2^{-1} + b_1 \times 2^{-2} + b_0 \times 2^{-3}. \quad (2.1)$$

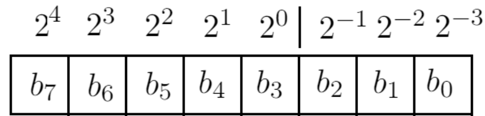


Figura 2.1: Representação de ponto fixo

A notação de ponto fixo possui grande similaridade com a notação de complemento de dois utilizada para operações com números inteiros. Esta é uma grande vantagem pois circuitos de aritmética inteira podem ser aproveitados, com pequenas modificações, para operações em ponto fixo. Por este motivo esta é uma representação mais simples de ser utilizada se comparada à notação de ponto flutuante que veremos a seguir.

2.1.2 Representação em Ponto Flutuante

A notação de ponto flutuante, diferente da de ponto fixo, possui uma estrutura de ordenação dos bits diferente da notação de complemento de dois. A aritmética de ponto flutuante consiste na representação de um número em uma base predefinida na forma de notação científica

$$x = (mantissa) \times (base)^{expoente}. \tag{2.2}$$

Ou seja, em base decimal a representação em ponto flutuante do número $105 = (1,05) \times (10)^2$. Uma das vantagens da utilização de notação em ponto flutuante é o fato de que números representados nesta notação são capazes de, com a mesma quantidade de bits de números em ponto fixo, alcançar uma faixa maior de valores com maior precisão. Veremos na Subseção 2.1.3 um pouco mais sobre como é definido um número em notação de ponto flutuante, mais especificamente o definido pelo padrão IEEE 754.

2.1.3 Padrão IEEE 754

Na computação, a representação em ponto flutuante mais utilizada é a de base 2 definida pelo padrão IEEE 754 [7]. Este padrão define o formato e os métodos para implementação de aritmética em ponto flutuante para sistemas de computação em base 2 e base 10. Ele também garante que implementações das operações aritméticas possam ser realizadas em software, hardware ou uma combinação de ambos sem que haja diferença nos resultados obtidos desde que as entradas e a operação realizada sejam iguais. Além da garantia do resultado, o padrão também apresenta recomendações de formas de conversão entre diferentes formatos de representação de dados.

Um valor é representado em notação de ponto flutuante no padrão IEEE 754 como

$$x = (-1)^{sinal} \times (2^0 + mantissa) \times 2^{(expoente-offset)}. \tag{2.3}$$

Um fato importante a ser lembrado é o de que a mantissa contém um bit de valor 1 mais significativo implícito na sua representação que aparece na equação 2.3 pelo valor 2^0 somado à mantissa. Note,

também, que foi subtraído um *offset* do expoente. Esta subtração nos permite representar o expoente em uma cadeia de bits como um valor *unsigned*.

O padrão IEEE 754 revisado em 2008 define quatro precisões para implementação de aritmética de ponto flutuante, sendo elas: precisão simples de 32 bits, precisão dupla de 64 bits, precisão *quad* de 128 bits e precisão *half* de 16 bits. A quantidade de bits para sinal, mantissa e expoente assim como o valor do *offset* para cada uma delas é apresentada na tabela 2.1.

Tabela 2.1: Quantidade de bits para as precisões definidas pelo IEEE 754

Parâmetro	Precisão half	Precisão simples	Precisão dupla	Precisão quad
Bits de sinal	1	1	1	1
Bits de expoente	5	8	11	15
Bits de mantissa	10	23	52	112
Valor de offset	15	127	1023	16383

A disposição da cadeia de bits para as representações de ponto flutuante de precisão simples e dupla são apresentadas na Figura 2.2. O padrão de ordenamento de bits para as precisões simples e dupla se mantêm para os casos *half* e *quad*, sendo o bit mais significativo para o sinal, seguido pelos bits que representam o expoente e em seguida a mantissa.

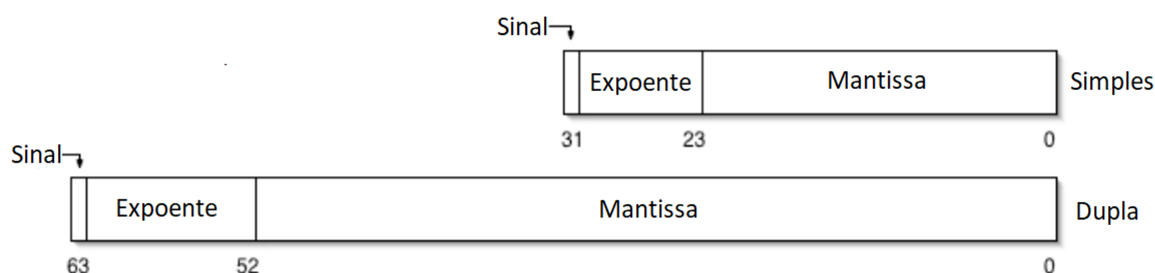


Figura 2.2: Cadeia de bits para representação em ponto flutuante IEEE 754 de precisão simples e dupla

2.2 Redes neurais

Sistemas bio-inspirados são objetos de estudo nas mais diversas áreas do conhecimento, e o cérebro humano não é exceção. Sendo considerado um processador não-linear e altamente paralelo [8], percebe-se as vantagens em se obter um modelo matemático que emule o funcionamento do cérebro para o processamento de informações. As redes neurais artificiais (RNA) surgiram de

tentativas de modelar tal sistema e, assim como sua contraparte biológica, é formada por neurônios artificiais interconectados através de sinapses. A estrutura mais simples de uma rede neural artificial é a conhecida por *feedforward* densamente conectada, sendo os neurônios agrupados em camadas consecutivas e com todos os neurônios de uma determinada camada se conectando a todos os outros da camada posterior. Um exemplo deste tipo de rede é a RNA perceptron multicamadas apresentada na Seção 2.2.3.

2.2.1 Neurônios artificiais

Neurônios artificiais são os elementos atômicos de uma rede neural artificial. Eles são os responsáveis pelo processamento das informações recebidas pela rede. Nas redes artificiais perceptron e recorrentes baseadas em modelos perceptron, o neurônio é constituído de três elementos básicos: Um conjunto de ligações ponderadas também chamadas de sinapses, um somador e uma função de ativação. O resultado apresentado na saída do somador é chamado de campo local induzido do neurônio. A representação de um neurônio é apresentada na Figura 2.3.

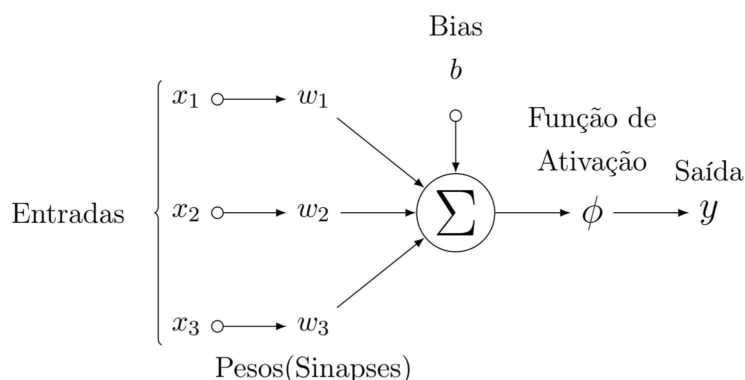


Figura 2.3: Representação de um neurônio artificial[1]

Temos, então, que um neurônio processa a informação de entrada de acordo com

$$y_k = \phi(\sum \omega_{kj} \cdot x_j + b_k), \tag{2.4}$$

sendo $\phi(\cdot)$ a função de ativação, ω_{kj} o peso da ligação entre o j-ésimo neurônio da camada anterior e o k-ésimo neurônio da camada atual, x_j o valor de saída do neurônio j da camada anterior, b_k e y_k o bias e a saída do k-ésimo neurônio da camada atual, respectivamente. De todos os elementos apresentados com partes constituintes do neurônio, o único desconhecido até o momento é a função de ativação. Vamos, então, dar uma olhada em alguns exemplos de funções de ativação e qual o seu papel no processamento de informações da rede neural na Subseção 2.2.2.

2.2.2 Funções de ativação

Funções de ativação são responsáveis pelo mapeamento entre o campo local induzido do neurônio e sua saída e geralmente são utilizadas de forma a limitar o valor de saída do neurônio. Algumas das funções de ativação conhecidas na literatura são:

- função sigmoide logística:
$$\phi(x) = \frac{1}{1+e^{-x}} \tag{2.5}$$

- função sigmoide tangente-hiperbólica:
$$\phi(x) = tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \tag{2.6}$$

- função degrau:
$$\phi(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases} \tag{2.7}$$

Aqui serão utilizadas as funções sigmoide logística e tangente hiperbólica apresentadas. Note que a sigmoide logística é capaz de mapear entradas para o intervalo $[0, 1]$, sendo por isto chamada de função de ativação unipolar. Funções de ativação sigmoides em que a saída varia em intervalo $[-1, 1]$, como a tangente hiperbólica são chamadas de função de ativação bipolares. A Figura 2.4 mostra as funções de ativação apresentadas nas Equações 2.5, 2.6 e 2.7.

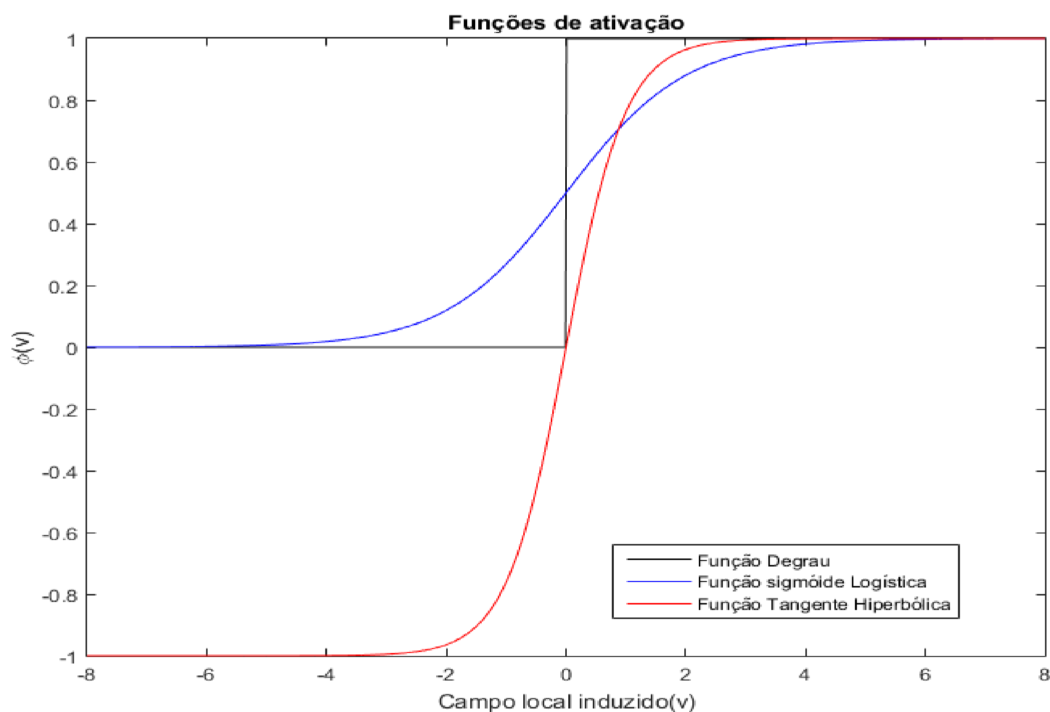


Figura 2.4: Representação de funções de ativação

2.2.3 Redes Perceptron

É uma das estruturas de RNAs mais conhecidas e amplamente utilizada devido a sua simplicidade. Ela é caracterizada como uma rede *Feedforward* devido a sua característica de o sinal se propagar somente em uma única direção. As RNA perceptron podem ser classificadas como *Single-Layer Perceptron* (SLP) e *Multi-Layer Perceptron* (MLP) a depender da presença de camadas intermediárias escondidas.

A SLP consiste de duas camadas, sendo uma de entrada e uma de saída da RNA. Tais redes são conhecidas por conseguirem realizar classificações de mais de duas classes de dados, entretanto as classes devem ser linearmente separáveis [8].

A rede MLP recebe este nome pois é construída de forma a possuir pelo menos uma camada escondida. Ou seja, uma camada intermediária de neurônios entre as camadas de entrada e saída. Esta camada escondida permite à RNA realizar a separação de duas ou mais classes que não sejam linearmente separáveis [8]. A Figura 2.5 apresenta um diagrama representando uma RNA MLP contendo n entradas, j neurônios na camada escondida e 1 na camada de saída.

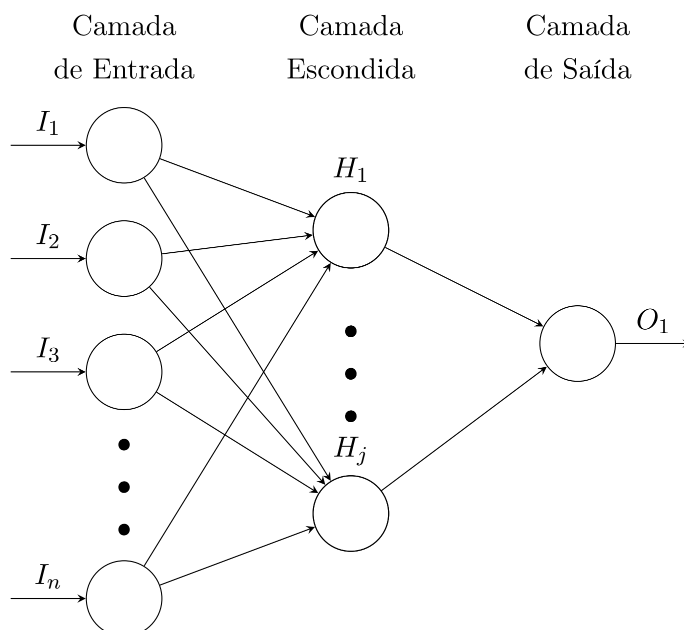


Figura 2.5: Representação de uma rede neural artificial perceptron com uma camada escondida

2.2.4 Redes recorrentes

A classificação de uma RNA como rede recorrente é na verdade bastante ampla, uma vez que a recorrência pode se dar de diversas formas. As recorrências podem se dar da camada de saída da rede para a camada de entrada como acontece nas redes NARX [8], ou da saída de uma camada para a entrada dela mesma como nas redes de Elman [9]. Neste trabalho iremos trabalhar com as redes de Elman quando tratarmos de recorrências nas RNAs.

O ponto característico das redes de Elman é o fato de elas possuírem uma camada que recebe as

saídas dos neurônios da camada escondida e retroalimenta-a no próximo ciclo com o valor guardado. Esta camada é chamada de camada de contexto. Uma representação de uma RNA de Elman de 2 entradas, 4 neurônios na camada escondida e de contexto, e um nerônio de saída é apresentada na Figura 2.6.

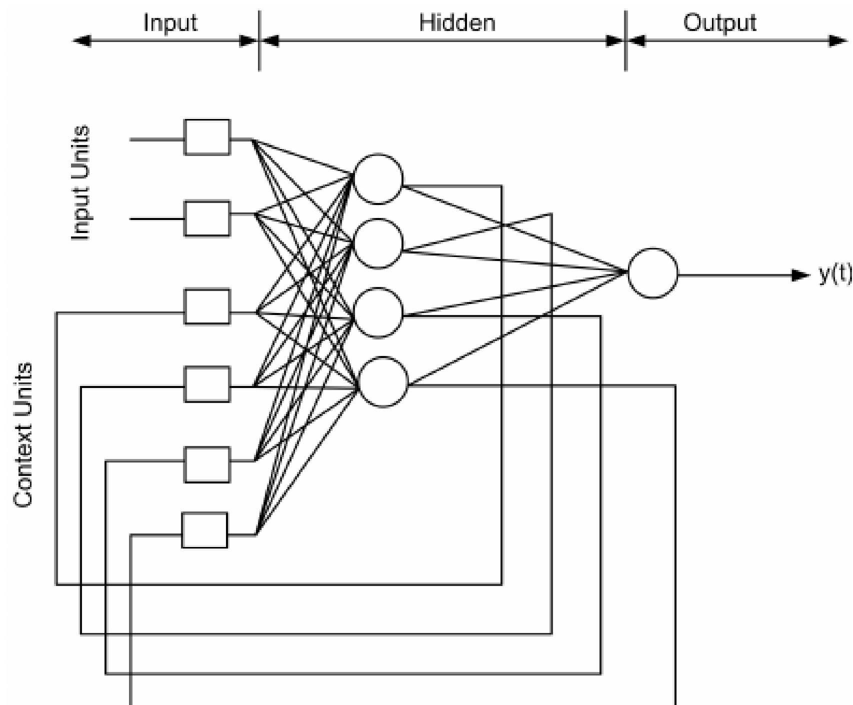


Figura 2.6: Representação de uma rede de Elman[2]

2.3 Algoritmos de treinamento

Após definida a estrutura da rede neural, faz-se necessária a obtenção do conjunto de pesos para cada neurônio a fim de mapear o conjunto de dados de entrada para o conjunto de dados de saída desejados da rede. Com essa finalidade são utilizados algoritmos de treinamento da rede. Estes algoritmos de aprendizagem podem ser classificados em três categorias.

- Aprendizagem não-supervisionada: Esta abordagem utiliza padrões intrínsecos aos dados de entrada para classificá-los. Desta forma, não há qualquer tipo de *feedback* quanto ao erro de saída do sistema. Algoritmos de clusterização, como o *k-means*[10], são exemplos de algoritmos de aprendizagem não-supervisionada.

- Aprendizagem Supervisionada: Dois conjuntos de dados são fornecidos para o treinamento, sendo eles um conjunto de dados de entrada (*Inputs*) e um conjunto de dados de saída desejada (*Target*). Deste modo, utiliza-se o conjunto de dados de entrada para obter os dados de saída da RNA, que são então comparados aos valores desejados. O erro entre a saída real e a desejada é utilizado para a alteração do conjunto de pesos da RNA.

- Aprendizagem por reforço: Similar à aprendizagem supervisionada, entretanto utiliza-se valores binários indicando se a saída da rede obteve o valor desejado ou não.

O algoritmo de Levenberg-Marquardt [11] é bastante conhecido em aplicações de treinamento de redes neurais, sendo implementado utilizando *error backpropagation*. Fato este que faz com este seja um algoritmo de aprendizagem supervisionada. Algoritmos meta-heurísticos como algoritmos evolutivos ou os bio-inspirados como o *particle swarm optimization* [12][13] também foram extensivamente pesquisados para busca do conjunto de pesos de redes neurais.

2.4 FPGA

Field Programmable Gate Array (FPGA) é um dispositivo que consiste de uma matriz de células lógicas interconectadas através de *switches* [3]. As células são programáveis e capazes de executar simples funções lógicas e os *switches* são reconfiguráveis de forma a realizar o roteamento dos sinais entre estas células. Após definidas as funções para cada célula lógica e as conexões dos *switches*, há a descarga desta configuração para o chip FPGA através de um cabo de comunicação entre o sistema no qual foi realizado o projeto e o dispositivo FPGA. O dispositivo é considerado *Field Programmable*, ou programável em campo, pois a etapa de síntese do circuito pode ser realizada pelo usuário no local onde o circuito será utilizado e não previamente pelo fabricante. A Figura 2.7 apresenta uma representação simples da configuração descrita.

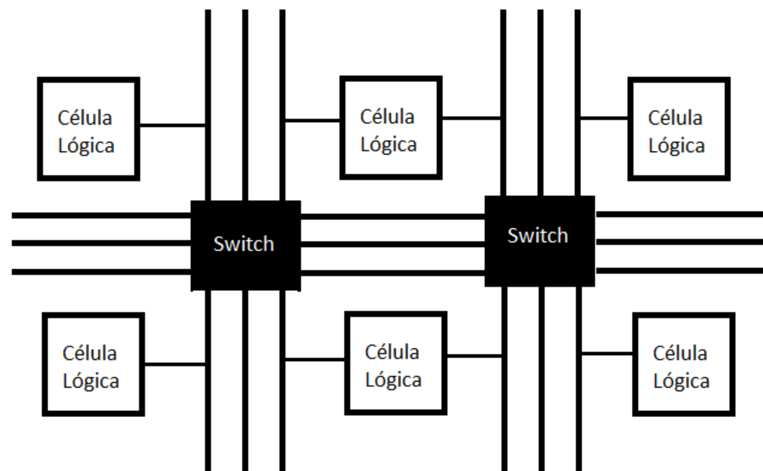
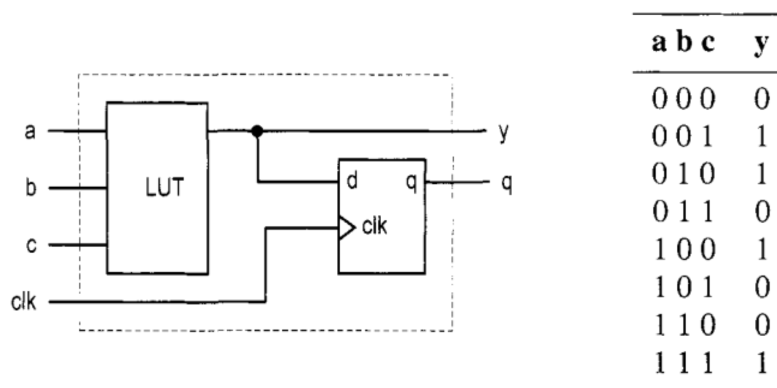


Figura 2.7: Representação simplificada de uma FPGA

2.4.1 Implementação das células lógicas

As células lógicas são implementadas utilizando circuitos combinacionais reconfiguráveis, geralmente baseados em *look-up tables* (LUT). A saída do bloco pode, então, ser a saída da LUT ou a saída de um flip-flop tipo D que recebe a saída da LUT como entrada. A utilização de um flip-flop é necessária caso se faça necessária a implementação de circuitos sequenciais síncronos.

A Figura 2.8 apresenta como pode ser realizada a implementação de uma célula lógica simples baseada em uma LUT de 3 entradas com um exemplo de tabela verdade para a célula.



a b c	y
0 0 0	0
0 0 1	1
0 1 0	1
0 1 1	0
1 0 0	1
1 0 1	0
1 1 0	0
1 1 1	1

Figura 2.8: Representação simplificada de uma célula lógica[3]

2.4.2 Placa Altera DE2-115

A placa Terasic DE2-115 utilizada neste trabalho é um kit de desenvolvimento que, além do chip do dispositivo FPGA cyclone IV EP4CE115F29C7N, possui periféricos que facilitam a comunicação do chip com dispositivos externos. Exemplos destes periféricos são: Entrada USB blaster, socket para cartão SD, quatro botões, 18 chaves deslizantes, 18 LEDs vermelhos, 9 LEDs verdes, conversor digital-analógico para VGA, transmissor-receptor RS-232 com conector de 9 pinos, 8 displays de 7 segmentos e módulo LCD 16x2. A Figura 2.9 apresenta um diagrama com as conexões entre os periféricos e o chip FPGA da placa de desenvolvimento DE2-115.

Para a implementação de circuitos no kit DE2-115 o usuário deve utilizar o software Quartus-II ou Quartus Prime produzido pela Intel[14]

2.5 Protocolo de comunicação

Para facilitar o uso, pelo usuário, do sistema de rede neural embarcada em FPGA, é utilizado um sistema de comunicação serial para que um computador pessoal possa enviar dados a serem processados e receber os resultados da rede neural embarcada. Nas sub-seções que seguem são apresentados o dispositivo e o protocolo utilizado para a criação do sistema de comunicação.

2.5.1 Recommended Standard 232 - RS232

Recommended Standard 232 (RS232) é um protocolo de comunicação que define características elétricas, funcionais e mecânicas do canal de comunicação para garantir a qualidade do envio de dados entre dispositivos através de portas seriais.

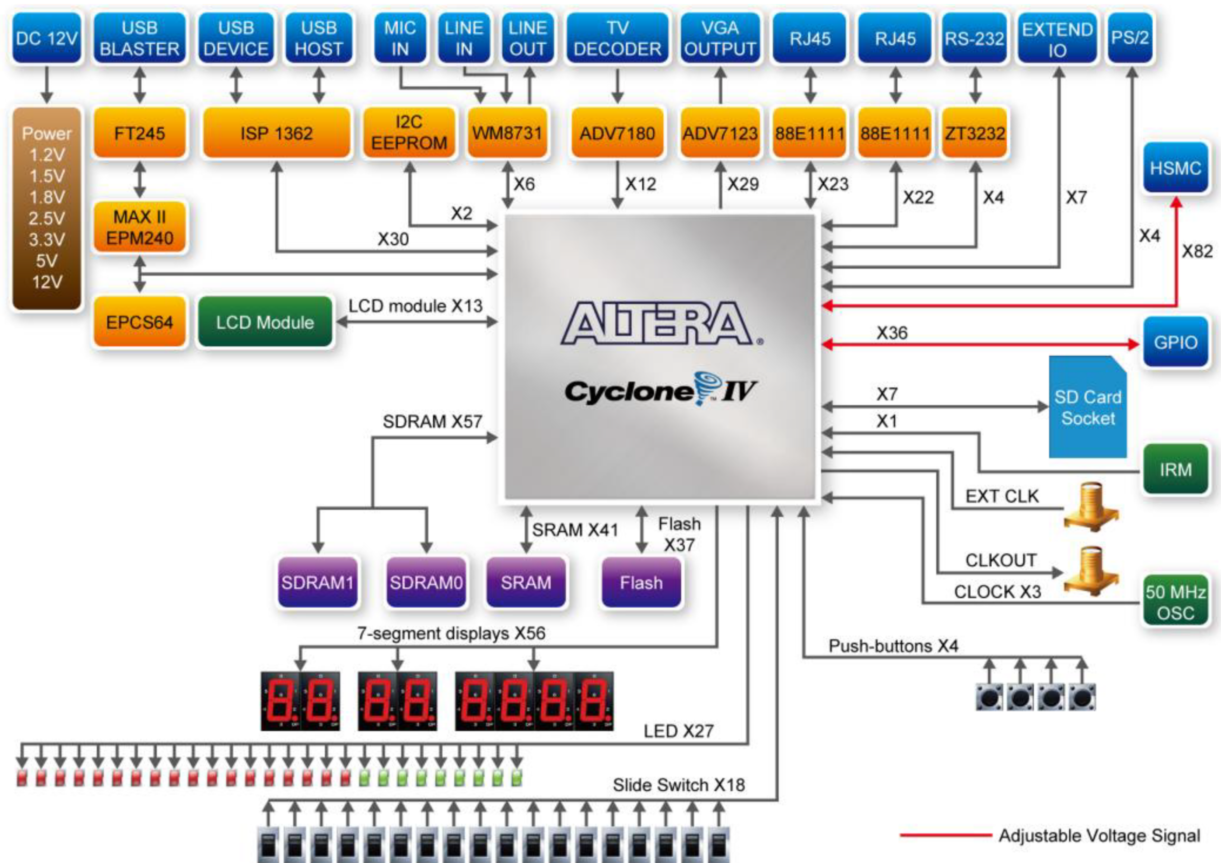


Figura 2.9: Diagrama da placa Terasic DE2-115[4]

As características elétricas definem níveis de tensão, taxa de variação (*slew rate*) máxima do sinal na saída do dispositivo de comunicação, taxa máxima de transmissão de dados e impedância da interface que conecta os dispositivos [5]. Seus respectivos valores são:

- Nível de tensão alto: entre +5V e +15V
- Nível de tensão baixo: entre -5V e -15V
- Taxa de variação máxima do sinal: 30V/μs
- Taxa de transmissão de dados máxima: 20 Kbits/s
- Impedância máxima: 2500pF

As características funcionais definem as funções para os sinais utilizados na comunicação, que são divididos em quatro categorias: comum, dados, controle e temporização. Tais sinais são apresentados na Tabela 2.2. Note que não se faz necessária a utilização de todos os sinais para a configuração de um canal de comunicação RS232. De fato, pode-se realizar tal tarefa utilizando somente dois dos sinais apresentados caso não se faça necessária a utilização de protocolo de *handshake*; sendo eles os sinais TD e RD, enquanto que para a definição da comunicação com

handshake são necessários somente quatro dos sinais apresentados na Tabela 2.2, sendo eles os sinais TD, RD, RTS e CTS [5].

Tabela 2.2: Sinais definidos pelo protocolo RS232

Nome do sinal	Tipo do sinal
Signal Common	Common
Transmitted Data (TD)	Data
Received Data (RD)	
Request to Send (RTS)	Control
Clear to Send (CTS)	
DCE ready (DSR)	
DTE ready (DTR)	
Ring Indicator (RI)	
Received Line Signal Detector (DCD)	
Signal Quality Detector	
Data Signal Rate Detector from DTE	
Data Signal Rate Detector from DCE	
Ready for Receiving	
Remote Loopback	
Local Loopback	
Test Mode	
Transmitter Signal Element Timing from DCE	Timing
Receiver Signa Element Timing from DCE	
Secondary Transmitted Data	Data
Secondary Received Data	
Secondary Request to Send	Control
Secondary Clear to Send	
Secondary Received Line Signal Detector	

As características mecânicas definem os conectores e pinos a serem utilizados para realização do protocolo RS232. A Figura 2.10 apresenta o conector de 9 pinos e as conexões que este realiza junto ao chip RS232 (ZT3232LEEY) do kit de desenvolvimento DE2-115.

2.5.2 Universal Asynchronous Receiver-Transmitter - UART

Como visto, o protocolo RS232 descreve os aspectos elétricos, mecânicos e funcionais para a correta transmissão de um bit através dos sinais RD e TD apresentados na tabela 2.2. Precisamos, no entanto, de enviar cadeias de bits que representam valores binários. Para esta tarefa utilizamos o dispositivo *Universal Asynchronous Receiver-Transmitter* (UART). O UART é utilizado para a conversão paralela/serial dos bits de dados transmitidos, buferização do dado a ser recebido/enviado, interface entre o dispositivo e a linha de comunicação e controle da sequência de dados enviados/recebidos [15]. Normalmente é utilizado para transmissão de 1 byte de dados.

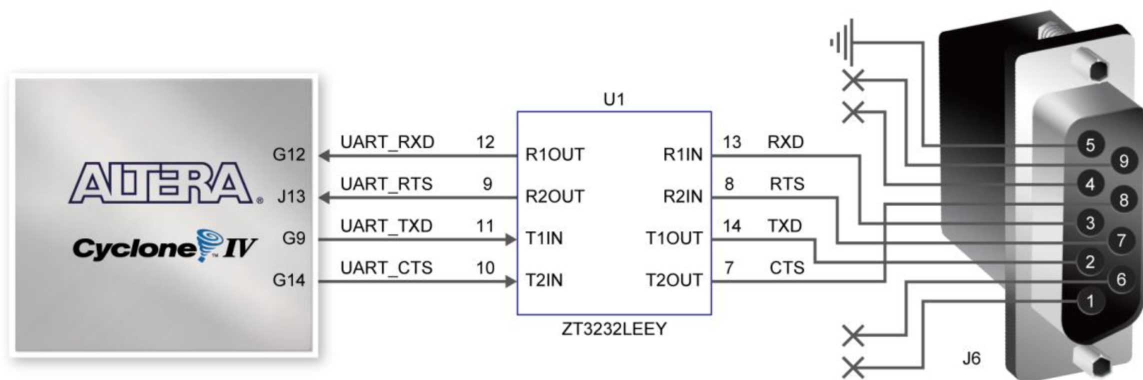


Figura 2.10: Conectores com 9 e 25 pinos para comunicação serial RS232[5]

Além dos oito bits de dados, são incluídos bits de controle para aumentar a qualidade da transmissão. Os bits de controle são: Um bit de início, um ou dois bits de final da transmissão (ou de parada) e um bit de paridade opcional. O arranjo destes é apresentado na Figura 2.11.

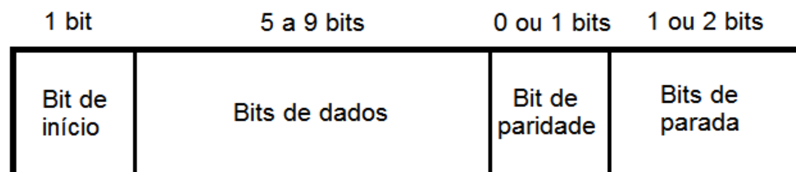


Figura 2.11: Sequência de bits para transmissão de dados UART

Sabe-se que os dados são interpretados de forma paralela. Para a transmissão, no entanto, deve-se ter os bits de forma sequencial. Por este motivo utiliza-se um *shift register* que faça um deslocamento a cada bit transmitido de forma a passar por e enviar todos os bits de forma sequencial.

Neste capítulo foram apresentados os conceitos básicos utilizados para o desenvolvimento do sistema proposto que será detalhado no próximo capítulo.

Capítulo 3

Sistema Proposto

Após apresentados os modelos das redes a serem implementadas e seus funcionamentos, deve-se definir os métodos para implementar as redes neurais em hardware através de linguagem de descrição de hardware Verilog. Neste capítulo são apresentadas as propostas de implementações em hardware para as funções de ativação, neurônios, ligações sinápticas e processo de realimentação, no caso das redes recorrentes. Note que os circuitos implementados neste trabalho são em sua totalidade feitos baseados em aritmética de ponto flutuante de 32 bits padrão IEEE 754.

3.1 Módulos de propriedade intelectual utilizados

Em todos os módulos criados em linguagem Verilog foram utilizados módulos de propriedade intelectual (IP) da *Altera Corporation*. Todos os módulos utilizados são disponibilizados através do *software Quartus Prime Lite Edition* Versão 17.1. Nesta subseção iremos apresentar os módulos IP utilizados para operações aritméticas em ponto flutuante juntamente ao seu tempo de execução em ciclos de *clock* e recursos físicos utilizados para implementação na DE2-115.

Para a implementação das funções de ativação e das redes neurais discutidas mais adiante, foi necessária a utilização das operações em ponto flutuante de soma, subtração, multiplicação, divisão, arredondamento, conversão para representação de número inteiro, exponenciação e comparação. A Tabela 3.1 apresenta os nomes dos módulos IP no Quartus juntamente com a operação realizada e as informações de tempo de execução (em ciclos de *clock*) e recursos físicos (número de elementos lógicos, número de registradores, bits de memória e número de multiplicadores de 9 bits) para cada bloco IP. Note que os módulos IP apresentados foram configurados de forma a realizar a operação desejada no menor tempo de latência em ciclos de *clock* possível.

Tabela 3.1: Módulos de IP para operações de ponto flutuante.

Nome	Função	Latência em *clocks*	Recursos físicos			
			Elementos lógicos	Regs	Bits de memória	Mult. de 9 bits
ALTFP_ADD	Soma	7	417	244	39	0
ALTFP_SUB	Subtração	7	352	201	36	0
ALTFP_DIV	Divisão	6	298	196	4624	16
ALTFP_MULT	Multiplicação	5	228	204	0	7
ALTFP_EXP	Exponenciação	17	1160	478	260	31
ALTFP_CONVERT	Conversão	6	423	252	0	0
ALTFP_COMPARE	Comparação	3	98	27	0	0

3.2 Função de ativação

A implementação da função de ativação em hardware é um grande desafio, pois a depender da aplicação da rede neural, aproximações podem levar a falhas de processamento na rede neural, levando a resultados errôneos. Por outro lado, não desejamos utilizar muitos recursos de hardware da FPGA para implementar uma função de ativação. Desta forma temos que encontrar o ponto médio aceitável entre erros de aproximação e espaço gasto em hardware aceitáveis para cada rede neural artificial e sua aplicação específica. Para fins de comparação, três possíveis implementações são apresentadas nesta seção. As funções de ativação a serem elaboradas em hardware neste trabalho serão as funções sigmoides unipolar e bipolar apresentadas na Seção 2.2.2.

3.2.1 Implementação por aproximação polinomial

A aproximação polinomial consiste em dividir a função de ativação em intervalos, sendo cada intervalo aproximado por um polinômio.

Para implementações da sigmoide unipolar, utilizando aproximações com polinômios de segundo grau, pode-se obter um módulo em hardware com erros entre a avaliação da sigmoide no MATLAB e no hardware da ordem de 10^{-3}, como apresentado em [16]. Neste trabalho, iremos aproximar a função de ativação em três intervalos, sendo eles $(-\infty, a)$, $[a, b]$ e (b, ∞). Os intervalos que seguem ao infinito positivo e infinito negativo são aproximados pelos valores das assíntotas da função de ativação e o intervalo central de a a b é aproximado por uma função polinomial. Utilizaremos, no entanto, uma manipulação para aumentar a precisão da aproximação polinomial no intervalo intermediário entre a e b. Sabemos que as funções sigmoide logística e tangente hiperbólica possuem as propriedades apresentadas nas Equações 3.1 e 3.2, respectivamente.

$$\phi_{unipolar}(x) + \phi_{unipolar}(-x) = 1 \implies \phi_{unipolar}(-x) = 1 - \phi_{unipolar}(x), \tag{3.1}$$

$$\phi_{bipolar}(x) = -\phi_{bipolar}(-x) \implies \phi_{bipolar}(-x) = -\phi_{bipolar}(x). \tag{3.2}$$

Percebemos, então, que se tivermos $\phi(x)$ é possível obter $\phi(-x)$ utilizando operações matemáticas simples como subtração e inversão de sinal. Lembre que a inversão de sinal para números em notação de ponto flutuante consiste em inverter o bit mais significativo, o que implica pouca utilização de hardware para a operação. Deste modo, podemos realizar a aproximação polinomial somente no intervalo $[0, b]$ e para um valor x positivo encontramos facilmente o valor da função para $-x$. Note que neste caso o valor de $a = -b$. Esta abordagem é interessante quando fizermos aproximações utilizando polinômios de grau maior do que 1.

Iremos definir uma função $\widetilde{\phi}(x)$ que aproxima a função de ativação $\phi(x)$ por intervalos de acordo com

$$\widetilde{\phi}(x) = \begin{cases} P(x) & x \in [0, b] \\ 1 & x \in (b, 8) \end{cases}, \tag{3.3}$$

sendo $P(x)$ o polinômio que melhor aproxima a função de ativação no intervalo $[0, b]$. Note que o intervalo (b, ∞) foi limitado em $(b, 8)$ de forma a possibilitar o cálculo do erro de aproximação entre a função aproximada $\widetilde{\phi}(x)$ e a função de ativação $\phi(x)$. Precisamos, então, encontrar o valor de b e o polinômio $P(x)$ que minimizem

$$\epsilon = \min_{b, P(x)} \{[\widetilde{\phi} - \phi(x)]^2\}. \tag{3.4}$$

Para a determinação do polinômio $P(x)$. Neste trabalho será utilizada a equação de mínimos quadrados em batelada apresentada em [17] dada por

$$\Theta = [X^T X]^{-1} X^T y. \tag{3.5}$$

Note que utilizamos a notação $P(x) = \theta_1 + \theta_2 x + \theta_3 x^2 + \cdots + \theta_k x^{k-1}$ para representar o polinômio. Vamos, então, definir os termos que compõem a Equação 3.5. Iniciamos com o vetor coluna Θ que contém os coeficientes do polinômio na forma

$$\Theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_k \end{bmatrix}, \tag{3.6}$$

com $k - 1$ sendo o grau do polinômio.

O vetor coluna y possui os valores da função de ativação $\phi(x)$, como em

$$y = \begin{bmatrix} \phi(x(1)) \\ \phi(x(2)) \\ \vdots \\ \phi(x(n)) \end{bmatrix}. \tag{3.7}$$

Note que os valores $x(i)$ são amostras de x no intervalo $[0, b]$ e n é o número de amostras que temos neste intervalo.

O único elemento que ainda deve ser apresentado é a matriz X que depende do grau do polinômio que se queira aproximar. A equação

$$X = \begin{cases} \begin{bmatrix} 1 & 1 & \dots & 1 \\ x(1) & x(2) & \dots & x(n) \end{bmatrix}^T & P(x) \ de \ grau \ 1 \\ \\ \begin{bmatrix} 1 & 1 & \dots & 1 \\ x(1) & x(2) & \dots & x(n) \\ x^2(1) & x^2(2) & \dots & x^2(n) \end{bmatrix}^T & P(x) \ de \ grau \ 2 \\ \vdots \\ \begin{bmatrix} 1 & 1 & \dots & 1 \\ x(1) & x(2) & \dots & x(n) \\ x^2(1) & x^2(2) & \dots & x^2(n) \\ \vdots & \vdots & \ddots & \vdots \\ x^k(1) & x^k(2) & \dots & x^k(n) \end{bmatrix}^T & P(x) \ de \ grau \ k \end{cases} \tag{3.8}$$

apresenta como é formada a matriz X para a determinação do polinômio $P(x)$.

Agora que temos o polinômio $P(x)$, vamos definir como encontrar o valor de b. Iremos utilizar um algoritmo de força bruta, iterando o valor de b de $0,5$ a 8 em intervalos de $0,001$ e calculando o erro médio quadrático (MSE) como em

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (\widetilde{\phi}(x(i)) - \phi(x(i)))^2 \tag{3.9}$$

para cada intervalo. Guardamos o valor de b que tenha o menor erro médio quadrático e o polinômio associado a ele para implementar a função de ativação aproximada $\widetilde{\phi}(\cdot)$. Os valores obtidos para os intervalos e polinômios para as funções sigmoide logística e tangente hiperbólica em aproximações com polinômios de graus 1, 2 e 3 são apresentados na tabela 3.2.

Tabela 3.2: Intervalos e polinômios para as aproximações polinomiais das funções sigmoide logística e tangente hiperbólica.

Função de ativação		θ_1	θ_2	θ_3	θ_4	a	b
Sigmoide logística	Grau 1	0,5000	0,1923	-	-	-2,6000	2,6000
	Grau 2	0,5052	0,2565	-0,0350	-	-4,1230	4,1230
	Grau 3	0,4914	0,2975	-0,0603	0,0042	-5,7560	5,7560
Tangente hiperbólica	Grau 1	$6,6 \times 10^{-16}$	0,7692	-	-	-1,3000	1,3000
	Grau 2	0,0105	1,0258	-0,2797	-	-2,0620	2,0620
	Grau 3	-0,0171	1,1901	-0,4826	0,0667	-2,8790	2,8790

Uma vez que temos o intervalo e os valores dos coeficientes Θ do polinômio, podemos criar um diagrama que realiza a função de ativação aproximada por polinômios. As Figuras 3.1 e 3.2 representam diagramas para a implementação por aproximação polinomial de segundo grau para as funções de ativação sigmoide logística e tangente hiperbólica, respectivamente. Nestas figuras os valores em hexadecimal $0x3F800000$, $0x00000000$ e $0xBF800000$ representam os números 1, 0 e -1 no formato IEEE 754, respectivamente. Neste trabalho serão realizadas aproximações com polinômios de grau 1, 2 e 3.

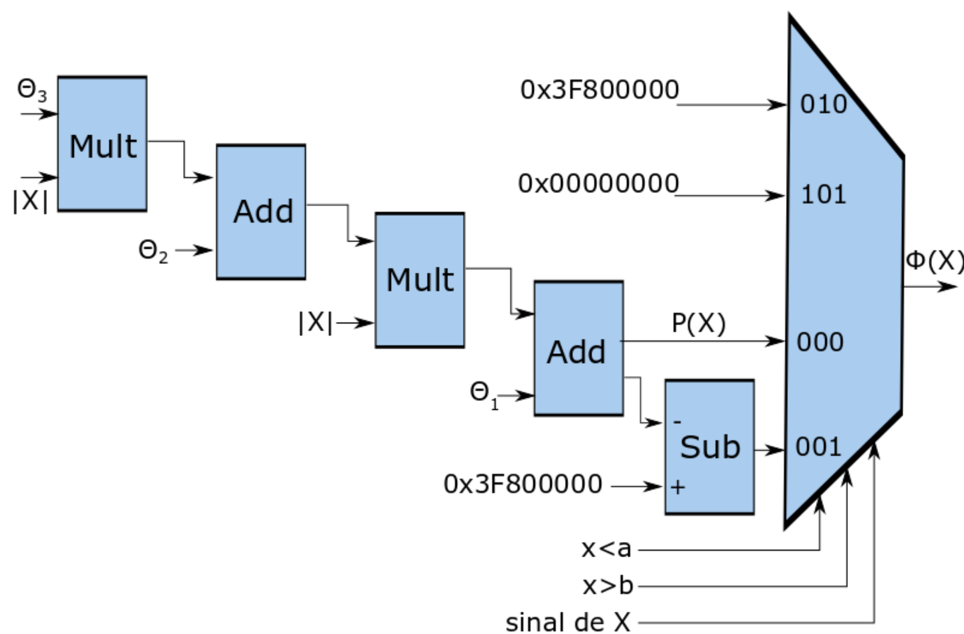


Figura 3.1: Diagrama para implementação de função de ativação sigmoide logística aproximada por polinômio de grau 2

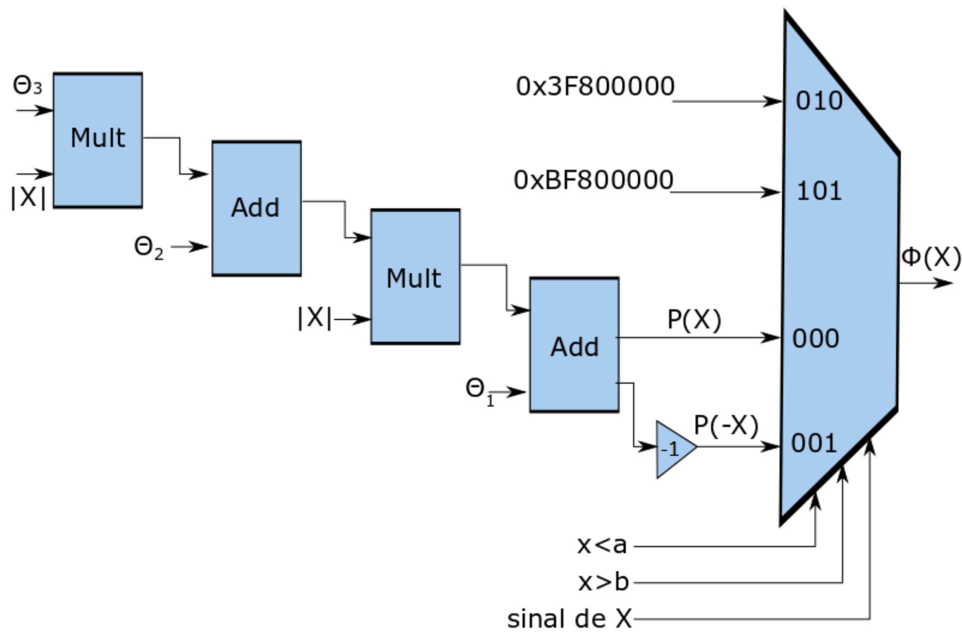


Figura 3.2: Diagrama para implementação de função de ativação tangente hiperbólica aproximada por polinômio de grau 2

Note que devemos utilizar o valor absoluto de x para calcular o valor polinomial aproximado da função de ativação, e no caso de x ser negativo o multiplexador irá alterar a saída para o valor correto. Nos casos em que o valor x estiver fora do intervalo $[a, b] = [-b, b]$, a saída da função de ativação aproximada será a assíntota da função de ativação.

Vamos, agora, encontrar o caminho crítico para calcular o número de ciclos de *clock* necessários para a propagação do valor de entrada ($|x|$) até a saída dos módulos das Figuras 3.1 e 3.2. Percebemos que o pior caminho é aquele em que o valor do polinômio aproximado ($P(x)$) deve ser calculado e ainda deve-se espelhar e encontrar o valor de $P(-x)$. Na Tabela 3.3 são apresentados os ciclos de *clock* necessários, utilizando os valores de latência em ciclos de *clock* apresentados na Tabela 3.1, para o processamento dos módulos de função de ativação aproximada por polinômios de grau 1, 2 e 3.

Tabela 3.3: Tempo de execução em ciclos de *clock* para as funções de ativação implementadas com aproximação polinomial

		Ciclos de*clock*
Sigmoide Logística	Grau 1	12
	Grau 2	31
	Grau 3	43
Tangente Hiperbólica	Grau 1	12
	Grau 2	24
	Grau 3	37

As Figuras II.1 e II.2, apresentadas no Anexo I, apresentam as formas de onda para as simulações das funções de ativação sigmoide logística e tangente hiperbólica, respectivamente. Nestas formas de onda os sinais `grau1`, `grau2` e `grau3` apresentam a saída das aproximações polinomiais de grau 1, 2 e 3 respectivamente. A análise das formas de onda também foram utilizadas para verificar que a latência apresentada na Tabela 3.3 é adequada.

3.2.2 Implementação utilizando módulo exponencial

Nesta subseção são apresentadas as implementações das funções de ativação sigmoide logística e tangente hiperbólica apresentadas nas Equações 2.5 e 2.6, respectivamente. Chamaremos, neste trabalho, os módulos sigmoide propostos aqui por implementações exatas uma vez que utilizam funções de exponenciação de propriedade intelectual da Altera. Note que mesmo que sejam chamadas de implementações exatas, ainda existirá um erro entre estes módulos e as implementações em MATLAB uma vez que aqui utilizamos aritmética em precisão simples (32 bits) enquanto o MATLAB implementa as funções em precisão dupla (64 bits).

Para a realização da sigmoide logística ou unipolar utilizamos um módulo exponencial, um módulo de soma, um módulo de divisão e um inversor para inverter o bit de sinal do número em ponto flutuante IEEE 754. A Figura 3.3 apresenta o diagrama de blocos da implementação.

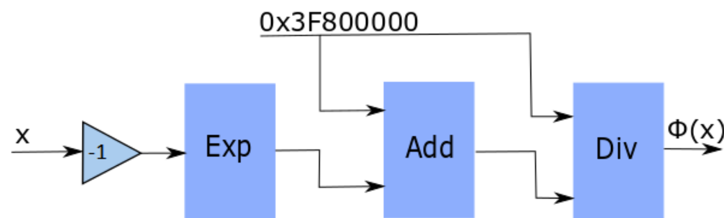


Figura 3.3: Módulo para implementação exata da função de ativação sigmoide logística

Podemos implementar a função sigmoide bipolar utilizando dois módulos IP exponenciais, um módulo IP de soma, um módulo IP de subtração e um módulo IP de divisão, como apresentado na Figura 3.4.

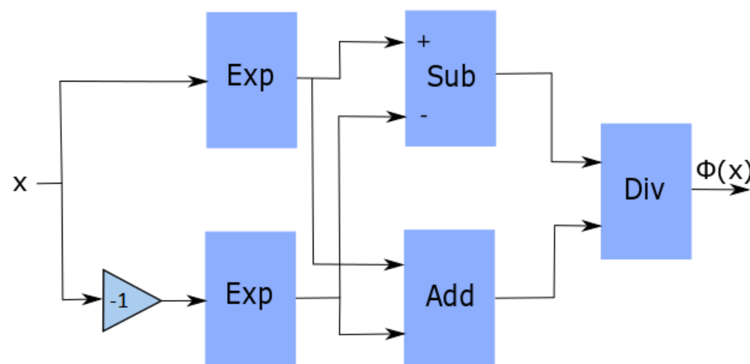


Figura 3.4: Módulo para implementação exata da função de ativação tangente hiperbólica

Observando as Figuras 3.3 e 3.4 percebemos que o caminho crítico para a propagação do sinal de entrada até a saída do módulo é igual para as duas funções de ativação. Este caminho é formado por três módulos IP de ponto flutuante em sequência sendo eles o módulo de exponenciação, de soma e de divisão. São apresentados, na Tabela 3.4, a quantidade de ciclos de *clock* necessários para o correto funcionamento das funções de ativação. Os valores de latência em ciclos de *clock* utilizados para o cálculo são aqueles apresentados na Tabela 3.1. Note que as formas de onda apresentadas nas Figuras II.1 e II.2 também foram analisadas para garantir a adequação dos ciclos apresentados na Tabela 3.4. Nestas formas de onda o sinal `OUTPUT_exato` representa a saída dos módulos de função de ativação implementados com blocos IP de exponenciação.

Tabela 3.4: Tempo de execução em ciclos de *clock* para as funções de ativação implementadas com módulos IP de exponenciação

	Ciclos de*clock*
Sigmoide Logística	31
Tangente Hiperbólica	31

3.2.3 Implementação por aproximação em intervalos fixos

Suponha que temos um intervalo $[a, b)$ e queremos aproximar a função sigmoide. Digamos ainda, que tal aproximação seja feita por um número fixo de pontos equidistantes neste intervalo, ou seja, com intervalos iguais entre quaisquer dois pontos de amostragem. Aqui utilizamos N pontos para a aproximação. Desta forma podemos amostrar a função de acordo com o pseudocódigo:

Data: função de ativação $\phi(.)$
Result: função amostrada em intervalos regulares $f_amostrada$
initialization;
for *i de 0 a N-1* **do**
  $f_{amostrada}[i] \leftarrow \phi(a + (b - a)\frac{i}{N})$
end

Algorithm 1: Amostragem da função de ativação

como saída do pseudocódigo apresentado temos um vetor $f_amostrada$ no intervalo desejado. Para acessar o elemento correto do vetor para um valor x, basta fazer

$$i = round\Big(\frac{(x - a)}{(b - a)} \cdot N\Big), \tag{3.10}$$

então temos que

$$f_amostrada[i] \approx \phi(x). \tag{3.11}$$

Neste trabalho os valores de a, b e N foram escolhidos de forma a simplificar a Equação 3.10 e facilitar a implementação na FPGA. Escolhemos quatro valores para N de forma a verificar a relação entre custo de implementação e erros de aproximação, sendo que os resultados serão apresentados no Capítulo 4. Os valores escolhidos foram, então

- $a = -4$
- $b = 4$
- $N = 64, 128, 256, 512.$

Com estas escolhas é possível simplificar a Equação 3.10 em

$$i = \begin{cases} round\{(x+4) \times 8\} \implies round\{(x+4) \times 2^3\} & N = 64 \\ round\{(x+4) \times 16\} \implies round\{(x+4) \times 2^4\} & N = 128 \\ round\{(x+4) \times 32\} \implies round\{(x+4) \times 2^5\} & N = 256 \\ round\{(x+4) \times 64\} \implies round\{(x+4) \times 2^6\} & N = 512. \end{cases} \tag{3.12}$$

Temos, então, que para a implementação do módulo da função de ativação aproximada em intervalos fixos faz-se necessário o uso de dois módulos IP para operações de ponto flutuante. Um módulo somador para adicionar o valor `0x40800000` (número $4,0$ em notação de ponto flutuante) ao x e um módulo de conversão para realizar o arredondamento indicado por *round* na Equação 3.12. Uma das vantagens de se utilizar o intervalo escolhido é o de que não se faz necessária a utilização de um módulo IP de multiplicação em ponto flutuante, pois a multiplicação de um número em ponto flutuante qualquer por uma potência 2^n pode ser realizada pela soma em complemento de dois do expoente do número em ponto flutuante por n.

Um ponto importante a salientar é o fato de que devemos determinar o tratamento de valores de entrada que estejam fora do intervalo de aproximação. A abordagem tomada aqui foi a de, caso o valor de entrada da função de ativação esteja fora do intervalo de aproximação, determinar a saída da função de acordo com a Equação 3.13 no caso da função sigmoide logística e Equação 3.14 para a função tangente hiperbólica.

$$\phi_{unipolar}(x) = \begin{cases} 0 & x < a \\ 1 & x \geq b \end{cases} \tag{3.13}$$

$$\phi_{bipolar}(x) = \begin{cases} -1 & x < a \\ 1 & x \geq b \end{cases} \tag{3.14}$$

As Figuras 3.5 e 3.6 apresentam diagramas para a implementação das funções de ativação sigmoide logística e tangente hiperbólica, respectivamente, com aproximação por amostragem em intervalos fixos.

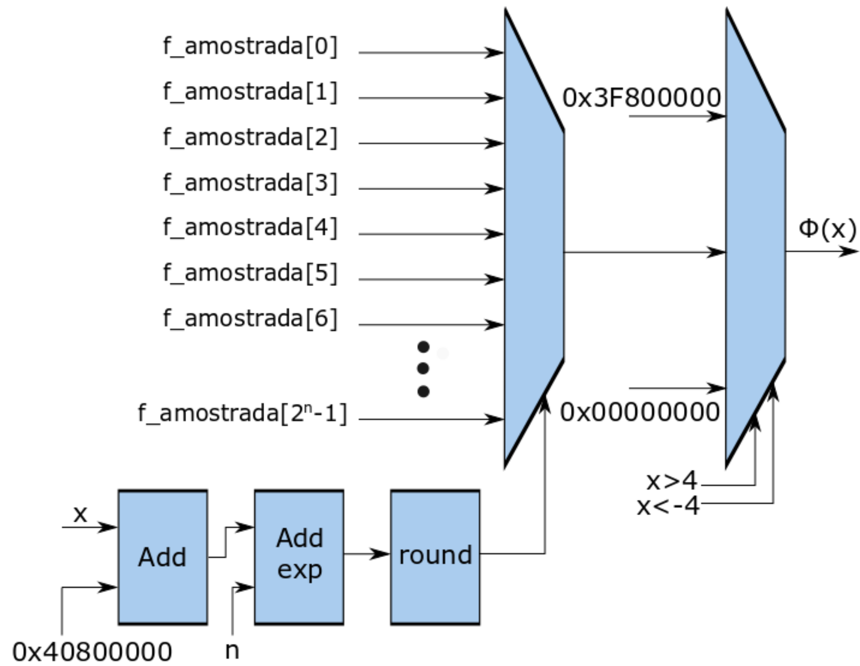


Figura 3.5: Diagrama para implementação da função de ativação sigmoide logística por amostragem em intervalos fixos

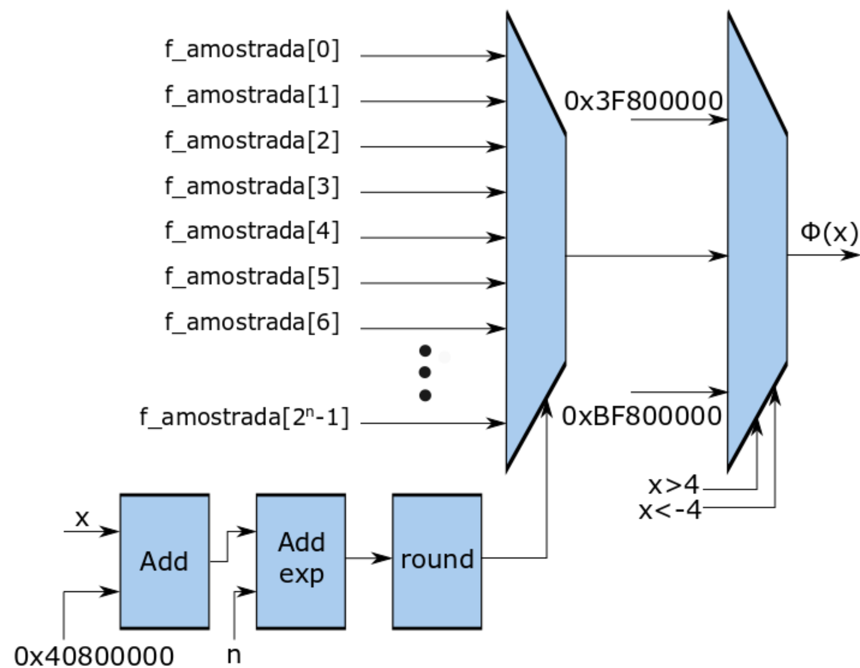


Figura 3.6: Diagrama para implementação da função de ativação tangente hiperbólica por amostragem em intervalos fixos

Nas figuras o bloco `Add exp` indica a soma em complemento de 2 dos oito bits do expoente do número em ponto flutuante na saída do bloco `Add` com o valor n. O bloco *round* indica conversão de um número em ponto flutuante para um número inteiro utilizando arredondamento. Analisando os diagramas podemos perceber que o caminho crítico é aquele formado pelos blocos `Add`, `Add exp` e `round`, uma vez que o restante do diagrama é formado por lógicas combinacionais. A latência em ciclos de *clock* necessária para o correto funcionamento das funções de aproximação descritas são apresentadas na Tabela 3.5.

Tabela 3.5: Tempo de execução em ciclos de *clock* para as funções de ativação com aproximação por amostragem em intervalos fixos

	Ciclos de *clock*
Sigmoide Logística	13
Tangente Hiperbólica	13

As formas de onda apresentadas nas Figuras II.1 e II.2 também foram analisadas para garantir a adequação dos valores de ciclos apresentados. Nestas formas de onda os sinais `OUTPUT_64`, `OUTPUT_128`, `OUTPUT_256` e `OUTPUT_512` representam as saídas dos módulos de função de ativação aproximada por intervalos fixos de 64, 128, 256 e 512 amostras, respectivamente.

3.3 Neurônio

Para a criação do módulo de um neurônio são utilizados os módulos IP da Intel de soma e multiplicação de ponto flutuante, assim como uma das implementações apresentadas na Seção 3.2 para a função de ativação. Como discutido na Subseção 2.2.1, o neurônio é constituído de três elementos básicos: Um conjunto de pesos de ligações ponderadas, ou sinapses, que multiplicam cada entrada do neurônio, um somador que realiza a soma de todas as entradas já ponderadas pelos seus respectivos pesos sinápticos mais o *bias* formando o campo local induzido do neurônio, e uma função de ativação capaz de mapear o campo local induzido à valores limitados de saída do neurônio. Desta forma, ao criarmos a representação em hardware para o neurônio também podemos dividi-la em três partes. A primeira é um conjunto de módulos multiplicadores que realizam a ponderação das entradas do neurônio, a segunda é um conjunto de módulos somadores que realiza a soma das entradas ponderadas mais o *bias* e a terceira é a utilização de um dos módulos de função de ativação discutidos na Seção 3.2. A Figura 3.7 apresenta o diagrama em blocos de um neurônio com 3 ($[X_0, X_1, X_2]$) valores de entradas.

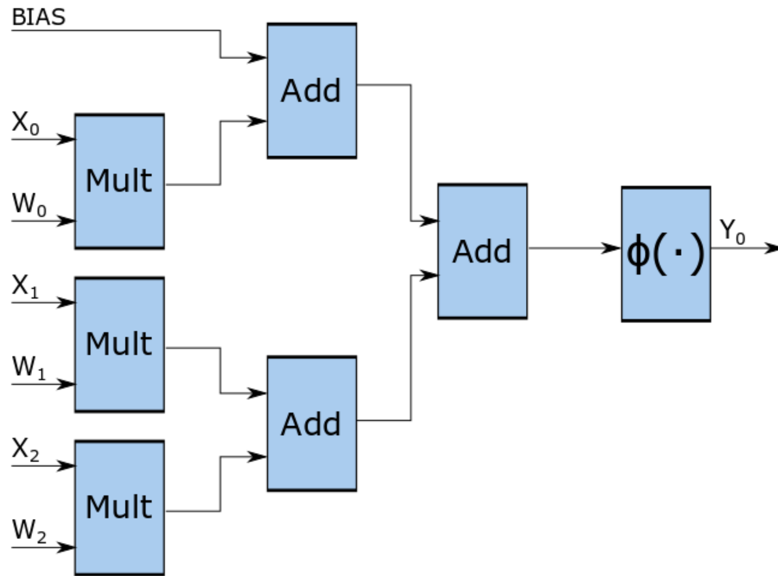


Figura 3.7: Diagrama de um neurônio com 3 entradas

É importante enfatizar que as redes estudadas neste trabalho são todas densamente conectadas, ou seja, o neurônio de uma camada se conecta a todos os neurônios da camada seguinte. Desta forma podemos criar somente um módulo de neurônios em Verilog para cada camada da rede neural, uma vez que os neurônios de uma mesma camada possuem uma mesma configuração interna.

Ao embarcarmos a rede neural em FPGA, devemos saber a latência em ciclos de *clock* para o processamento desta. Esta latência pode ser calculada como a soma dos ciclos de *clock* necessários ao processamento dos neurônios de cada camada. Pela estrutura do neurônio como apresentado na Figura 3.7, podemos dividir ele em três partes: os multiplicadores em paralelo, camadas de módulos somadores e uma função de ativação. Calcularemos, então, a latência para cada uma destas partes utilizando os valores apresentados na Tabela 3.1 para os módulos IP de ponto flutuante e as Tabelas 3.3 , 3.4 e 3.5 para as funções de ativação.

Os multiplicadores, como estão dispostos em paralelo, possuem latência de um único módulo multiplicador de 5 ciclos de *clock*. Definindo n como o número de entradas e l como o número de camadas de módulos de soma, iremos utilizar a expressão

$$l = ceil(log_2(n+1)) \tag{3.15}$$

para encontrar o tempo de execução dos módulos somadores, que será $7 \times l$. Note que devemos somar o valor 1 ao n pois além das entradas da rede iremos contar o *bias*. Finalmente, o tempo de processamento das funções de ativação serão os apresentados nas tabelas mencionadas. Temos, desta forma, que o tempo de processamento de um neurônio T_n, em ciclos de *clock*, pode ser calculado por

$$T_n = 5 + 7ceil(log_2(n)) + T_\phi \tag{3.16}$$

com T_ϕ sendo a latência em *clocks* para processamento da função de ativação. A latência total da

rede L_{rede} será

$$L_{rede} = \sum_{i=1}^{c} T_i \tag{3.17}$$

com c sendo o número de camadas na RNA e T_i a latência de cada camada.

3.4 Realimentações

A realimentação das redes neurais artificiais recorrentes neste trabalho se dão através da camada de contexto, que armazena os valores das saídas dos neurônios das camadas escondidas e realimenta estes valores à entrada dos neurônios das respectivas camadas escondidas no próximo ciclo de processamento da rede. Para implementarmos a camada de contexto, esperamos a quantidade de ciclos de *clock* necessários para um ciclo de processamento da rede neural e então atualizamos registradores de 32 bits que armazenam a saída dos neurônios da camada escondida. Este grupo de registradores de 32 bits forma a camada de contexto e é responsável por realimentar os valores armazenados das saídas dos neurônios para as suas entradas no próximo ciclo de processamento da rede. A listagem 3.1 apresenta um trecho de código Verilog mostrando como foi implementada a atualização dos registradores da camada de contexto.

Listing 3.1: Trecho de código para a atualização dos registradores da camada de contexto

```
1  always @(posedge CLOCK_50)
2          begin
3          if(reset == 1'b1) begin
4                  R1[0] <= 32'h00000000;
5                  R1[1] <= 32'h00000000;
6                  R1[2] <= 32'h00000000;
7                  R1[3] <= 32'h00000000;
8          end
9          else if(data_ready == 1'b1) begin
10                 R1[0] <= wZ1_0;
11                 R1[1] <= wZ1_1;
12                 R1[2] <= wZ1_2;
13                 R1[3] <= wZ1_3;
14         end
15 end
```

Note que no exemplo apresentado, a camada de contexto e a camada escondida possuem quatro neurônios. `R1[0]`, `R1[1]`, `R1[2]` e `R1[3]` são os registradores de 32 bits que formam a camada de contexto, `wZ1_0`, `wZ1_1`, `wZ1_2` e `wZ1_3` são os resultados apresentados na saída de cada um dos neurônios da camada escondida e `data_ready` é o sinal que indica fim de um ciclo de processamento da rede. O sinal de `reset` é utilizado para zerar a camada de contexto para a realização de novos testes na rede neural.

3.5 Rede Neural

Utilizando os modelos discutidos para os neurônios e para as realimentações, podemos então realizar a rede neural tanto perceptron quanto recorrente. Os pesos da rede para todos os neurônios serão fixos e inicializados no módulo Verilog da rede denominado `net.v`. Tais pesos serão então passados às instâncias criadas de cada neurônio em cada camada. Neste módulo também será feita a conexão entre os neurônios e no caso da rede recorrente serão feitas a atualização da camada de contexto e suas conexões de realimentação para a camada escondida. A Listagem 3.2 apresenta o módulo `net.v` para uma rede neural artificial recorrente com dois neurônios na camada escondida e um valor para entrada e para saída da rede, sendo eles $X0$ e $Z0$ respectivamente. `node1` é o módulo do neurônio da primeira camada, a camada escondida, enquanto `node2` é o módulo do neurônio da camada de saída. Os pesos e `bias` são definidos nas linhas de 12 a 22, a camada de contexto é apresentada nas linhas 57 a 67.

Listing 3.2: Módulo em Verilog de uma rede neural artificial recorrente

```
1  module net(
2      input[31:0] X0,
3      output[31:0] Z0,
4      input iclk, data_ready, reset
5  );
6
7  reg[31:0]  R1[1:0];
8
9  wire[31:0] wZ1_0, wZ1_1;
10 wire[31:0] wBias1[1:0], wP1_1[2:0], wP1_2[2:0], wBias2[0:0], wP2_1[1:0];
11
12 assign wBias1[0] = 32'hc149235a;//      camada:1 |neuronio:1
13 assign wP1_1[0] = 32'h4142fbcc;//       camada:1 |neuronio:1 |peso:1
14 assign wP1_1[1] = 32'h4149b9d9;//       camada:1 |neuronio:1 |peso:2
15 assign wP1_1[2] = 32'h41129300;//       camada:1 |neuronio:1 |peso:3
16 assign wBias1[1] = 32'hbf6bc2e1;//      camada:1 |neuronio:2
17 assign wP1_2[0] = 32'h40f67c90;//       camada:1 |neuronio:2 |peso:1
18 assign wP1_2[1] = 32'hc115482f;//       camada:1 |neuronio:2 |peso:2
19 assign wP1_2[2] = 32'h40f4dbf1;//       camada:1 |neuronio:2 |peso:3
20 assign wBias2[0] = 32'h3f6b542c;//      camada:2 |neuronio:1
21 assign wP2_1[0] = 32'h3dfcab1d;//       camada:2 |neuronio:1 |peso:1
22 assign wP2_1[1] = 32'hbf7ababa;//       camada:2 |neuronio:1 |peso:2
23
24 node1 N1_1(
25         .X0(X0),
26         .P0(wP1_1[0]),
27         .X1(R1[0]),
28         .P1(wP1_1[1]),
29         .X2(R1[1]),
30         .P2(wP1_1[2]),
31         .Bias(wBias1[0]),
```

```
32          .Z(wZ1_0),
33          .iclk(iclk)
34  );
35
36  node1 N1_2(
37          .X0(X0),
38          .P0(wP1_2[0]),
39          .X1(R1[0]),
40          .P1(wP1_2[1]),
41          .X2(R1[1]),
42          .P2(wP1_2[2]),
43          .Bias(wBias1[1]),
44          .Z(wZ1_1),
45          .iclk(iclk)
46  );
47  node2 N2_1(
48          .X0(wZ1_0),
49          .P0(wP2_1[0]),
50          .X1(wZ1_1),
51          .P1(wP2_1[1]),
52          .Bias(wBias2[0]),
53          .Z(Z0),
54          .iclk(iclk)
55  );
56
57  always @(posedge iclk)
58  begin
59  if(reset == 1'b1) begin
60          R1[0] <= 32'h00000000;
61          R1[1] <= 32'h00000000;
62  end
63  else if(data_ready == 1'b1) begin
64          R1[0] <= wZ1_0;
65          R1[1] <= wZ1_1;
66  end
67  end
68
69  endmodule
```

3.6 Ferramenta para geração automática do código Verilog a partir da rede em MATLAB

A criação de uma rede neural em Verilog tendo como base uma rede criada e treinada em MATLAB de forma manual exige tempo. Por este motivo uma ferramenta para conversão da rede MATLAB em código Verilog foi desenvolvida. Desta forma a validação de diversas redes com

variados números de camadas e neurônios em cada camada podem ser rapidamente realizadas. O *script* desenvolvido `gera_verilog.m`, em MATLAB, cria um módulo Verilog de neurônio para cada camada escondida e de saída e um módulo Verilog da rede neural onde são inicializados os pesos, instanciados os neurônios e as conexões são realizadas; para o caso da rede recorrente, os registradores de realimentação são criados e conectados. É importante notar que a rede em Verilog não realiza funções de preprocessamento dos dados, sendo necessário que o usuário execute o preprocessamento antes do envio dos dados à FPGA.

Uma rede neural em MATLAB é definida através de um objeto que contém informações quanto aos valores de pesos sinápticos, *bias*, tipos de funções de ativação de cada camada e diversos parâmetros de treinamento. Para caracterizar a rede neural e criar sua descrição em Verilog, no entanto, basta conhecer sua estrutura, seus pesos sinápticos, *bias* e funções de ativação de cada neurônio. Iremos chamar o objeto que define uma rede neural em MATLAB por `net`. Neste objeto, os pesos sinápticos são guardados nos campos `net.IW` e `net.LW`, que possuem estruturas similares. Ambos os campos são formados por uma matriz de células e cada célula possui uma matriz de pesos.

Dado o elemento `net.LW` definido por

$$net.LW = \begin{bmatrix} C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,1} & C_{3,2} & C_{3,3} \end{bmatrix} \tag{3.18}$$

sendo $C_{i,j}$ uma célula contendo uma matriz de pesos sinápticos. Temos então que a célula $C_{i,j}$ é aquela que possui o conjunto de pesos sinápticos que conectam a saída dos neurônios da camada j para a entrada dos neurônios da camada i. Note que para o caso onde $i = j$ estamos falando de realimentação da saída de uma camada para a entrada dela mesma. Cada célula $C_{i,j}$ configura as ligações entre os neurônios das camadas i e j de acordo com

$$net.LW\{i,j\} = C_{i,j} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \\ W_{3,1} & W_{3,2} \\ W_{4,1} & W_{4,2} \end{bmatrix} \tag{3.19}$$

Cada linha da matriz $C_{i,j}$ representa os pesos sinápticos que ligam cada um dos neurônios da camada j a um dos neurônios da camada i. Ou seja, $W_{a,b}$ em $C_{i,j}$ conecta a saída do neurônio b da camada j à entrada do neurônio a da camada i. A estrutura de *net.IW* é similar à apresentada para *net.LW* com a diferença que a camada 1 representada em *net.LW* se refere à primeira camada escondida, enquanto a camada 1 em *net.IW* se refere à camada de entrada da rede.

Após os pesos sinápticos vamos apresentar a estrutura que contém os valores de *bias* para todos os neurônios da rede. Tais valores ficam no campo *net.b* que é composto de um vetor de células onde cada elemento do vetor possui um vetor de valores de *bias*. Esta estrutura é mais simples do que a apresentada para os pesos da rede, uma vez que ela não indica conexão entre diferentes

neurônios. Sendo

$$net.b = \begin{bmatrix} C_1 \\ C_2 \\ C_3 \end{bmatrix} \tag{3.20}$$

onde cada célula C_i contém o conjunto de *bias* para os neurônios da camada i da rede. Caso tenhamos

$$C_i = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{3.21}$$

então b_j representa o *bias* do neurônio j na camada i.

Já sabemos como os pesos e os *bias* são apresentados no objeto da rede neural em MATLAB, falta agora verificar a função de ativação que a rede utiliza. Isto pode ser verificado através do campo $net.layers\{i\}.transferFcn$ que contém o nome da função de ativação utilizada na camada i. Alguns exemplos de funções de ativação que podem ser encontradas neste campo são

- 'logsig' que representa a função sigmoide logística
- 'tansig' representando a função tangente hiperbólica
- 'purelin' indicando função de ativação linear.

Com isto, todo o conhecimento a respeito da representação de uma rede neural em MATLAB necessária para a geração de uma ferramenta de conversão para Verilog foi apresentada. Vamos explicar o funcionamento da ferramenta a seguir.

Como já discutido, uma ferramenta que permita a conversão de uma rede MATLAB para código Verilog é proposta nesta subseção. As estruturas de redes que serão passíveis de conversão se restringem às redes perceptron e recorrentes desde que a recorrência se dê da saída de uma camada para a entrada dela mesma. A ferramenta desenvolvida em linguagem de programação MATLAB consiste de um *script* `generate_verilog.m` e três funções `generate_neuron.m`, `generate_net.m` e `count_clocks.m`.

O *script* `generate_verilog.m` é o código principal que utiliza as demais funções para gerar a representação da rede neural em Verilog. Para utilizar ele, basta carregar um objeto definindo uma rede neural no MATLAB com o nome de *net* e executar o *script*. Será pedido que o usuário escolha qual das implementações de funções de ativação dentre as apresentadas na Seção 2.2.2 deve ser utilizada. O próximo passo é a definição automática do número de entradas em cada neurônio para cada camada, o que nos permitirá criar os módulos dos neurônios. O passo seguinte é definir o número de neurônios em cada camada, o que nos permitirá criar o módulo da rede com as instanciações dos neurônios. Ambos os passos são obtidos utilizando as matrizes de pesos sinápticos. Uma vez que temos estes valores, o *script* irá chamar a função `generate_neuron.m` para cada camada da rede de forma a gerar um módulo Verilog de neurônio por camada. Após gerados os neurônios

a função `generate_net.m` é chamada para criar um módulo que instancie os neurônios de cada camada e realize a conexão entre eles. Se houverem realimentações, estas serão criadas e conectadas à rede neste módulo. Após a execução de `generate_net.m` já temos os códigos Verilog da rede desejada nos arquivos `net.v` que define a estrutura da rede e os arquivos `neuronio_camada_1.v`, `neuronio_camada_2.v`, ..., `neuronio_camada_i.v` no caso de a rede neural possuir i camadas. Há, no entanto, mais uma função a ser executada pelo *script*. A função `count_clocks.m` é responsável por contar a quantidade de ciclos de *clocks* mínima para a correta execução da rede. Esta estimativa leva em consideração a estrutura da rede e a forma de implementação da função de ativação escolhida pelo usuário.

3.7 Comunicação de dados entre DE2-115 e MATLAB

Até aqui vimos como é representada a RNA em Verilog que será embarcada no kit de desenvolvimento DE2-115. Veremos agora como a rede neural embarcada se comunica com o MATLAB.

Sabemos que o protocolo RS232 define os padrões para envio e recebimento de um bit de dado e que o protocolo UART padroniza a comunicação assíncrona de um byte de dados com alguns bits de controle adicionais. Neste trabalho todos os cálculos da rede são realizados em ponto flutuante de precisão simples (32 bits). Além disto, a rede neural pode possuir mais de um valor de entrada ou saída. Para lidar com estes problemas foi projetada uma máquina de estados finitos que irá controlar o número de bytes recebidos e enviados pelo kit DE2-115, o número de entradas e saídas da rede recebidos e enviados pelo kit DE2-115 e o tempo necessário para a rede neural realizar o processamento dos dados. A máquina de estados é apresentada na Figura 3.8 e pode ser separada em três partes principais A, B e C.

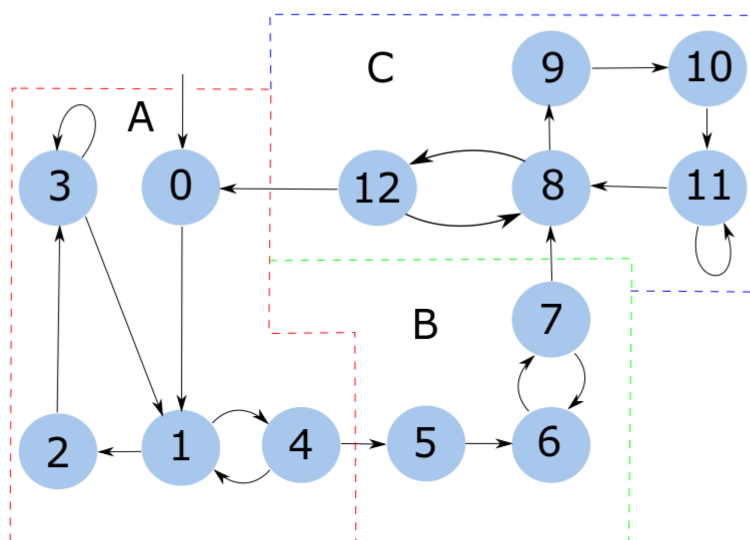


Figura 3.8: Máquina de estados para o recebimento, processamento e envio dos dados entre MATLAB e DE2-115

Começaremos a explicação pelo conjunto de estados representado por A. Este é o responsável

por receber os dados de entrada da rede. As variáveis de controle são `count_bytes`, `count_inputs` e `data_ready` responsáveis por contar o número de bytes lidos, entradas da rede lidas e disponibilidade de byte para leitura, respectivamente. A tarefa de cada estado neste conjunto é:

O estado 0 é responsável por zerar as variáveis de controle `count_bytes` e `ativa_NN` responsável por atualizar os registradores da camada de contexto.

O estado 1 é o responsável por verificar se foi realizada a leitura de 4 bytes consecutivos pelo `loop` $1-2-3$, formando assim uma das entradas da rede. No caso de ter lido uma entrada completa da rede, incrementa-se o valor de `count_inputs`, zera o valor de `count_bytes` e o próximo estado será o 4. Caso não tenha lido os 4 bytes e o bit de `data_ready` for 1, indicando um byte esperando para ser lido, o próximo estado será o 2; caso `data_ready` seja 0, esperamos até aparecer um byte a ser lido.

O estado 2 lê um byte, designa qual byte de qual entrada da rede receberá o valor lido e incrementa o valor de `count_bytes` em uma unidade. Após ler o byte e passar à entrada correspondente seguimos ao estado 3.

O estado 3 tem por única finalidade esperar o bit `data_ready` se tornar zero. Garantindo que o byte já lido não será lido novamente. No momento que `data_ready` voltar ao valor lógico zero, segue-se para o estado 1 novamente.

O estado 4 é responsável por controlar o número de entradas da rede já lido. Primeiramente ele zera o valor de `count_bytes`, uma vez que foram lidos 4 bytes. Caso `count_inputs` seja igual ao número de entradas da rede, este valor é zerado e o próximo estado é o 5. Caso não tenham sido lidos todos os valores para as entradas da rede, volta ao estado 1.

Foram apresentados os estados responsáveis por ler as entradas da rede. Agora será apresentado o conjunto de estados B, responsáveis por controlar o tempo de execução da rede neural. Os sinais de controle para este grupo são `count_cycles` e `n_cycles` que representam o número de ciclos desde que a rede começou a processar e o número de ciclos para o processamento completo da rede, respectivamente.

O estado 5 zera o valor de `count_cycles`.

O estado 6 incrementa o valor de `count_cycles` em uma unidade.

O estado 7 compara os valores de `count_cycles` e `n_cycles`. Caso sejam iguais passa-se ao estado 8, caso contrário retorna ao estado 6.

O conjunto de estados B foi apresentado e iremos agora explicar o conjunto de estados C, responsável pela transmissão dos resultados de processamento da rede neural. Este conjunto de estados possui os sinais de controle `TxStart`, `count_bytes`, `count_outputs`, `RsBusy` e `ativa_NN` responsáveis por iniciar a transmissão dos dados, contar o número de bytes e o número de saídas da rede enviados, indicar a disponibilidade do canal de comunicação e atualizar os registradores da camada de contexto caso a rede seja recorrente, respectivamente.

O estado 8 inicialmente zera o valor de `TxStart`. Depois ele verifica se `count_bytes` é igual a 4, caso seja incrementa o valor de `count_outputs` pois um dado de saída completo foi transmitido.

Caso não tenham sido enviados 4 bytes, e `RsBusy` seja zero indicando disponibilidade da transmissão o próximo byte de alguma das saídas da rede será escolhido para transmissão. Se `RsBusy` tiver valor um, deve-se esperar neste estado até que ele seja zero.

O estado 9 ativa o bit de sinal `TxStart`, iniciando a transmissão do byte selecionado no estado 8.

O estado 10 desativa o bit de sinal `TxStart` para evitar o envio consecutivo do mesmo dado.

O estado 11 aguarda o bit de sinal `RsBusy` indicar que a transmissão está disponível, iniciando novamente o ciclo de transmissão.

O estado 12 é responsável por verificar se todos os valores de saída da rede foram enviados. Caso tenham sido enviados, o valor de `count_outputs` e `count_bytes` são zerados e o sinal de atualização da camada de contexto `ativa_NN` é ativado retornando depois ao estado 0. Caso contrário o valor de `count_bytes` é zerado e retorna-se ao estado 8 para um novo ciclo de transmissão.

Neste capítulo apresentamos o desenvolvimento das diferentes implementações das funções de ativação, da rede neural perceptron e recorrente, da ferramenta criada para a conversão automática das redes criadas em MATLAB para linguagem de descrição de hardware Verilog e da máquina de estados responsável pela comunicação entre MATLAB e DE2-115 assim como pelo tempo de processamento da rede. Os resultados obtidos e análises de estudos de casos reais serão apresentados no próximo capítulo.

Capítulo 4

Resultados Obtidos

Nesta seção apresentamos os recursos físicos utilizados para a comunicação entre a FPGA e o MATLAB. Serão, também, comparados os resultados das implementações das aproximações das funções de ativação sigmoide com relação ao erro e espaço em hardware ocupado. Serão, também, apresentados os resultados das implementações das redes neurais para quatro casos de teste.

Simulações do MATLAB são executadas nesta seção, deste modo iremos apresentar o sistema computacional no qual estas foram realizadas. Foi utilizado um computador pessoal com processador Intel(R) Core(TM) i5-6600K CPU @ 3.50GHz, memória principal de 8 GiB, sistema operacional *Windows* 10 de 64 bits. A versão do software MATLAB utilizada foi a R2015a.

4.1 Comunicação entre FPGA e MATLAB

Iremos avaliar o custo de implementação de um sistema de comunicação para transmissão ao chip FPGA pelo MATLAB de uma *word* de dados. A Tabela 4.1 apresenta a utilização dos elementos físicos da FPGA.

Tabela 4.1: Recursos de hardware para a comunicação entre MATLAB e FPGA

Elementos lógicos	Registradores	Bits de memória	Multiplicadores de 9 bits
444(<1%)	247	0	0(0%)

O tempo médio em 20 transmissões de um byte de dados do MATLAB ao chip FPGA foi de 6.8080×10^{-4} segundos.

4.2 Implementações da função de ativação

Nesta seção são apresentadas as comparações entre as diferentes implementações das funções de ativação para as funções sigmoide logística e tangente hiperbólica discutidas na Seção 3.2. Os recursos físicos do chip FPGA necessários para cada implementação são apresentadas nesta seção

assim como os erros quadráticos médios entre os resultados obtidos em hardware e em software. Para o cálculo do erro quadrático médio foi utilizado o algoritmo apresentado na Listagem 4.1, sendo MSE o erro quadrático médio, $phi_approx(\cdot)$ o resultado da função de ativação em hardware e $phi(\cdot)$ o resultado da função de ativação em software.

Listing 4.1: Algoritmo para cálculo do erro médio quadrático

```
1
2   MSE = 0;
3   x = -10:0.01:10;
4   for i=1:length(x)
5       MSE = MSE + (phi_approx(x(i))-phi(x(i)))^2;
6   end
7   MSE = MSE/length(x);
```

Note que `phi_approx` se refere à aproximação realizada da função de ativação $\phi(\cdot)$.

4.2.1 Implementação da função sigmoide logística

As Tabelas 4.2 e 4.3 apresentam, respectivamente, os custos de hardware e os erros associados às implementações da função sigmoide logística em hardware.

Note que dentro dos parênteses da Tabela 4.2 estão as ocupações percentuais aproximadas de cada tipo de elemento no chip FPGA da DE2-115 para cada implementação realizada. Percebemos que quanto maior a ordem da aproximação polinomial ou maior número de amostras em intervalos fixos utilizados mais recursos físicos da FPGA são necessários.

Tabela 4.2: Custo de hardware para as implementações da função sigmoide logística

		Elementos Lógicos	Registradores	Bits de Memória	Multiplicadores de 9 bits
Aproximação polinomial	Aproximação de ordem 1	1.063(<1%)	548	45(<1%)	7
	Aproximação de ordem 2	2.704(2%)	1.351	154(<1%)	14(3%)
	Aproximação de ordem 3	3.721(3%)	1.867	207(<1%)	21(4%)
Aproximação por amostragem	Aproximação 64 amostras	1.335(1%)	568	48(<1%)	0
	Aproximação 128 amostras	1.473(1%)	568	48(<1%)	0
	Aproximação 256 amostras	1.771(2%)	568	48(<1)	0
	Aproximação 512 amostras	2.302(2%)	568	48(<1%)	0
Implementação exata	Implementação exata	1.942(2%)	875	4.944(<1%)	47(9%)
Número total de elementos no kit de desenvolvimento DE2-115	Total	/114.480		/3.981.312	/532

Tabela 4.3: Erros das implementações da função de ativação sigmoide logística.

		Erro quadrático médio	Maior erro absoluto
Aproximação polinomial	Aproximação de ordem 1	$494,7000 \times 10^{-06}$	0,0691
	Aproximação de ordem 2	$24,4670 \times 10^{-06}$	0,0172
	Aproximação de ordem 3	$2,4746 \times 10^{-06}$	0,0086
Aproximação por amostragem	Aproximação 64 amostras	$2,7323 \times 10^{-05}$	0,0191
	Aproximação 128 amostras	$1,9120 \times 10^{-05}$	0,0185
	Aproximação 256 amostras	$1,7055 \times 10^{-05}$	0,0182
	Aproximação 512 amostras	$1,6593 \times 10^{-05}$	0,0180
Implementação exata	Implementação exata	$5,4213 \times 10^{-10}$	0,0012

Como esperado o erro médio diminui a medida que aumentamos o grau do polinômio de aproximação ou o número de amostras em intervalos fixos.

As Figuras 4.1, 4.2 e 4.3 apresentam os resultados obtidos do processamento das implementações discutidas da sigmoide logística na FPGA. Os dados de entrada das funções de ativação foram 2001 pontos igualmente espaçados no intervalo $[-10, 10]$.

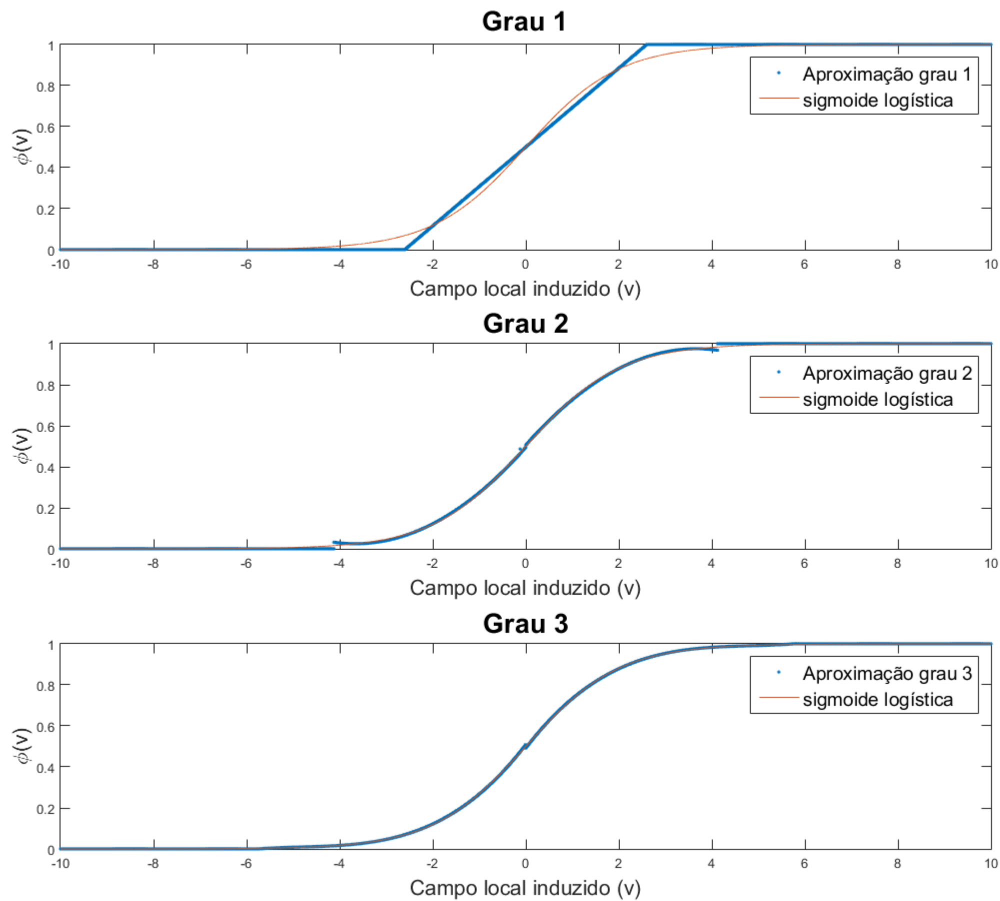


Figura 4.1: Resultados das implementações por aproximação polinomial de grau 1, 2 e 3 para a função sigmoide logística

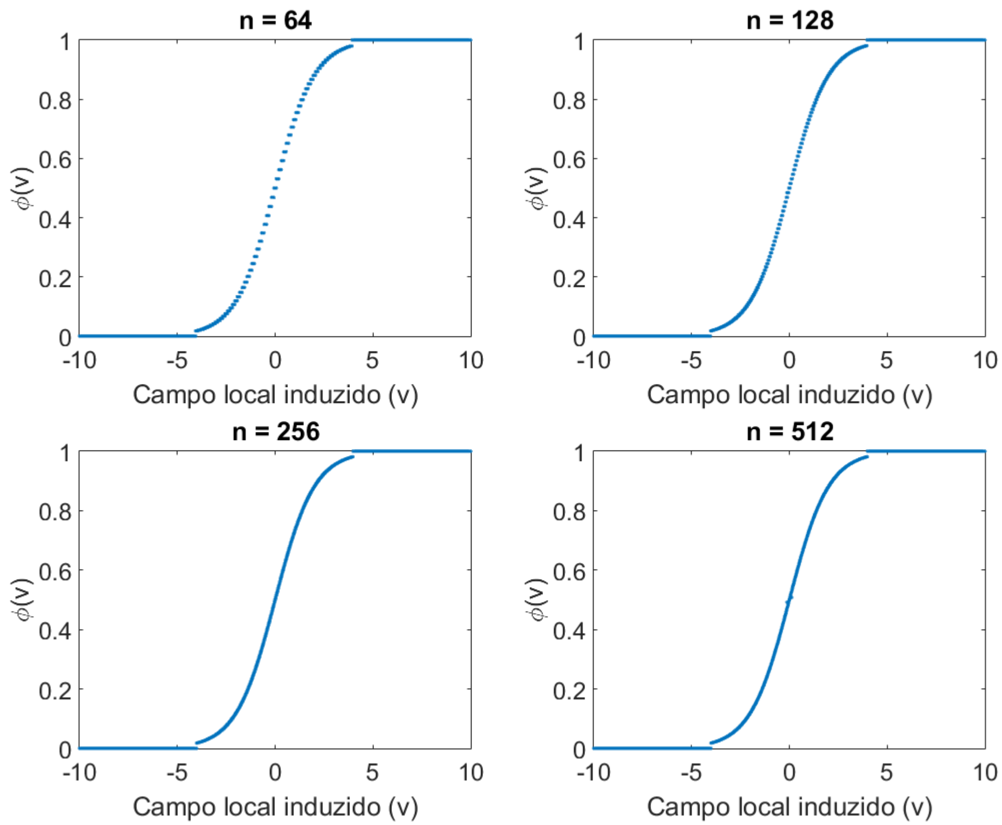


Figura 4.2: Resultados das implementações de aproximação da função sigmoide logística por amostragem

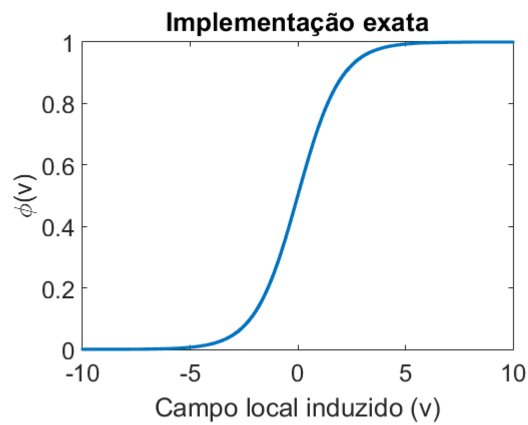


Figura 4.3: Resultado da implementação exata da função sigmoide logística

Como apresentado na Tabela 4.3 percebemos que as curvas em 4.1 e 4.2 se aproximam do formato da sigmoide a medida que aumentamos a ordem dos polinômios e o número de amostras nas aproximações da função de ativação.

4.2.2 Implementação da função tangente hiperbólica

As Tabelas 4.4 e 4.5 apresentam, respectivamente, os custos de hardware e os erros associados às implementações da função sigmoide logística em hardware.

Tabela 4.4: Custo de hardware para as implementações da função tangente hiperbólica

		Elementos Lógicos	Registradores	Bits de Memória	Multiplicadores de 9 bits
Aproximação polinomial	Aproximação de ordem 1	1.045(<1%)	548	45(<1%)	7(1%)
	Aproximação de ordem 2	2.004(2%)	1066	90(<1%)	14(3%)
	Aproximação de ordem 3	3.022(3%)	1.557	162(<1%)	21(4%)
Aproximação por amostragem	Aproximação 64 amostras	1.321(1%)	568	48(<1%)	0
	Aproximação 128 amostras	1.477(1%)	568	48(<1%)	0
	Aproximação 256 amostras	1.787(2%)	568	48(<1%)	0
	Aproximação 512 amostras	2.307(2%)	568	48(<1%)	0
Implementação exata	Implementação exata	3.309(3%)	1.495	5.055(<1%)	78(15%)
Número total de elementos no kit de desenvolvimento DE2-115	Total	/114.480		/3.981.312	/532

De forma análoga à sigmoide logística, ao aumentarmos o grau do polinômio ou o número de amostras nas funções de aproximação, mais recursos físicos da FPGA são demandados.

Tabela 4.5: Erros das implementações da função de ativação tangente hiperbólica

		Erro quadrático médio	Maior erro absoluto
Aproximação polinomial	Aproximação de ordem 1	$989,30 \times 10^{-06}$	0,1383
	Aproximação de ordem 2	$48,913 \times 10^{-06}$	0,0319
	Aproximação de ordem 3	$4,9463 \times 10^{-06}$	0,0171
Aproximação por amostragem	Aproximação 64 amostras	$67,913 \times 10^{-06}$	0,0530
	Aproximação 128 amostras	$21,710 \times 10^{-06}$	0,0310
	Aproximação 256 amostras	$5,4362 \times 10^{-06}$	0,0155
	Aproximação 512 amostras	$1,3676 \times 10^{-06}$	0,0077
Implementação exata	Implementação exata	$1,7936 \times 10^{-9}$	0,0026

Novamente, com o aumento do grau das aproximações polinomiais e do número de amostras temos uma redução dos erros médios quadráticos, como esperado.

As Figuras 4.4, 4.5 e 4.6 apresentam os resultados obtidos do processamento das implementações discutidas da função de ativação na FPGA. Os dados de entrada das funções de ativação foram 2001 pontos igualmente espaçados no intervalo $[-10, 10]$.

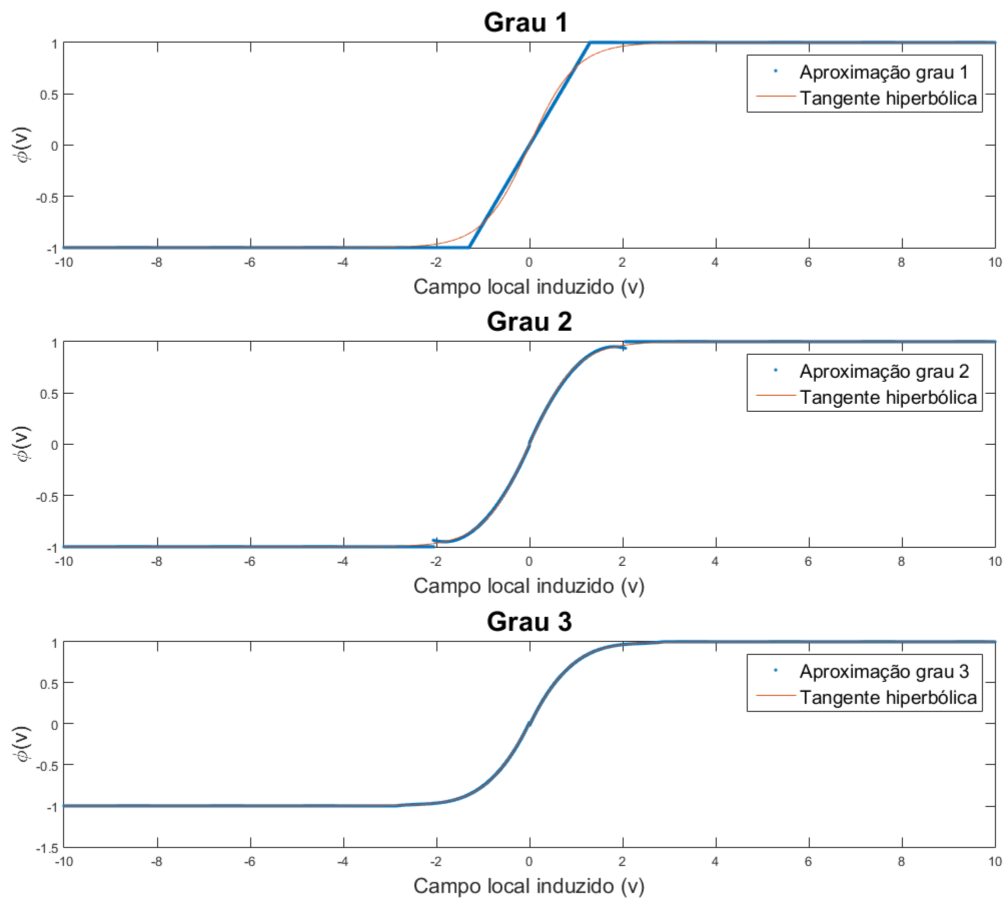


Figura 4.4: Resultados das implementações por aproximação polinomial de grau 1, 2 e 3 para a função tangente hiperbólica

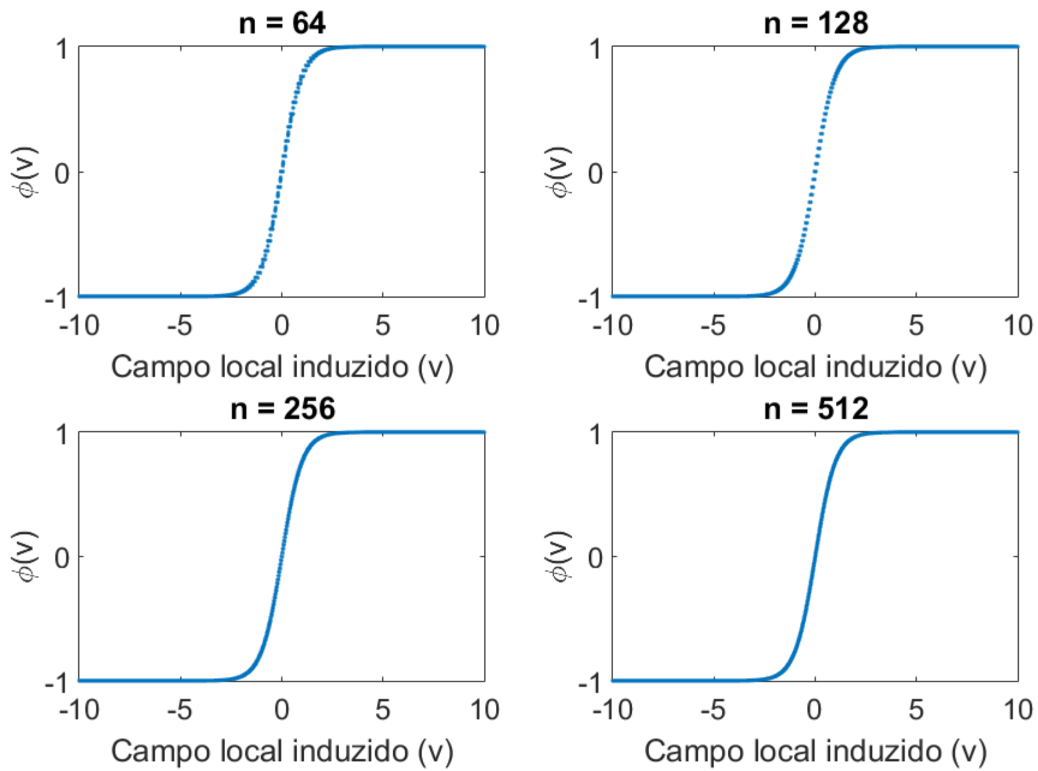


Figura 4.5: Resultados das implementações da função tangente hiperbólica por amostragem em intervalos fixos

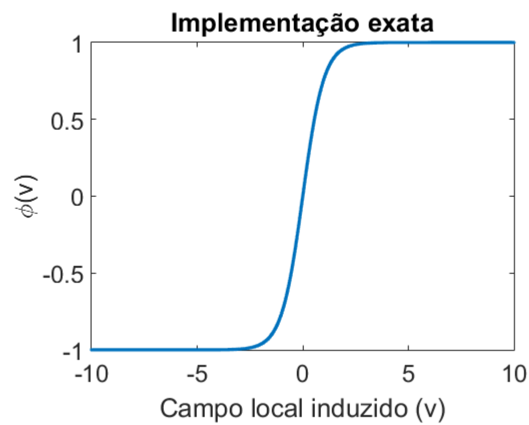


Figura 4.6: Resultado da implementação exata da função tangente hiperbólica

Percebemos que o aumento do grau do polinômio produz curvas com formas que se assemelham cada vez mais a da função tangente hiperbólica.

4.3 Casos de teste

A seguir apresentamos casos de teste que nos permitem validar a ferramenta criada para conversão de redes criadas e treinadas em software MATLAB para linguagem de descrição de hardware Verilog.

4.3.1 Rede neural perceptron para implementação de porta lógica XOR

O primeiro, e mais simples, dos casos de teste é a realização de uma rede neural perceptron *feedforward* capaz de emular o comportamento de uma porta lógica XOR de duas entradas. Para isto a estrutura de rede escolhida é a apresentada na Figura 4.7. Note que esta rede possui 4 neurônios na camada escondida com funções de ativação sigmoide logística e 1 neurônio na camada de saída com função de ativação linear. O treinamento desta rede em MATLAB foi realizado pelo código apresentado em I.1.

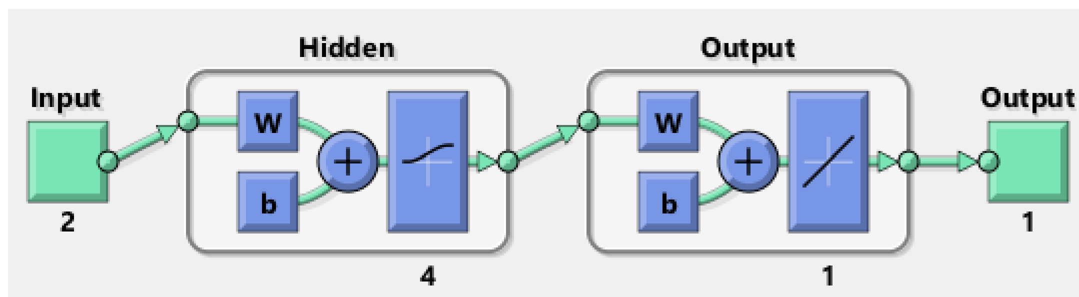


Figura 4.7: Estrutura da rede perceptron para resolução do problema XOR

O custo da implementação desta rede com as diferentes aproximações de funções de ativação são apresentadas na Tabela 4.6. O custo inclui a síntese do módulo para a comunicação entre a FPGA e o MATLAB.

Tabela 4.6: Custo da implementação da rede neural perceptron para implementação de porta lógica XOR

		Elementos Lógicos	Registradores	Bits de Memória	Multiplicadores de 9 bits
Aproximação polinomial	Aproximação de ordem 1	16.392(14%)	8.350	895(<1%)	112(21%)
	Aproximação de ordem 2	23.075(20%)	11.622	1.287(<1%)	140(26%)
	Aproximação de ordem 3	27.091(24%)	13.617	1.547(<1%)	168(32%)
Aproximação por Intervalos fixos	Aproximação 256 amostras	19.122(17%)	8.349	884(<1%)	84(16%)
Implementação exata	Implementação exata	19.741(17%)	9.567	20.524(<1%)	272(51%)
Número total de elementos no kit de desenvolvimento DE2-115	Total	/114.480		/3.981.312	/532

Os resultados obtidos pelas diferentes implementações da rede neural proposta são apresentados na Tabela 4.7

Tabela 4.7: Resultados obtidos pelas diferentes implementações da rede perceptron para problema XOR

		Entradas			
		[0,0]	[0,1]	[1,0]	[1,1]
MATLAB	MATLAB	$-1,5887 \times 10^{-8}$	1,0000	1,0000	$1,9262 \times 10^{-9}$
Aproximação polinomial	Aproximação de ordem 1	0,0366	1,0346	0,9314	$3,4596 \times 10^{-4}$
	Aproximação de ordem 2	0,0067	1,0017	0,9909	-0,0024
	Aproximação de ordem 3	0,0024	0,9988	0,9992	-0,0026
Aproximação por amostragem	256 amostras	-0,0032	1,0007	0,9955	-0,0034
Implementação exata	Implementação exata	$-5,9605 \times 10^{-8}$	1,0000	1,0000	$-1,1176 \times 10^{-8}$

Iremos, agora, comparar as latências das redes neurais implementadas na FPGA e em MATLAB

com relação à um ciclo de processamento da rede. Para o cálculo da latência das redes em FPGA serão utilizadas as Equações 3.16 e 3.17. Para a rede em MATLAB, no entanto, utilizaremos a média do processamento da rede em 20 ciclos. Os resultados são apresentados na Tabela 4.8.

Tabela 4.8: Latência em segundos para as redes neurais para o problema da porta lógica XOR

	Latência em segundos
MATLAB	$0,0058$
Aproximação de ordem 1	$1,14 \times 10^{-6}$
Aproximação de ordem 2	$1,52 \times 10^{-6}$
Aproximação de ordem 3	$1,76 \times 10^{-6}$
256 amostras	$1,16 \times 10^{-6}$
Implementação Exata	$1,52 \times 10^{-6}$

Note que os resultados obtidos relacionam a quantidade de recursos físicos da FPGA utilizados com a precisão, e foi observado que maior utilização de espaço na FPGA implica menor erro entre as redes em FPGA e MATLAB, como esperado.

4.3.2 Rede perceptron para classificação de espécies da flor de íris

A classificação de espécies da flor de íris é um problema bastante conhecido na literatura e foi primeiramente apresentado em 1936 pelo estatístico Britânico Ronald Fisher em seu artigo [18]. O problema se resume em utilizar as características das flores íris para as classificar em três classes distintas, sendo elas: Íris virgínica, iris versicolor e iris setosa. As características utilizadas para a classificação são o comprimento da sépala, largura da sépala, comprimento da pétala e largura da pétala das flores. O conjunto de dados é disponibilizado pelo software MATLAB com o nome `fisheriris.mat` e consiste em 150 amostras. Cada amostra consiste nos dados de comprimento e largura das pétalas e sépalas e a classificação correta entre uma das três classes de íris. Para a realização desta tarefa foi utilizada uma rede neural perceptron com a estrutura mostrada na Figura 4.8. Note que a rede consiste em uma camada escondida com 8 neurônios com função de ativação sigmoide logística e 3 neurônios de saída que também utilizam a função sigmoide logística. Cada uma das saídas representa uma das classes de flor de íris. O conjunto de dados foi separado para a etapa de treinamento da seguinte forma: 70% para o treinamento, 15% para a validação e 15% para o teste. O treinamento desta rede foi feito utilizando o código MATLAB apresentado em I.2.

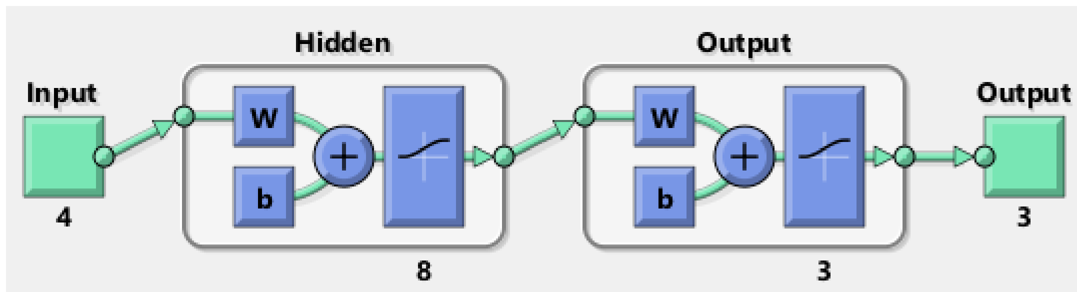


Figura 4.8: Estrutura da rede perceptron para classificação da flor de íris

A Tabela 4.9 apresenta o custo para a implementação da rede neural para classificação da flor de íris utilizando as diferentes aproximações da função de ativação sigmoide logística. Note que o custo inclui a síntese do módulo para a comunicação entre a FPGA e o MATLAB.

Tabela 4.9: Custo da implementação da rede neural perceptron para classificação da flor de íris

		Elementos Lógicos	Registradores	Bits de Memória	Multiplicadores de 9 bits
Aproximação polinomial	Aproximação de ordem 1	67.682(59%)	33.626	3.593(<1%)	469(88%)
	Aproximação de ordem 2	90.580(79%)	42.550	4.726(<1%)	546(103%)
	Aproximação de ordem 3	102.528(90%)	48.105	5.386(<1%)	623(117%)
Aproximação por Intervalos fixos	Aproximação 256 amostras	73.858(65%)	33.357	3.536(<1%)	392(74%)
Implementação exata	Implementação exata	82.474(72%)	38.049	52.528(1%)	909(171%)
Número total de elementos no kit de desenvolvimento DE2-115	Total	/114.480		/3.981.312	/532

Note que as implementações apresentadas na Tabela 4.9 que utilizam mais de 100% dos recursos da FPGA não são fisicamente realizáveis. Logo, para a rede neural da flor de íris não é possível utilizar as funções de ativação por aproximação polinomial de ordens 2 e 3 e nem a implementação exata utilizando o módulo IP de exponenciação da Altera. A Tabela 4.10 apresenta o erro quadrático médio e o erro máximo na classificação do conjunto de dados de teste entre as implementações em hardware e a implementação em MATLAB.

A relação entre os tempos de execução da rede em software MATLAB e hardware FPGA também nos é de interesse, desta forma os tempos médios para um ciclo de processamento da rede são apresentados na Tabela 4.11.

Tabela 4.10: Erros associados a cada implementação da rede neural para classificação da flor de íris quando comparados aos resultados da rede em MATLAB

		Erro médio quadrático	Maior erro absoluto
Aproximação polinomial	Aproximação de ordem 1	0,0056	0,1148
Aproximação por Intervalos fixos	Aproximação 256 amostras	$7,3977 \times 10^{-5}$	0,0198

Tabela 4.11: Latência em segundos para as redes neurais de classificação da flor de íris.

	Latência em segundos
MATLAB	0,0064
Aproximação de ordem 1	$1,16 \times 10^{-6}$
256 amostras	$1,14 \times 10^{-6}$

Note que os tempos de processamento da rede para as implementações em FPGA são obtidos do número de ciclos de clock resultado da Equação 3.16, este valor é então multiplicado pelo inverso da frequência da DE2-115 que é de 50MHz.

Observando-se as tabelas 4.9 e 4.10 percebemos que a rede sintetizada com função de ativação aproximada por amostragem obteve erros menores quando comparada a rede sintetizada por aproximação polinomial de grau 1. O custo da implementação de ambas as redes foi similar e desta forma pode-se dizer que a função de ativação por amostragem é uma escolha mais eficiente para resolução do problema de classificação de espécies de flores de íris.

4.3.3 Rede recorrente para realização de operação lógica XOR temporal

A lógica XOR temporal é definida, neste trabalho, como um circuito com uma entrada e uma saída, capaz de realizar a operação XOR entre duas entradas consecutivas. A estrutura da rede utilizada para resolver este problema é a apresentada na Figura 4.9. O código utilizado para o treinamento desta rede é aquele apresentado em I.1.

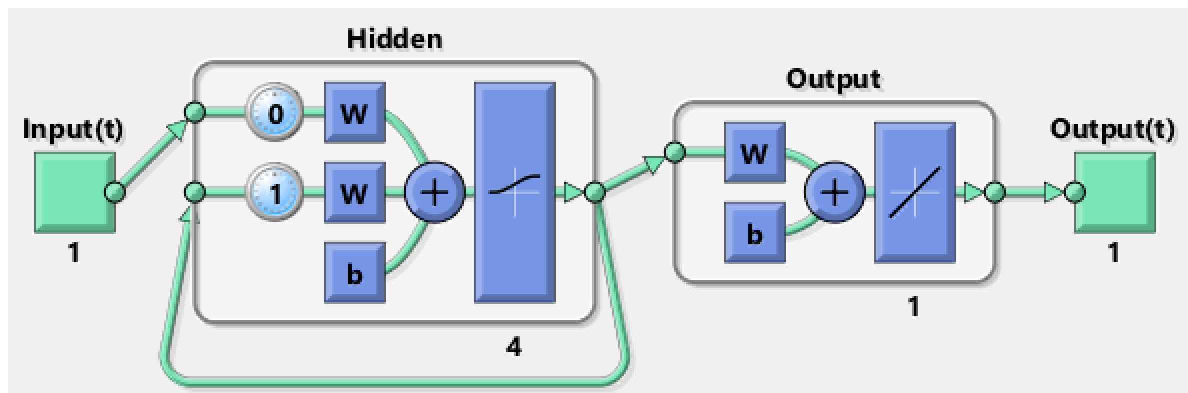


Figura 4.9: Estrutura da rede neural recorrente para implementação da lógica XOR temporal

O custo de implementação da rede neural na FPGA utilizando as funções de ativação aproximadas junto ao módulo de comunicação entre FPGA e MATLAB é apresentado na Tabela 4.12

Tabela 4.12: Custo da implementação da rede neural recorrente para resolver o problema XOR temporal

		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

O erro quadrático médio e o erro absoluto máximo para as implementações são apresentadas nas Tabelas 4.12 e 4.13. Para a obtenção do erro apresentado foi utilizado o vetor

$$v = [0\ 0\ 1\ 1\ 0\ 0\ 1\ 0\ 0\ 0\ 1\ 1\ 1] \quad (4.1)$$

com os elementos do vetor sendo as entradas consecutivas da rede.

Tabela 4.13: Erros associados a cada implementação da rede neural para resolução do problema XOR temporal

		Erro médio Quadrático	Erro absoluto máximo
Aproximação polinomial	Aproximação de ordem 1	0,0126	0,2088
	Aproximação de ordem 2	$3,3751 \times 10^{-4}$	0,0313
	Aproximação de ordem 3	$5,5562 \times 10^{-5}$	0,0163
Aproximação por Intervalos fixos	Aproximação 256 amostras	$4,03 \times 10^{-4}$	0,0324
Implementação exata	Implementação exata	$1,1968 \times 10^{-13}$	$5,8277 \times 10^{-7}$

As latências para um ciclo de processamento das RNA para o problema do xor temporal são apresentadas na Tabela 4.14.

Tabela 4.14: Latência em segundos para as redes neurais para o problema do xor temporal.

	Latência em segundos
MATLAB	$5,07 \times 10^{-4}$
Aproximação de grau 1	$1,28 \times 10^{-6}$
Aproximação de grau 2	$1,68 \times 10^{-6}$
Aproximação de grau 3	$1,90 \times 10^{-6}$
256 amostras	$1,30 \times 10^{-6}$
Implementação Exata	$1,66 \times 10^{-6}$

Para a implementação da rede neural para resolução do problema XOR temporal foi possível embarcar as redes com todos os tipos de aproximações de funções de ativação e desta forma é possível observar como a diminuição do erro está relacionada ao aumento de recursos físicos utilizados. Na questão de tempo de processamento não há vantagem explícita sendo que todas as latências possuem a mesma ordem de grandeza.

4.3.4 Rede de Elman para previsão *online* de oportunidades em transmissões oportunísticas em redes de comunicação *wireless*

Um dos objetivos deste trabalho é a implementação em FPGA da rede neural, desenvolvida em [19] e implementada em ponto fixo em [20], que de forma *online* consiga prever oportunidades de transmissões em redes sem fio, utilizando aritmética de ponto flutuante. A Figura 4.10 apresenta a estrutura desta rede.

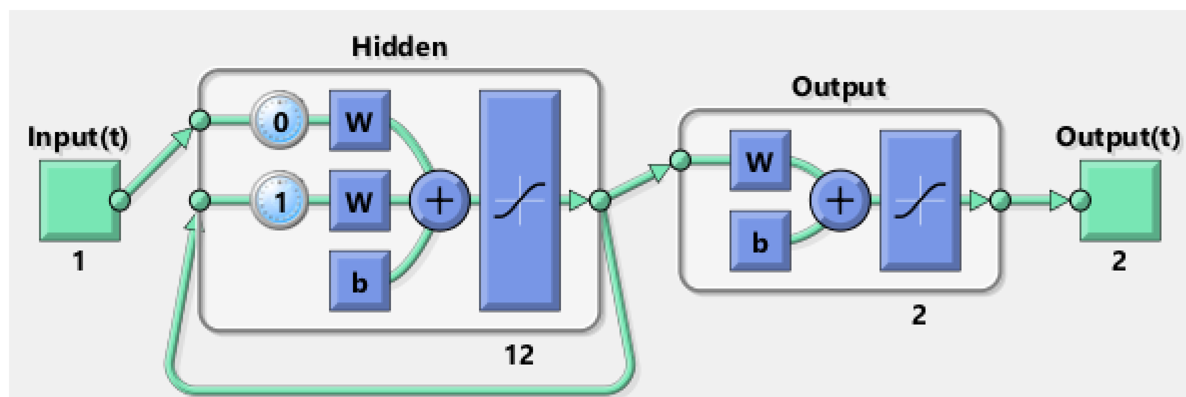


Figura 4.10: Estrutura da rede para previsão de oportunidades em transmissões oportunísticas em redes de comunicação *wireless*

Porém, para os módulos criados e as configurações adotadas neste projeto para descrição de redes neurais, não foi possível implementar esta rede mesmo quando utilizados os módulos que menos consumiam recursos físicos da FPGA. Na Tabela 4.15 apresentamos a quantidade de recursos utilizados na tentativa de sintetizar esta rede, retirado o módulo de comunicação, utilizando funções de ativação polinomial de grau 1, aproximada por 64 amostras de intervalos fixos e implementação exata.

Tabela 4.15: Tabela de utilização de recursos de hardware para a implementação da rede neural para previsões de oportunidades de transmissão em redes sem fio

Módulo de função de ativação	Elementos lógicos	Registradores	Bits de memória	Multiplicadores de 9 bits
Aproximação polinomial grau 1	191.033(167%)	88.636	10.250(<1%)	1.358(255%)
Aproximação 64 amostras	193.676(169%)	88.562	10.204(<1%)	1.260(237%)
Implementação exata	218.022(190%)	100.835	79.600(2%)	2.290(430%)

Note que o número de elementos lógicos e multiplicadores de 9 bits necessários para as implementações estão distantes da capacidade real da FPGA presente no kit de desenvolvimento

DE2-115. Desta forma não é possível realizar os testes para esta topologia de rede requerida.

4.3.5 Limite físico para implementação de redes neurais no kit de desenvolvimento DE2-115

Na Subseção 4.3.4 foi apresentada uma rede neural que não foi capaz de ser sintetizada na FPGA para nenhuma das aproximações de funções de ativação propostas neste trabalho. Iremos agora estudar qual o tamanho limite de uma rede que emule uma porta lógica XOR de duas entradas que possa ser sintetizada utilizando todas os módulos de funções de ativação propostas neste trabalho. Para isto iremos tentar sintetizar as redes com módulos de função de ativação exatas, uma vez que estas requerem mais recursos físicos quando comparadas as outras funções de ativação. Fizemos os testes para redes com 7 e 8 neurônios na camada escondida, os resultados são apresentados na Tabela 4.16.

Tabela 4.16: Recursos físicos utilizados para a implementação das redes neurais para o problema do XOR para teste de limite físico da FPGA

Número de neurônios	Elementos lógicos	Registradores	Bits de memória	Multiplicadores de 9 bits
7	34.500(30%)	16.432	35.884	476(89%)
8	42.199(37%)	19.557	41.004	544(102%)

Podemos observar que a implementação com 8 neurônios não foi capaz de ser sintetizada, uma vez que necessita de mais multiplicadores de 9 bits do que os disponíveis no chip FPGA. A rede com 7 neurônios, por outro lado, foi sintetizada e é deste modo o limite que pode ser implementado em um kit de desenvolvimento DE2-115.

Neste capítulo apresentamos os resultados obtidos da implementação das redes neurais em quatro casos de testes.

Capítulo 5

Conclusões

Neste trabalho foi proposta uma ferramenta de conversão de redes neurais artificiais criadas e treinadas em software MATLAB para linguagem de descrição de hardware Verilog. Tal conversão foi feita de modo que o código resultante pudesse ser prontamente descarregado e executado em um kit de desenvolvimento DE2-115 com um chip FPGA embarcado. A rede neural embarcada no chip FPGA para o problema do XOR temporal obteve um erro mínimo de $1,1968 \times 10^{-13}$ entre o resultado da rede em MATLAB e aquela criada pela ferramenta proposta neste trabalho e embarcada em FPGA quando utilizando módulos de função de ativação exatos. Este foi o menor erro obtido dentre todos os casos de teste. O maior erro, por outro lado, ocorreu para a rede emulando a porta lógica XOR. O valor máximo de erro obtido nesta rede foi de $0,0366$ quando utilizada uma função de ativação aproximada por polinômio de grau 1.

O caso de teste para classificação das flores de íris nos apresenta a primeira rede neural que não foi sintetizável para todas as funções de ativação neste trabalho, sendo implementada somente pelas funções de ativação aproximadas por polinômio de grau 1 e por 256 amostras. Foi apresentado o caso de teste de uma rede neural recorrente de Elman para previsão de oportunidades de transmissão em redes *wireless* que não foi possível ser sintetizada no chip FPGA do kit de desenvolvimento DE2-115 por nenhuma das implementações de funções de ativação propostas neste trabalho. O limite de tamanho de uma rede neural que seja sintetizável por todas as funções de ativação propostas para o problema da porta lógica XOR foi de uma rede com 7 neurônios na camada escondida. Desta forma fica evidente as limitações impostas a este trabalho por conta do hardware limitado no chip FPGA do kit de desenvolvimento DE2-115.

É importante ressaltar que as redes produzidas pela ferramenta proposta são capazes de ser implementadas em todas as placas da Altera, sendo que deste modo o limite de tamanho para síntese de redes neurais está nas limitações de hardware dos chips FPGA. Melhorias, no entanto, podem ser realizadas de forma a tornar a utilização de recursos físicos mais eficiente.

5.1 Trabalhos Futuros

Como visto neste trabalho, o chip FPGA utilizado não possui recursos físicos suficientes para síntese de redes neurais de maior porte como apresentado em alguns dos casos de teste estudados. Desta forma trabalhos futuros podem seguir a linha de tentar reduzir a utilização de hardware da rede. Isto pode ser obtido por melhorias nas implementações das funções de ativação, utilização de módulos IP para operações de ponto flutuante mais otimizados e redução da notação de ponto flutuante da precisão simples utilizada para a precisão *half* de 16 bits. Além dos recursos, percebemos que o protocolo de comunicação implementado foi o maior *bottleneck* do sistema, prejudicando de forma significativa o tempo total de processamento do sistema com redes neurais em aplicações de tempo real. Nesta parte um próximo passo seria realizar a comunicação através de portas USB ou Ethernet de forma a aumentar a velocidade das transmissões de dados.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] READING, D. *Typesetting neural network diagrams with TeX*. 2019. `https://medium.com/momenton/typesetting-neural-network-diagrams-with-tex-4920b6b9fc19`. Online; Acesso: 2019-06-24.

[2] MAHMOUD, S.; LOTFI, A.; LANGENSIEPEN, C. Behavioural pattern identification and prediction in intelligent environments. *Applied Soft Computing*, v. 13, p. 1813–1822, 04 2013.

[3] CHU, P. P. *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version*. [S.l.: s.n.], 2007. ISBN 0470185317, 9780470185315.

[4] TERASIC. *DE2-115: User manual*. 2012. `https://www.intel.com/content/dam/altera-www/global/en_US/portal/dsn/42/doc-us-dsnbk-42-1404062209-de2-115-user-manual.pdf`. Online; Acesso: 2019-07-03.

[5] SEMICONDUCTOR, D. *Fundamentals of RS–232 Serial Communications*. 1998. `http://ecee.colorado.edu/~mcclurel/dan83.pdf`. Online; Acesso: 2019-06-24.

[6] HAYCOCK, R.; YORK, T. Hardware implementation of boolean neural networks. In: *IEE Colloquium on Hardware Implementation of Neural Networks and Fuzzy Logic*. [S.l.: s.n.], 1994. p. 3/1–3/4.

[7] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, p. 1–70, Aug 2008.

[8] HAYKIN, S. *Neural networks: a comprehensive foundation*. [S.l.]: Prentice Hall PTR, 1994.

[9] ELMAN, J. L. Finding structure in time. *Cognitive Science*, v. 14, n. 2, p. 179 – 211, 1990. ISSN 0364-0213. Disponível em: <http://www.sciencedirect.com/science/article/pii/036402139090002E>.

[10] MACQUEEN, J. Some methods for classification and analysis of multivariate observations. In: *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*. Berkeley, Calif.: University of California Press, 1967. p. 281–297. Disponível em: <https://projecteuclid.org/euclid.bsmsp/1200512992>.

[11] LV, C. et al. Levenberg-marquardt backpropagation training of multilayer neural networks for state estimation of a safety critical cyber-physical system. *IEEE Transactions on Industrial Informatics*, PP, p. 1–1, 11 2017.

[12] ZHANG, Z. et al. Short-term prediction for opening price of stock market based on self-adapting variant pso-elman neural network. In: *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*. [S.l.: s.n.], 2017. p. 225–228. ISSN 2327-0594.

[13] LU, Y. et al. Vegetable price prediction based on pso-bp neural network. In: *2015 8th International Conference on Intelligent Computation Technology and Automation (ICICTA)*. [S.l.: s.n.], 2015. p. 1093–1096.

[14] ALTERA. *Introduction to the Quartus II software*. 2010. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/archives/intro_to_quartus2.pdf`. Online; Acesso: 2019-07-03.

[15] BELL, C.; MUDGE, C.; MCNAMARA, J. E. *Computer engineering : A DEC view of hardware systems design*. [S.l.: s.n.], 1978. ISBN 0-932376-00-2.

[16] XIE, Z. A non-linear approximation of the sigmoid function based on fpga. In: *2012 IEEE Fifth International Conference on Advanced Computational Intelligence (ICACI)*. [S.l.: s.n.], 2012. p. 221–223.

[17] HASTIE ROBERT TIBSHIRANI, J. F. T. *The elements of statistical learning: Data mining, inference, and prediction*. 2nd ed. 2009. corr. 3rd printing 5th printing.. ed. [S.l.]: Springer, 2009. (Springer Series in Statistics). ISBN 0387848576,9780387848570,9780387848587.

[18] FISHER, R. A. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, v. 7, n. 7, p. 179–188, 1936.

[19] FERREIRA, P. A. L.; FERNANDES, S. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação), *Previsão de oportunidades de acesso utilizando redes neurais artificiais recorrentes*. Universidade de Brasília, Departamento de ciência da computação, 2016. Disponível em: <http://bdm.unb.br/handle/10483/17246>.

[20] MAKIUCHI, M. R. Trabalho de Graduação em Engenharia de Controle e Automação, *Desenvolvimento de rede neural artificial recorrente em FPGA para previsão online de oportunidades em transmissões oportunísticas em redes de comunicação wireless*. Universidade de Brasília, Departamento de ciência da computação, 2018. Disponível em: <http://bdm.unb.br/handle/10483/22068>.

ANEXOS

I. CÓDIGOS UTILIZADOS NO TREINAMENTO DAS REDES NEURAIS

Neste anexo são apresentados os códigos em MATLAB utilizados no treinamento das redes neurais apresentadas na Seção 4.3.

Listing I.1: Código para treinamento da rede neural para problema da porta lógica XOR e da lógica XOR temporal

```
1
2  clear all
3  clc
4  str = 'Rede feedforward(0) ou recorrente(1)?\n';
5  type_net = input(str);
6
7  if type_net == 0
8
9  net = feedforwardnet([4]);
10 P = [0 0;0 1;1 0;1 1;0 0;0 1;1 0;1 1;0 0;0 1;1 0;1 1;0 0; ...
11      0 1;1 0;1 1;0 0;0 1;1 0;1 1;0 0;0 1;1 0;1 1;0 0;1 1;0 1 1]';
12 T = [0  ;1  ;1  ;0  ;0  ;1  ;1  ;0  ;0  ;1  ;1  ;0  ;0  ;1  ;  ...
13      1  ;0  ;0  ;1  ;1  ;0  ;0  ;1  ;1  ;0  ;0  ;1  ;1  ;0  ]';
14 net = configure(net, P, T);
15 net.trainParam.goal = 1e-15;
16 net.trainParam.epochs = 1000;
17 net.divideFcn = '';
18 net.inputs{1,1}.processFcns = {};
19 net.outputs{1,2}.processFcns = {};
20 for i=1:net.numLayers-1
21     net.layers{i}.transferFcn = 'logsig';
22 end
23 net = train(net, P, T);
24 sim(net, P);
25
26
27 else
28     % Rede com camada de contexto
29     x1 = [1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
           ...
30          0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0];
```

```
31      t1 = [NaN 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            ...
32           1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  1];
33      x2 = [1  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            ...
34           1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  1];
35      t2 = [NaN 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            ...
36           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0];
37      x3 = [0  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            ...
38           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0];
39      t3 = [NaN 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            ...
40           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0];
41      X = catsamples(x1, x2, x3);
42      T = catsamples(t1, t2, t3);
43      X = num2cell(X);
44      T = num2cell(T);
45      net = layrecnet(1,[4]);
46      net.divideFcn = '';
47      net.trainParam.goal = 1e-20;
48      net.trainParam.epochs = 10000;
49      net.trainParam.min_grad = 0;
50      for i=1:net.numLayers-1
51          net.layers{i}.transferFcn = 'logsig';
52      end
53      net.inputs{1,1}.processFcns = {};
54      net.outputs{1,2}.processFcns = {};
55
56      [Xs,Xi,Ai,Ts] = preparets(net,X,T);
57      net = train(net,Xs,Ts,Xi,Ai);
58      view(net)
59      Y = net(Xs,Xi,Ai);
60
61  end
```

Listing I.2: Código para treinamento da rede neural para problema da classificação das flores de íris

```
1
2  clear all
3  clc
```

```
4 load fisheriris.mat
5
6 % 70% dos dados para o treinamento 15% para validacao e teste
7 aux = round(50*0.7);
8 a = round(150*0.7);
9 b = round(a+150*0.15);
10 c = round(b+150*0.15)-1;
11
12 x_setosa = meas(1:50,:);
13 x_versicolor = meas(51:100,:);
14 x_virginica = meas(101:150,:);
15
16 P = [x_setosa(1:aux,:);x_versicolor(1:aux,:);x_virginica(1:aux,:)];
17 Tra = [x_setosa(aux+1:50,:);x_versicolor(aux+1:50,:);...
18                                 x_virginica(aux+1:50,:)];
19 P = [P;Tra];
20
21 T = [ones(aux,1) zeros(aux,1) zeros(aux,1); zeros(aux,1) ones(aux,1)
     ...
22     zeros(aux,1); zeros(aux,1) zeros(aux,1) ones(aux,1);];
23 T_tra = [ones(15,1) zeros(15,1) zeros(15,1); zeros(15,1) ones(15,1) ...
24     zeros(15,1); zeros(15,1) zeros(15,1) ones(15,1);];
25 T = [T;T_tra];
26
27 net = feedforwardnet([4]);
28 net = configure(net, P, T);
29 net.trainParam.goal = 1e-15;
30 net.trainParam.epochs = 1000;
31 net.divideFcn = 'divideind';
32 net.divideParam.trainInd = 1:a;
33 net.divideParam.valInd   = a+1:b;
34 net.divideParam.testInd  = b+1:c;
35 net.inputs{1,1}.processFcns = {};
36 net.outputs{1,2}.processFcns = {};
37 for i=1:net.numLayers-1
38     net.layers{i}.transferFcn = 'logsig';
39 end
40 net = train(net, P', T');
41 sim(net, P);
```

II. FORMAS DE ONDA

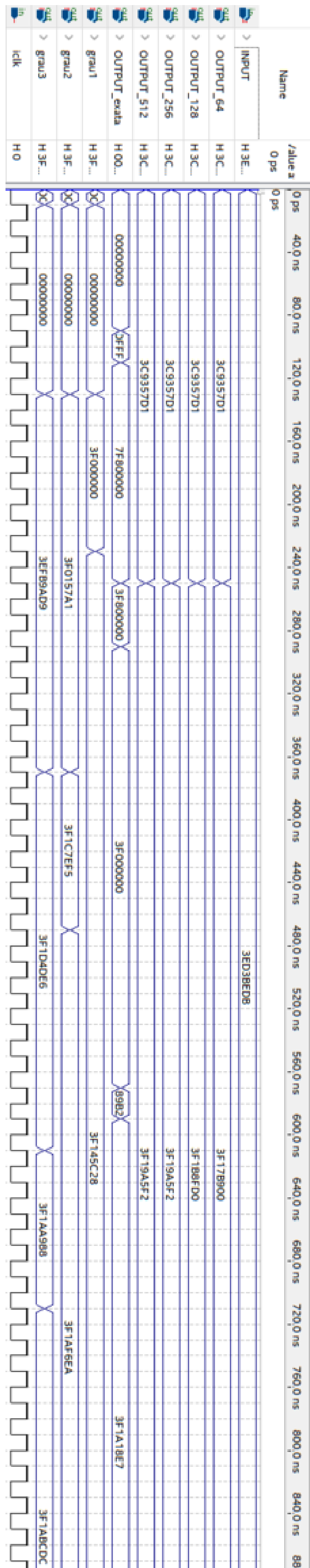


Figura II.1: Forma de onda das implementações das funções de ativação sigmoide logísticas

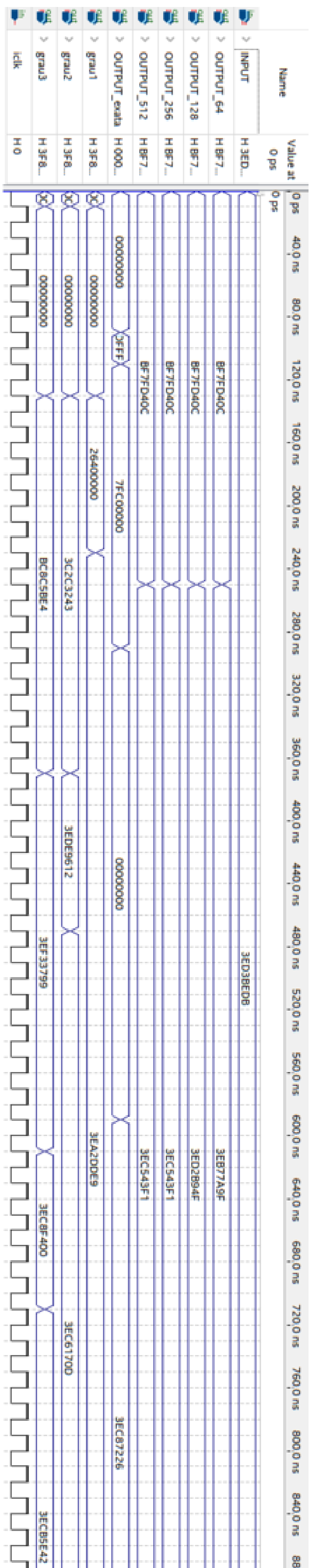


Figura II.2: Forma de onda das implementações das funções de ativação tangente hiperbólica