CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
CAMPUS TIMÓTEO

Alysson Kelvim Caetano da Silva

IMPLEMENTAÇÃO E TREINAMENTO DE UMA REDE NEURAL ARTIFICIAL EM UM DISPOSITIVO FPGA COM BASE DE DADOS DE CÂNCER DE MAMA

Timóteo

2021

Alysson Kelvim Caetano da Silva

IMPLEMENTAÇÃO E TREINAMENTO DE UMA REDE NEURAL ARTIFICIAL EM UM DISPOSITIVO FPGA COM BASE DE DADOS DE CÂNCER DE MAMA

Monografia apresentada à Coordenação de Engenharia de Computação do Campus Timóteo do Centro Federal de Educação Tecnológica de Minas Gerais para obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Elder de Oliveira Rodrigues

Timóteo

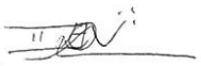
2021

Alysson Kelvim Caetano da silva

IMPLEMENTAÇÃO E TREINAMENTO DE UMA REDE NEURAL ARTIFICIAL EM UM DISPOSITIVO FPGA COM BASE DE DADOS DE CÂNCER DE MAMA

Trabalho de Conclusão de Curso apresentado ao Curso de Engenharia de Computação do Centro Federal de Educação Tecnológica de Minas Gerais, campus Timóteo, como requisito parcial para obtenção do título de Engenheiro de Computação.

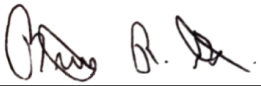
Trabalho aprovado. Timóteo, 13 de Abril de 2021:



Prof. Dr. Elder de Oliveira Rodrigues
Orientador

Prof. Me. Douglas Nunes de Oliveira
Professor Convidado

Prof. Me. Nelson Alexandre Estevão
Professor Convidado



Prof. Dr. Bruno Rodrigues Silva
Professor Convidado

Timóteo
2021

Aos meus pais.

Agradecimentos

Primeiramente, agradeço a Deus, autor da minha existência, por esta vida, pelo futuro e por tudo o que passou. Agradeço também minha família, meu pai, Josimar Loureiro, meu padrasto, José Manoel e meu irmão Álvaro Kalel por todo o suporte. Em especial, agradeço a minha mãe, Rose Caetano, que desde cedo me instruiu no caminho da sabedoria e me incentivou a obter conhecimento, meu esforço e dedicação devo a você.

Sou grato também aos meus professores, que para além das aulas, tiraram dúvidas, aconselharam e ajudaram. Em especial, agradeço o professor Elder Rodrigues, que desde meu curso técnico tem me orientado no desenvolvimento de projetos e que pacientemente compartilhou seu conhecimento e disponibilizou tempo para me orientar. Agradeço também aos professores Romerito Valeriano e Rodrigo Gaiba, que me confiaram trabalhos que foram de grande importância para mim.

Agradeço também ao meu supervisor de estágio na Fundação São Francisco Xavier, Wilson Guilherme, pela paciência e prontidão em ensinar tudo o que lhe foi possível durante meu estágio.

Não posso deixar de agradecer meus amigos do CEFET-MG: Ana Clara, Cristiane de Jesus, Enmili Inocência, Karine Rodrigues e Matheus Otero, pessoas que foram suporte durante todo esse tempo e tornaram esse trajeto mais prazeroso.

Agradeço também ao Wander Dias, que na coordenação do curso esteve sempre de prontidão a sanar nossas dúvidas e atender nossas demandas diárias. Ademais, agradeço às auxiliares de limpeza, que semanalmente simpaticamente nos encontravam nos laboratórios, aos bibliotecários que sem hesitar facilitavam nossas buscas, aos monitores que dividiram um pouco do seu aprendizado, enfim a cada pessoa que não citei o nome mas que diretamente ou indiretamente contribuiu para minha formação. Meus sinceros agradecimentos!

“Os grandes navegadores
devem sua reputação aos temporais e tempestades”.
Epicuro

Resumo

As redes neurais se dedicam a implementar modelos matemáticos que se aproximem à neurônios biológicos, essas implementações se dão através do agrupamento de unidades simples de processamento através de um algoritmo de aprendizagem. Porém, redes que possuem muitos neurônios podem tornar processamento matemático via *software* demorado, contudo, uma rede neural pode ser treinada em *hardware* fazendo o uso de paralelismo em diversas operações. Neste âmbito, os dispositivos FPGAs podem ser úteis, pois permitem a sintetização através de ferramentas de *software* Linguagem de Descrição de *Hardware*. Neste trabalho, é implementado o modelo *perceptron* multicamadas em *hardware* utilizando a linguagem VHDL, o mesmo foi sintetizado e gravado no dispositivo FPGA Cyclone II 2C35 utilizando o *kit* de desenvolvimento Altera DE2 juntamente com alguns periféricos. A rede neural realizou o treinamento da base de dados com sucesso utilizando o *hardware* proposto e através da integração criada com o computador é possível enviar diversas bases de dados ao dispositivo e treiná-las. O erro de aproximação utilizando uma base de teste se mostrou harmonioso com os resultados em Java e o tempo de treinamento da rede neural foi menor de acordo com a quantidade de elementos lógicos utilizados na rede. O *hardware* projetado mostrou-se eficaz naquilo em que se propõe a solucionar. O comportamento da rede neural em VHDL foi semelhante ao da rede implementada em Java. Devido à limitação de blocos lógicos, algumas estratégias tiveram de ser implementadas para contornar a restrição física do Cyclone II 2C35, porém, utilizando dispositivos mais novos e de maior capacidade, é possível trazer melhor performance.

Palavras-chave: *Multilayer Perceptron*, VHDL, FPGA.

Abstract

Neural networks are dedicated to implementing mathematical models that approach biological neurons, these implementations occur through the grouping of simple processing units through a learning algorithm. However, networks that have many neurons can make mathematical processing by way of software prolonged, however, a neural network can be trained in hardware using parallelism in several operations. In this context, FPGA devices can be useful as they allow synthesis through software tools and Hardware Description Language. In this work, the multilayer perceptron model is implemented in hardware using the VHDL language, so it was synthesized and recorded in the FPGA Cyclone II 2C35 device using the Altera DE2 development kit with some peripherals. The neural network has successfully trained the database using the proposed hardware and through the integration created with the computer it is possible to send several databases to the device and train them. The approximation error using a test base was found to be in harmony with the results in Java and the training time of the neural network was shorter according to the amount of logic elements used in the network. The designed hardware proved to be effective in what it proposes to solve. The behavior of the neural network in VHDL was similar to that of the network implemented in Java. Due to the limitation of logic blocks, some strategies had to be implemented to overcome the physical restriction of Cyclone II 2C35, however, using newer devices with greater capacity, it is possible to bring a better performance.

Keywords: Multilayer Perceptron, VHDL, FPGA.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

“O conselho da sabedoria é: procure obter sabedoria;
use tudo que você possui para adquirir entendimento.”
Provérbio de Salomão

Com a grande busca de se desenvolver sistemas que imitem a capacidade humana de aprender, raciocinar, evoluir e adaptar, a inteligência artifical é uma área da ciência da computação que se empenha em compreender o funcionamento do cérebro a fim de criar programas que possam produzir um comportamento inteligente para diversas aplicações (TEIXEIRA, 2019). O progresso da Inteligência Artificial gera produtos que se aplicam desde a indústria até o usuário de *smartphone*, seu uso cria impactos que levam a transformações econômicas e sociais (LANNA, 2018).

Ligada à inteligência artificial, o estudo de redes neurais se dedica a implementar modelos matemáticos que se aproximem à neurônios biológicos. Essas redes podem processar informações a partir do agrupamento de unidades simples de processamento através de um algoritmo de aprendizagem, dessa forma, o algoritmo responde um objetivo de projeto desejado (HAYKIN, 2007).

Num ambiente onde existem muitos neurônios, o processamento matemático via software pode se tornar demorado, contudo, uma rede neural pode ser treinada em hardware fazendo o uso de paralelismo em diversas operações (Hariprasath; Prabakar, 2012). Neste âmbito, os PLDs (*Programmable Logic Devices*, em Português, Dispositivos Lógicos Programáveis) podem ser úteis, pois permitem a sintetização através de ferramentas de software e Linguagem de Descrição de *Hardware* (HDL). Dentre os PLDs, encontra-se o dispositivo FPGA (*Field Programmable Gate Array*, em Português, Arranjo de Portas Programáveis em Campo), que é equivalente a um circuito integrado configurado a partir de uma HDL (KAPTANOGLU et al., 1999).

Alguns autores implementaram modelos de redes neurais embarcados em FPGA em diferentes pesquisas, como Restrepo et al. (2000) no trabalho *"A Networked FPGA-Based Hardware Implementation of a Neural Network Application"* e Hariprasath e Prabakar (2012), no trabalho, "*FPGA implementation of multilayer feed forward neural network architecture using VHDL*", trabalhos estes que serão devidamente explorados na seção 2.3. No presente trabalho, é implementada uma rede neural *Perceptron* Multicamadas em um dispositivo FPGA utilizando a placa Altera DE2 de forma a responder igualmente ao software em sua forma básica.

1.1 Justificativa

As redes neurais, através de sua estrutura paralelamente distribuída e de sua capacidade de produzir saídas adequadas para entradas que não estavam presentes durante o

treinamento, podem solucionar diversos problemas computacionais altamente intratáveis (HAYKIN, 2007).

Nesses modelos, são recorrentes as operações de soma e multiplicação, porém como a maioria dos *softwares* de redes neurais executam em arquiteturas sequenciais, não é possível executar simultaneamente essas operações. Portanto, implementações de *software* tendem a ser lentas, entretanto, através da implementação utilizando *hardware* é possível executar essas operações em paralelo deixando o treinamento mais rápido (Hariprasath; Prabakar, 2012).

Diminuir o tempo de treinamento de redes neurais, pode significar a redução de custos para determinado projeto. Dentre as variadas possibilidades, a linguagem VHDL (*Very High Speed Integrated Circuits*) mostra-se como uma boa alternativa para a implementação de redes neurais. Devido ser padronizada pela IEEE (*Institute of Eletrical and Eletronics Engineers*), esta linguagem permite a flexibilização em diversos dispositivos podendo atender a variados fabricantes e projetos de sistemas embarcados (D'AMORE, 2000).

1.2 Objetivos

O objetivo geral deste trabalho é implementar o modelo de rede neural artificial *perceptron* multicamadas em *hardware* utilizando a linguagem VHDL e sintetizá-lo em um dispositivo FPGA. No treinamento será utilizada uma base de dados de diagnóstico de câncer de mama disponibilizado publicamente pela UCI Machine Learning Repository (1998). Para realizar os testes, o *hardware* descrito será sintetizado e gravado em um *chip* utilizando o *kit* de desenvolvimento Altera DE2 juntamente com alguns periféricos. Também, objetivam-se mais especificamente:

1. Desenvolver o *software* em Java para base de comparação com o hardware via VHDL.
2. Projetar a representação binária de números reais com ponto fixo e também projetar as operações multiplicação e divisão de números reais com ponto fixo em VHDL.
3. Realizar testes de simulação do *hardware* descrito utilizando o *software* ModelSim conforme cada etapa do trabalho.
4. Desenvolver o modelo de rede neural artificial *perceptron* de multicamadas em VHDL.
5. Desenvolver uma interface para inserção da base de dados de câncer de mama a ser treinadas na rede neural.
6. Realizar testes no treinamento de bases de dados e avaliar o desempenho comparando-os a *softwares* que executam em arquiteturas sequenciais.

2 Revisão Bibliográfica

“A civilização avança ampliando o número de operações importantes que podemos realizar sem pensar nelas”.
Alfred North Whitehead

O principal objetivo deste capítulo é apresentar um embasamento da literatura já publicada sobre o mesmo tema e expor as tecnologias, conceitos e modelos utilizados ao longo do trabalho.

2.1 Sistemas Digitais

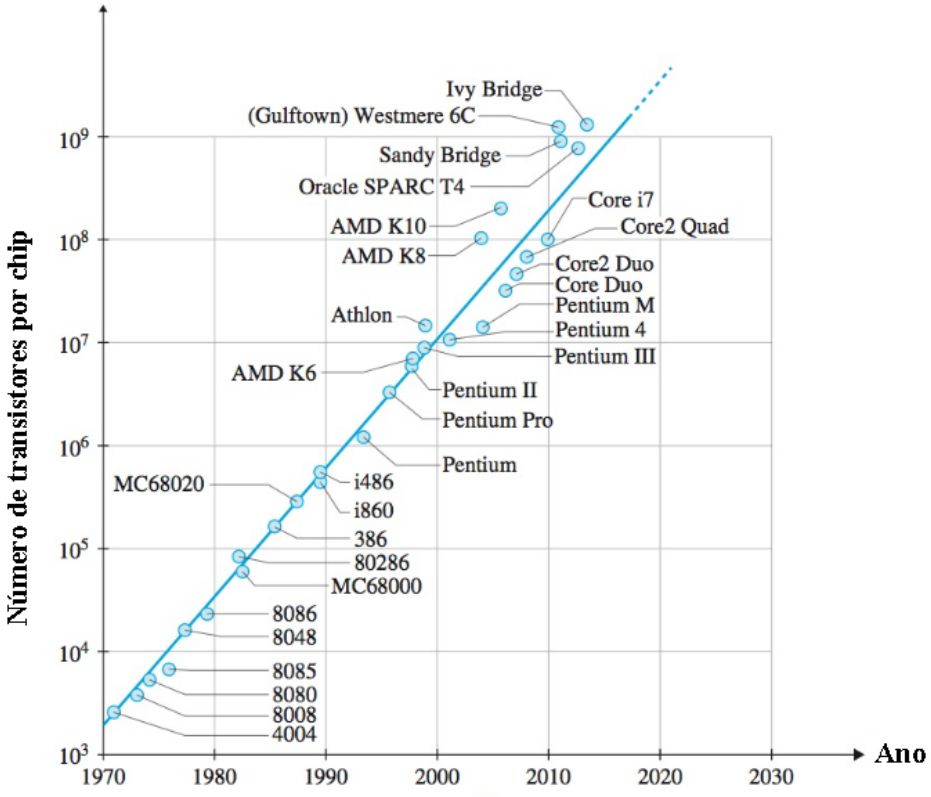
Segundo Tocci, Widmer e Moss (2010), um sistema digital é a combinação de dispositivos projetados para tratar informações lógicas ou grandezas físicas de forma discreta. Esses dispositivos são geralmente eletrônicos e implementados através de circuitos digitais que trabalham com sinais de tensão binários gerando sinais de saída a partir de sinais de entrada. Podendo assim, realizar funções que podem ser descritas pela álgebra booleana. Dentre os sistemas digitais mais comuns, estão os computadores, *smartphones*, relógios digitais, televisões, dentre outros.

2.1.1 Circuitos Integrados Digitais

O primeiro circuito integrado (CI) foi desenvolvido em 1958 simultaneamente por Kilby na Texas Instruments e Noyce e Moore na Fairchild Semiconductor, marcando o início de uma nova fase na revolução microeletrônica. Um CI, consiste em um circuito eletrônico que introduz diversos componentes eletrônicos, principalmente transistores, diodos, resistores e capacitores em um *chip* de silício.

Desde então, iniciou-se um grande esforço para o desenvolvimento de circuitos integrados de maior densidade com menor gasto de energia. Gordon E. Moore, fez uma observação em que ele afirma que o número de transistores dentro de um circuito integrado seria duplicado a cada 24 meses na medida em que também minimizaria o custo de um transistor. A Figura 1 mostra o crescimento do número de transistores com o passar dos anos (RASHID, 2017).

Figura 1 – Lei de Moore



Fonte: Rashid (2017), traduzida pelo autor

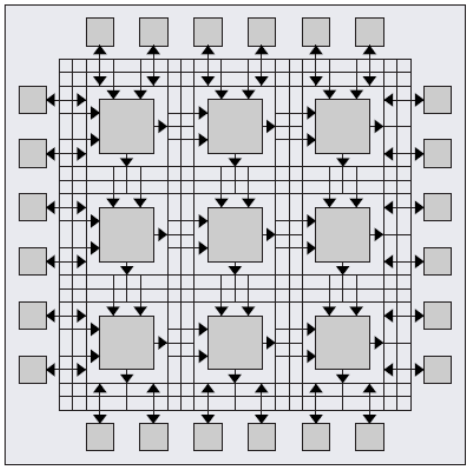
A tecnologia de semicondutores desenvolveu-se continuamente ao longo dos anos para reduzir o tamanho, o custo e o consumo de energia de dispositivos eletrônicos.

2.1.2 FPGA

O FPGA(*Field Programmable Gate Array*) pertence a uma família de dispositivos conhecidos como dispositivos lógicos programáveis. Esses dispositivos permitem que se crie um circuito digital, e o dispositivo se tornará esse circuito. Isso funciona configurando pequenos blocos para que forme funções lógicas e conecte-as para implementar seu projeto (RAJEWSKI, 2017).

Os blocos lógicos contêm elementos de processamento para executar lógica combinacional simples, bem como *flip-flops* para implementar lógica sequencial. A estrutura geral de roteamento permite fiação arbitrária, para que os elementos lógicos possam ser conectados da maneira desejada. A figura 2 ilustra a estrutura interna de um dispositivo FPGA. Devido a essa generalidade e flexibilidade, um FPGA pode implementar circuitos muito complexos (HAUCK; DEHON, 2010).

Figura 2 – Uma visão abstrata de um FPGA; as células lógicas são embutidas em uma estrutura de roteamento.



Fonte: Hauck e DeHon (2010)

Com o avanço dos dispositivos FPGA de grande capacidade, circuitos digitais de grande complexidade podem ser implementados em um *chip* com facilidade e rapidez. Além disso, como os chips FPGA são regraváveis, você pode usar o mesmo chip FPGA repetidamente para implementar circuitos diferentes (HWANG, 2018).

2.1.3 VHDL

VHDL (*VHSIC Hardware Description Language*) é uma linguagem de descrição de hardware padronizada pela IEEE. Esta linguagem se desenvolveu a partir da necessidade de uma ferramenta de projeto e documentação padrão para o projeto de circuitos integrados de alta velocidade do Departamento de Defesa dos Estados Unidos da América (DARPA) (D'AMORE, 2000).

Apesar de ser uma linguagem de descrição de hardware, a linguagem permite abstrações que facilitam a forma de se descrever um circuito. Existem três estilos de codificação em VHDL, a de mais baixo nível é a estrutural, que consiste em instanciar ou chamar explicitamente instâncias específicas de primitivos ou módulos definidos anteriormente e conectá-los. Embora todo um sistema possa ser desenvolvido dessa maneira, esse método consome muito tempo.

Com um pouco mais de abstração, RTL (*Register Transfer Level*, em português, Nível de Transferência de Registradores) refere-se a um estilo de codificação que define como as operações entre registradores ocorrerão, de forma a não precisar ser instanciado um componente que faz certa operação, mas podendo ser descrito a partir de equações.

No próximo nível de abstração está a codificação comportamental. Este estilo se aproxima mais das instruções de software, como "*loops*", "*if* e *else*"e "*case when*". Embora sejam diferentes estilos, é possível mesclá-los em um único projeto, fazendo combinações para otimizar o projeto (KAFIG, 2011).

2.2 Redes Neurais Artificiais

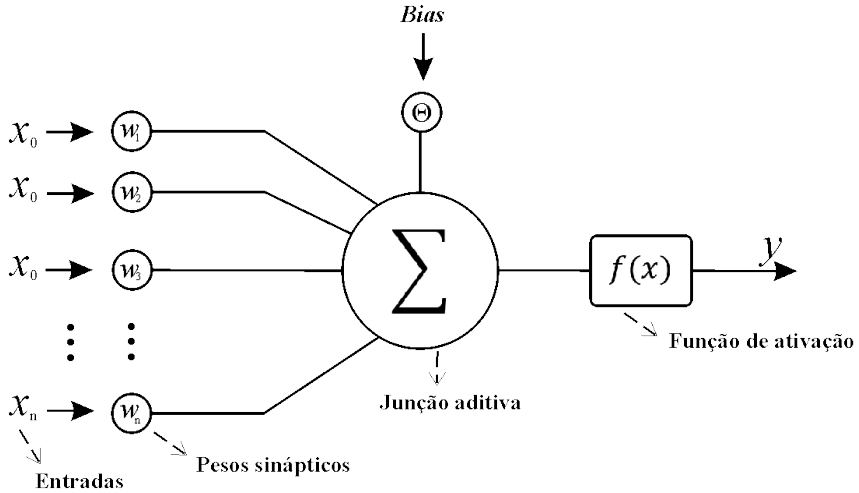
A inteligência artificial é uma área da ciência da computação que se dedica a desenvolver algoritmos que tentam reproduzir o comportamento do cérebro humano em computadores. Tarefas que são difíceis para um ser humano, como cálculos pesados e memorização de grandes volumes de dados são facilmente executados por computadores, enquanto tarefas que os humanos são naturalmente capazes de realizarem rapidamente, como reconhecer objetos, criatividade, invenção, compreensão do discurso, são difíceis para computadores (LIVSHIN, 2019).

Neste contexto, uma rede neural artificial é uma arquitetura de inteligência artificial que imita esquematicamente uma rede cerebral humana, que consiste em camadas de neurônios conectadas entre si (LIVSHIN, 2019). Haykin (2007) define redes neurais como:

> Uma rede neural é um processador maciçamente paralelo distribuído constituído de unidades de processamento simples, que têm a propensão natural para armazenar conhecimento experimental e torná-lo disponível para o uso. Ela se assemelha ao cérebro em dois aspectos:
>
> 1. O conhecimento é adquirido pela rede a partir de seu ambiente através de um processo de aprendizagem.
>
> 2. Forças de conexão entre neurônios, conhecidos como pesos sinápticos, são utiizadas para armazenar o conhecimento adquirido. (HAYKIN, 2007, p. 28)

O neurônio é uma unidade essencial deste processador. A partir dele são feitas as operações básicas que permitem o funcionamento de uma rede neural. A figura 3 mostra o modelo de um neurônio.

Figura 3 – Modelo não-linear de um neurônio



Fonte: Haykin (2007); adaptado pelo autor.

A partir do modelo ilustrado, é possível identificar alguns elementos básicos de um neurônio artificial. Este possui um conjunto de entradas conectados a um conjunto de sinapses, cada uma delas caracterizada por um peso. Cada entrada é multiplicada pelo seu respectivo peso sináptico associado e então na junção aditiva são somados os resultados destas

operações. Por último, uma função de ativação restringe a amplitude da saída de um neurônio (HAYKIN, 2007).

2.2.1 *Perceptron*

O *perceptron* é a construção mais simples de uma rede neural utilizada para classificar padrões linearmente separáveis. O mesmo foi proposto por Rosenblatt em 1959 e consiste em um ou mais neurônios de saída, dos quais cada saída está conectada a um fator de ponderação(peso) em cada entrada. A saída y do neurônio se dá pela soma de todas as entradas ponderadas pelos pesos aplicados a uma função de ativação, conforme é possível observar através da equação a seguir:

$$y = f\left(\sum_{i=1}^{n}(w_i x_i) + \Theta\right) \tag{2.1}$$

onde,

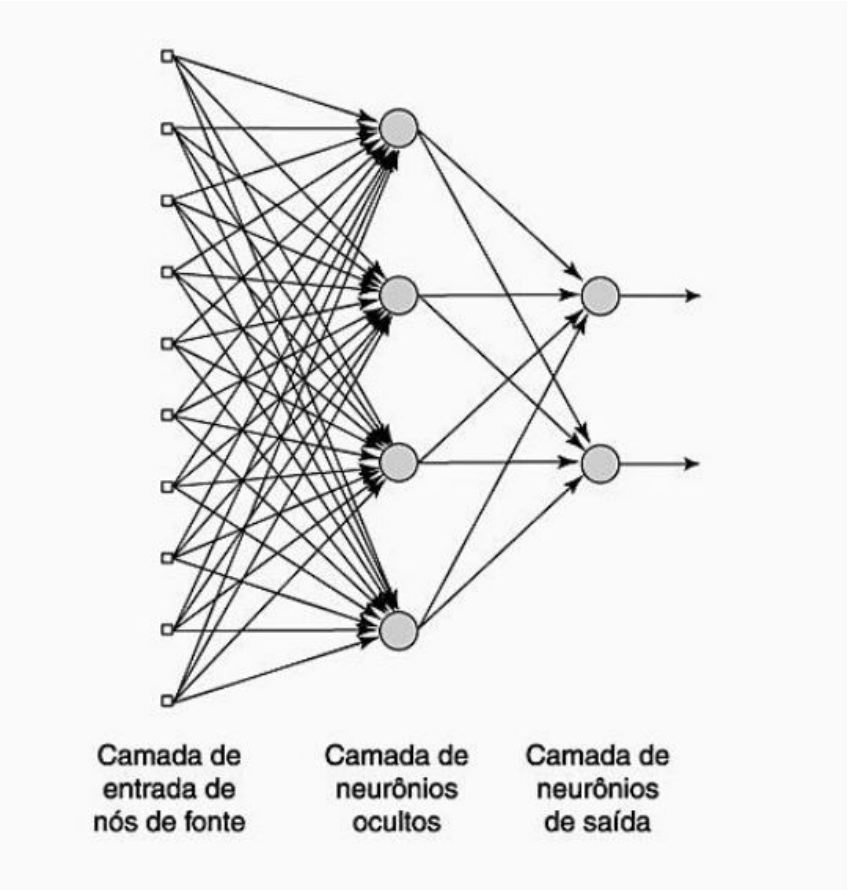
- y equivale à saída do neurônio.
- $f(s)$ representa a função de ativação utilizada.
- x são as entradas do neurônio.
- w são os pesos sinápticos.
- n a quantidade de entradas do neurônio.
- Θ representa o bias.

O *perceptron* construído utilizando-se apenas um único neurônio é limitado a classificar padrões com apenas dois tipos de saídas. Porém, se forem incluídos mais neurônios conectados às mesmas entradas, é possível realizar a classificação com mais padrões. No entanto, as classes devem ser linearmente separáveis para que o *perceptron* funcione corretamente (HAYKIN, 2007).

2.2.2 *Multilayer Perceptron*

Uma evolução do modelo do perceptron é o *Multilayer Perceptron* que trata-se da generalização do *perceptron* simples adicionando camadas ocultas de neurônios. Conforme é possível identificar na figura 4, os nós da primeira camada fornecem a entrada para a primeira camada oculta, e os sinais da saída da primeira camada oculta servem como entrada da segunda camada oculta e assim por diante (HAYKIN, 2007).

Figura 4 – Rede Neural *Multilayer Perceptron* totalmente conectada



Fonte: Haykin (2007)

A partir deste modelo é possível realizar a categorização de bases que possuem classes não linearmente separáveis, aumentando assim a abrangência de variedade de bases possíveis de serem treinadas (KRÖSE et al., 1993).

2.2.3 Funções de Ativação

As funções de ativação (representadas por φ) possuem um papel fundamental no *perceptron*, elas definem a saída produzida a partir da junção aditiva de um neurônio. Seu objetivo é produzir em sua saída valores monótonos entre 0 e 1 (ou entre -1 e 1) para todo o domínio da função (HAYKIN, 2007).

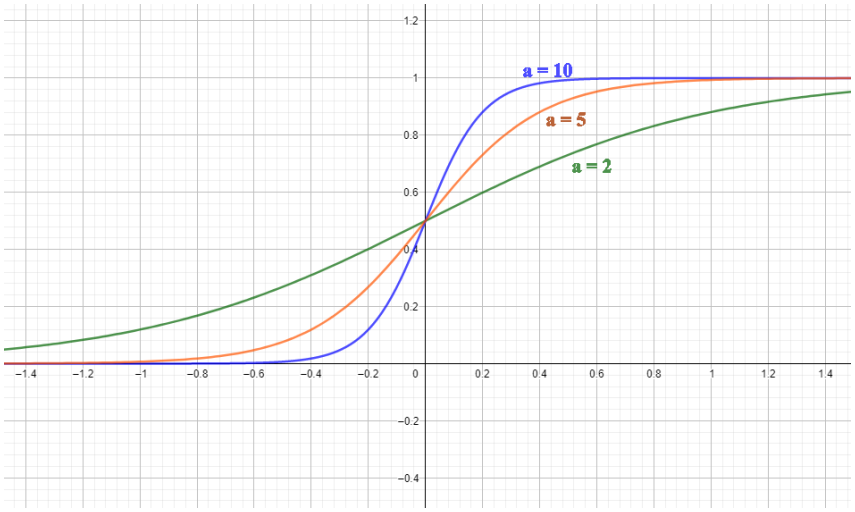
2.2.3.1 Sigmóide

A forma mais comum de função de ativação é a sigmóide. Um exemplo de função sigmóide, é a função logística:

$$\varphi(v) = \frac{1}{1 + exp(-av))} \tag{2.2}$$

Esta função gera um gráfico em forma de s, e conforme varia-se o parâmetro a, muda-se a inclinação da curva (conforme é possível visualizar no gráfico da figura 5). No limite, quando o parâmetro a aproxima-se do infinito, a função assume o valor de 0 ou 1.

Figura 5 – Gráfico da função sigmóide variando o parâmetro a.



Fonte: O Autor

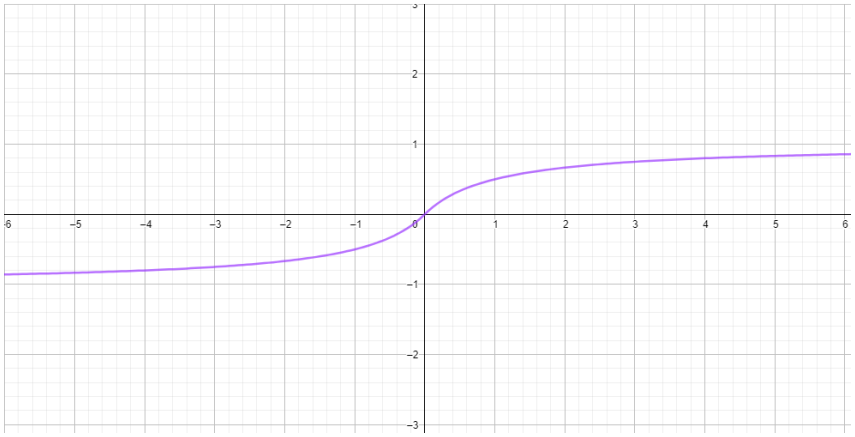
2.2.3.2 Elliot

Buscando diminuir o custo computacional da sigmóide, Elliott (1993), propôs uma nova função de ativação que fazia uso de um número muito menor de operações do que as funções exponenciais. A função proposta, satisfaz uma equação diferencial simples generalizando a equação logística. A função proposta foi:

$$\varphi(v) = \frac{v}{1 + |v|} \tag{2.3}$$

Conforme é possível observar pelo gráfico da figura 6, a função de Elliot tem uma aproximação da sigmóide e gera valores entre -1 e 1. Além disso, ela é diferenciável em todo seu domínio (ELLIOTT, 1993).

Figura 6 – Gráfico da Função de Elliot.



Fonte: O Autor

2.2.4 Algoritmo Error Back-Propagation

O *Multilayer Perceptron* é capaz de resolver diversos problemas através de seu treinamento de forma supervisionada utilizando o algoritmo *error back-propagation* (Em português, Algoritmo de Retropropagação de Erro) fazendo uso da regra de aprendizagem por correção de erro. Esta aprendizagem consiste em dois passos: um passo para frente, a propagação e um passo para trás, a retropropagação (HAYKIN, 2007).

Na propagação, a entrada é aplicada aos nós da rede e seu efeito se propaga em cada uma de suas camadas. Por fim, será produzido um sinal de saída que é a resposta da rede àquela determinada entrada. Durante a propagação, os pesos sinápticos se mantem fixos e então na retropropagação, estes pesos são ajustados de acordo com a regra de correção de erro (HAYKIN, 2007).

2.3 Trabalhos Relacionados

Durante a década de 1980 e início de 1990, houve um esforço significativo no *design* e implementação de neurocomputadores em *hardware*. No entanto, as *gate-arrays* (em português, arranjo de portas, é um chip pré-fabricado com transistores sem uma função predeterminada que após adicionada outra camada de circuitos, recebem uma função específica) desse período não eram grandes nem rápidas o suficiente para aplicações sérias de redes neurais artificiais. Porém, atualmente a capacidade e o desempenho dos FPGAs são tais que se mostram como uma alternativa realista. Consequentemente, redes neurais baseadas em FPGAs agora são uma proposição muito mais prática do que no passado (OMONDI; RAJAPAKSE, 2006). Esta seção apresenta três trabalhos em que são desenvolvidas redes neurais descritas em *hardware*.

2.3.1 Uma implementação em rede de *hardware* baseado em FPGA de uma aplicação de rede neural

No trabalho *"A Networked FPGA-Based Hardware Implementation of a Neural Network Application"*, Restrepo et al. (2000) implementaram uma rede neural artificial utilizando a arquitetura FAST (*Flexible Adaptable-Size Topology*, em português, Topologia de Tamanho Adaptável e Flexível) em um dispositivo FPGA.

A rede neural FAST é uma rede de aprendizado não supervisionado com topologia de tamanho adaptável e flexível. Esta rede é totalmente conectada e consiste em duas camadas: uma de entrada e outra de saída. O tamanho da rede aumenta com a adição de um novo neurônio para a camada de saída quando um vetor de entrada suficientemente distinto é encontrado e diminui ao excluir um neurônio operacional através da aplicação de desativação probabilística (RESTREPO et al., 2000).

A implementação final resumiu-se em uma rede de até oito placas Labomat 3 (Placa que utiliza tecnologia FPGA fabricada no laboratório dos autores). Cada placa contém dois neurônios FAST que suportam o processamento de 8 *bits* e vetores bidimensionais utilizando uma frequência máxima de aproximadamente 10 MHz.

Os autores concluíram que os experimentos feitos utilizando a rede neural implementada em VHDL indicaram que foi possível atingir alta performance em atividades de agrupamento (também conhecido como *clustering*). Como exemplo, aplicaram esta rede neural em um problema de segmentação e reconhecimento de imagens. Os resultados obtidos assemelham-se muito aos obtidos com outros algoritmos neurais que não são adequados para implementação de *hardware*.

2.3.2 Implementação de uma Arquitetura de Rede *Multilayer Feed Foward* em FPGA utilizando VHDL

Hariprasath e Prabakar (2012), no trabalho, "*FPGA implementation of multilayer feed forward neural network architecture using VHDL*"implementaram em *hardware* uma rede neural *multilayer feed foward* utilizando FPGA. Nesta implementação, utilizaram representação de dados tendo números binários sinalizados com ponto fixo no formato 1-6-9, em que o primeiro bit é reservado para o sinal do número, os próximos 6 representam a parte inteira e os últimos 9, a parte fracionária do número.

Para o treinamento, os autores utilizaram MATLAB e após obterem os pesos sinápticos, os gravaram em uma memória RAM. Tendo os pesos determinados, um somador foi usado para acumular os produtos entre o sinal de entrada dos neurônios de camada anterior e os pesos sinápticos correspondentes. Para efetuar a multiplicação digital dos números, o autor fez uso do algoritmo de Booth, que se mostrou como um meio eficiente de se multiplicar números sinalizados expressos na notação de complemento de dois (HARIPRASATH; PRABAKAR, 2012).

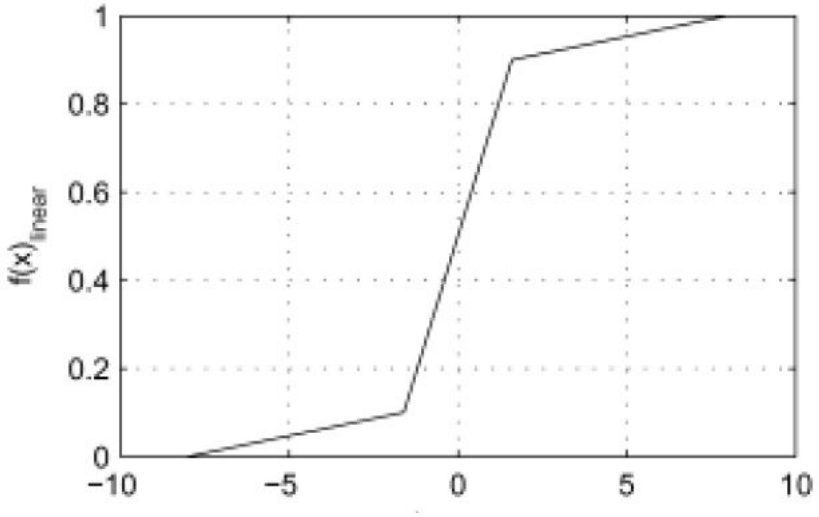
Com o objetivo de minimizar o *hardware* utilizado nas operações, é utilizada uma aproximação linear por partes da função sigmóide como função de ativação. A equação a seguir,

descreve a função utilizada:

$$f(x) = \left\{ \begin{array}{ccc} 0, & se & x \leq -8 \\ \frac{8-|x|}{64}, & se & -8 < x \leq 1.6 \\ \frac{x}{4} + 0.5, & se & |x| < 1.6 \\ 1 - \frac{8-|x|}{64}, & se & 1.6 \leq x < 8 \\ 1, & se & x > 8 \end{array} \right\} \tag{2.4}$$

Conforme é possível observar no gráfico da figura 9, esta função limita a saída na faixa entre 0 e 1.

Figura 7 – Gráfico da aproximação linear da função sigmóide.



Fonte: Hariprasath e Prabakar (2012)

Os autores concluíram que a rede apresentada reduz os recursos necessários para a implementação e também o tempo de execução principalmente pelo fato de o processamento de dados em todos os neurônios da camada serem realizados em paralelo. A função de ativação usada utiliza um número menor de recursos em comparação ao projeto lógico sequencial (HARIPRASATH; PRABAKAR, 2012).

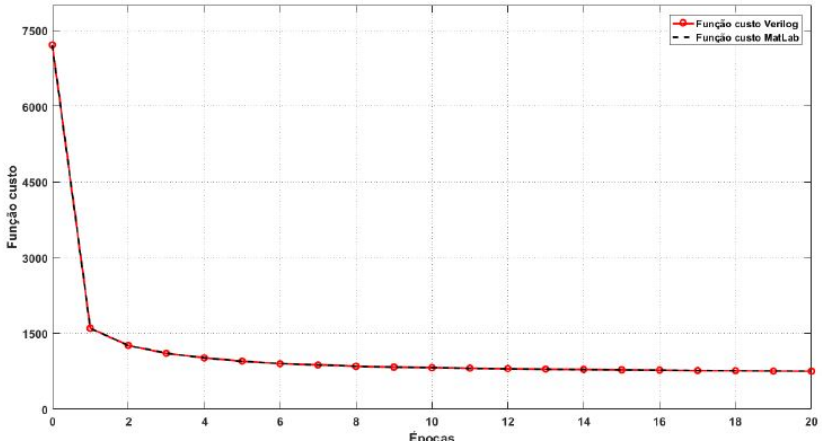
2.3.3 Treinamento de Redes Neurais com Arquitetura *Multilayer Perceptron* em FPGA.

No trabalho "Treinamento de Redes Neurais com Arquitetura *Multilayer Perceptron* em FPGA."(SOUZA et al., 2019), apresentaram uma implementação de um treinamento de redes neurais em um dispositivo FPGA utilizando a linguagem Verilog e comparou os resultados com o algoritmo executando em Python e MatLab.

Em Matlab, utilizou a mesma precisão e algoritmos que no dispositivo FPGA, a figura mostra o gráfico da função custo produzidas no MatLab e na simulação utilizando ModelSim. Seus resultados mostram que levando em consideração os erros de quantização devido à quantidade de *bits* utilizados para o treinamento, os resultados obtidos em Python e em Verilog

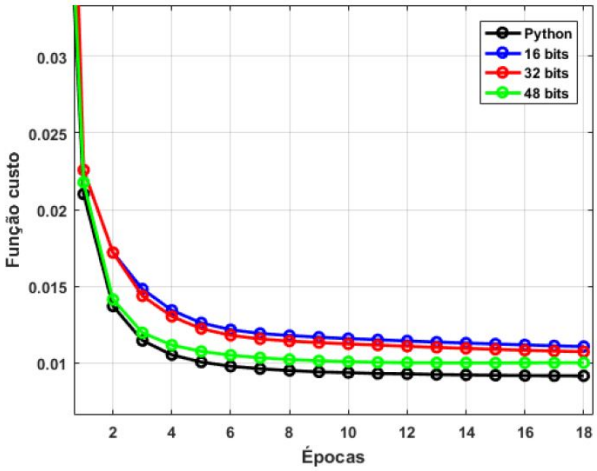
foram bem próximos. A figura mostra a comparação das funções custo em Python e Matlab com diferentes precisões.

Figura 8 – Gráfico das funções custo produzidas no MatLab e no ModelSim.



Fonte: Souza et al. (2019)

Figura 9 – Comparação das funções custo de Python e MatLab com diferentes precisões.



Fonte: Souza et al. (2019)

Os autores concluíram que o treinamento utilizando MatLab e Verilog foi equivalente aos resultados obtidos em Python, porém não identicos devido ao erro de quantização do ponto fixo utilizado no treinamento (SOUZA et al., 2019).

3 Materiais e Métodos

“Em uma questão importante,
nenhum detalhe é pequeno”.
Provérbio Francês

Segundo Gerhardt e Silveira (2009), no que se refere à natureza, esta pesquisa é classificada como aplicada visto que se trata de um problema particular com aplicações práticas. Quanto à abordagem, pode ser classificada como quantitativa uma vez que a análise de seus resultados se baseiam em quantidades. Pode ser classificada como descritiva, dado que busca dados objetivos do objeto de estudo. Quanto aos procedimentos, pode ser classificada como um estudo de caso, uma vez que se trata de uma pesquisa com um cenário bem definido e específico.

3.1 Materiais

A fim de alcançar o objetivo proposto deste trabalho, foram utilizados os seguintes equipamentos:

- Computador
  - Modelo: Lenovo IdeaPad 320
  - Processador: Intel Core i5-7200 CPU @ 2.50GHz 2.71GHz
  - Memória RAM: 8GB DDR4 2133 MHz
  - Sistema Operacional: Microsoft Windows 10 Pro de 64 bits
- Placa de Desenvolvimento Arduino Mega
  - Modelo:ARDUINO MEGA 2560 REV3 (Figura 10)
  - *Chip*: ATmega2560:
    - Entradas/Saídas Digitais: 54, 15 destes suportam PWM
    - Entradas/Saídas Digitais: 16
    - Tensões Suportadas (Volts): 7 ;
    - *Clock* Máximo: 16 MHz
  - Memória *Flash*: 256 KB
  - SRAM: 8 KB
  - EEPROM: 4 KB

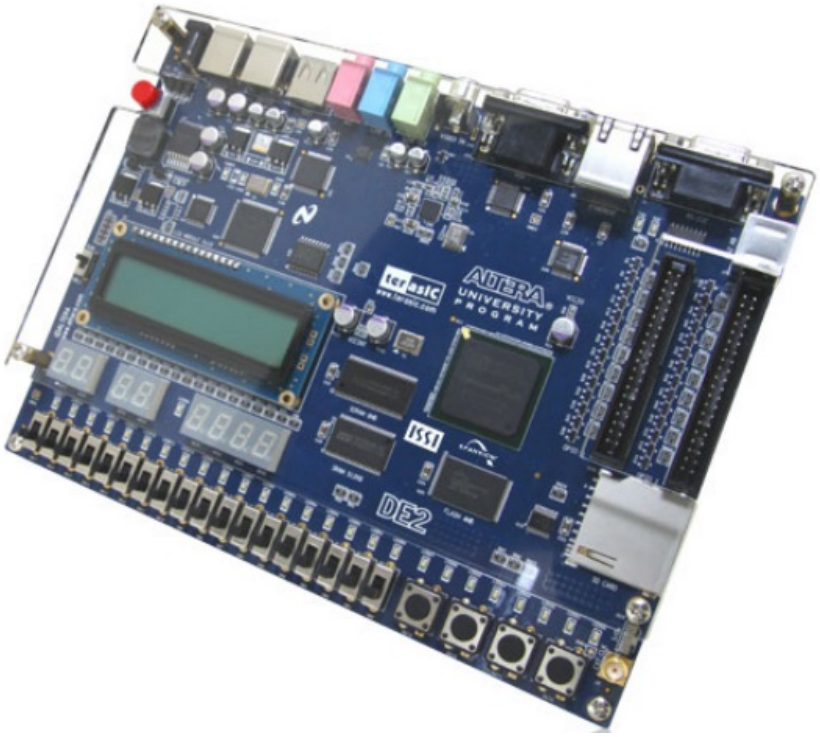
Figura 10 – Placa de desenvolvimento Arduino Mega, utilizada durante o desenvolvimento do projeto.

Fonte: Arduino (2018)

- Placa de Desenvolvimento FPGA
  - Modelo: Altera DE2 (Figura 11)
  - Chip: Intel Altera Cyclone II:
    - Modelo: EP2C35F672C6
    - Blocos Lógicos: 33,216 LEs
    - Quantidade de Entradas e Saídas para usuário: 475
    - PLL (*Phased-locked Loop*) de uso geral: 4;
    - Tensões Suportadas (Volts): 1.2, 1.5, 1.8, 2.5, 3.3;
    - *Clock* Máximo: @ 260 MHz
    - Blocos de memória: 484 Kb
  - Memória SRAM: ISSI IS61LV25616
    - Tempo de Acesso: 10, 12 e 15ns
    - Tensão de Alimentação: 3,3V
    - Operação completamente estática: não necessário clock ou atualização
    - Interface compatível com nível TTL
  - Periféricos:
    - 1x Botão de alimentação
    - 1x Botão de reset
    - 4x Botões de uso geral
    - 18x *Switches* de uso geral

* 18x LEDs (*Light Emitting Diode*) de cor vermelha
* 9x LEDs (*Light Emitting Diode*) de cor verde
* Modulo Display LCD
* 8x *Displays* de 7 Segmentos
* Entre outros recursos

Figura 11 – Placa de desenvolvimento Altera DE2, utilizada durante o desenvolvimento do projeto.



Fonte: Altera (2006)

- Ambientes de Desenvolvimento
  - Altera Quartus 12 *Service Pack 2 Web Edition*
  - Altera ModelSim SE 10.5
  - NetBeans 8.1
- Linguagens de Desenvolvimento
  - Oracle Java
  - VHDL 2008
  - C

3.2 Procedimentos metodológicos

Para a realização deste trabalho, o projeto foi dividido em cinco partes: a implementação do *Perceptron* Multicamadas em Java, a implementação do mesmo algoritmo em VHDL,

a criação de uma interface para inserção de dados na memória SRAM, o desenvolvimento de um software para conversão dos dados do *dataset* e a integração das etapas anteriores em um único circuito.

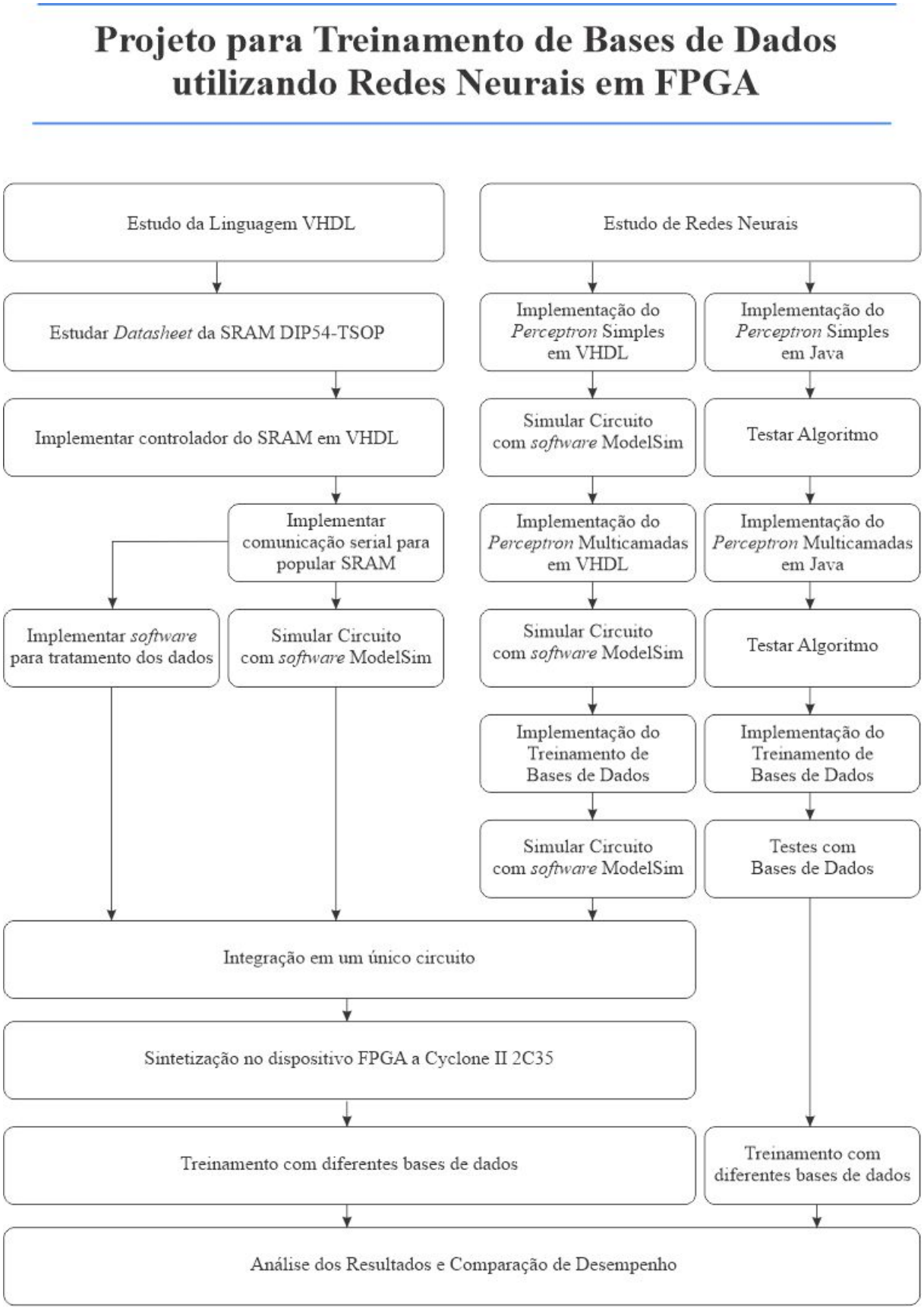
O desenvolvimento do algoritmo *Multilayer Perceptron* em Java assim como em VHDL, necessitaram de uma revisão na literatura sobre o assunto, e progrediram paralelamente. Iniciou-se primeiramente o desenvolvimento do *Perceptron* simples, e após testado e simulado, foi desenvolvido o *Multilayer Perceptron* que novamente passou por testes e simulações para ser validado.

Com o propósito de armazenar os dados da base a ser treinada, utilizou-se a memória SRAM DIP54-TSOP também disponível no kit de desenvolvimento Altera DE2. Fez-se necessário também estudar o *datasheet* e implementar um controlador para a memória SRAM. Além disso, para a comunicação do computador com a memória com o objetivo de transferir os dados da base a ser treinada, utilizou-se um Arduino Mega para criar a interface de comunicação.

Além disso, para tratar os dados que serão inseridos na memória SRAm, convertê-los para binario e fazer o correto endereçamento, observou-se a necessidade de se criar um software para realizar este controle.

Após finalizadas as etapas descritas acimas e todos controladores prontos, então fez-se necessário a integração de todos os módulos criados separadamente em um único circuito para o treinamento da base de dados. A figura 12, resume as etapas necessárias para o desenvolvimento deste trabalho.

Figura 12 – Fluxograma das etapas deste trabalho.



Fonte: O Autor

4 Implementação do *Multilayer Perceptron* em *Hardware*

"Nós só podemos ver um pouco do futuro,
mas o suficiente para perceber que há muito a fazer."
Alan Turing

Ao se projetar um *hardware*, é preciso levar em consideração o projeto como um todo: os componentes a serem utilizados, as comunicações necessárias, os protocolos a serem adotados, dentre outros. Neste capítulo será apresentado o desenvolvimento do presente trabalho. O progresso começou com o desenvolvimento do *perceptron* simples, então após isso o desenvolvimento do *multilayer perceptron* e por conseguinte as implementações necessárias para importação e treinamento de uma base de dados externa.

4.1 *Multilayer Perceptron* em Java

A primeira etapa deste trabalho trata-se da criação da rede neural artificial em Java, pois a partir dela será baseado todo o restante do projeto. Primeiramente foi criado a rede neural *perceptron* simples, e logo em seguida foi criada a rede *multilayer perceptron*. Ambas foram testadas e validadas com diferentes bases de dados para que pudessem servir de base para a criação do algoritmo em VHDL.

4.2 *Perceptron* Simples em VHDL

Por se tratar de um algoritmo não muito trivial, a implementação em VHDL se deu por partes. Primeiramente, foi implementado apenas a execução da rede neural e feitos os primeiros testes, tão somente depois foi implementado o treinamento.

4.2.1 Divisão de Números Reais

Para que se execute o algoritmo do *perceptron*, uma série de operações matemáticas se fazem necessárias, dentre elas a divisão. Geralmente, a implementação da divisão requer um circuito lógico muito complexo e, por essa razão, evita-se o uso deste operador, porém, há casos em que se faz necessário.

Para casos como esse, a biblioteca *ieee.numeric_std* fornece funções aritméticas para vetores. Neste caso em específico é possível fazer divisões inteiras de números binários utilizando o operador "/". Porém, neste projeto se fará necessário a utilização de números reais e não apenas de números inteiros, para isto, foi projetada uma representação de ponto fixo para estes números. Sendo assim, o primeiro *bit* ficou reservado para o sinal, os próximos 5 *bits*

mais significativos para parte inteira, e os demais 26 para a parte fracionária, utilizando assim 32 *bits* no total. A figura 13 ilustra como foram organizadas as *words*.

Figura 13 – Organização da *word* utilizada para representar os dados.

Fonte: O Autor

Visto que foi utilizada representação binária por ponto fixo, para trazer mais precisão às operações de divisão, estas foram feitas utilizando números de 64 *bits*. O pseudo código a seguir, ilustra a forma como foram feitas estas operações: Considere f a quantidade de bits reservada para a parte fracionária do número, a o dividendo, b o divisor, e q o quociente.

```
1: function DIVISAO(a, b, f)
2:     a ← a << 32
3:     q ← a/b
4:     q ← q >> (32 − f)
5: end function
```

Com estas operações, é possível garantir que as representações de ponto fixo se mantenham adequadas após as operações de divisão.

4.2.2 Função de Ativação

A função de ativação é uma parte essencial de uma rede neural, é ela que padroniza os resultados entre dois extremos, no trabalho em questão, entre zero e um. A função de ativação mais utilizada é a sigmóide, porém, por ter operações de potenciação seriam muito custosas para implementação em *hardware*, então optou-se por buscar algumas alternativas.

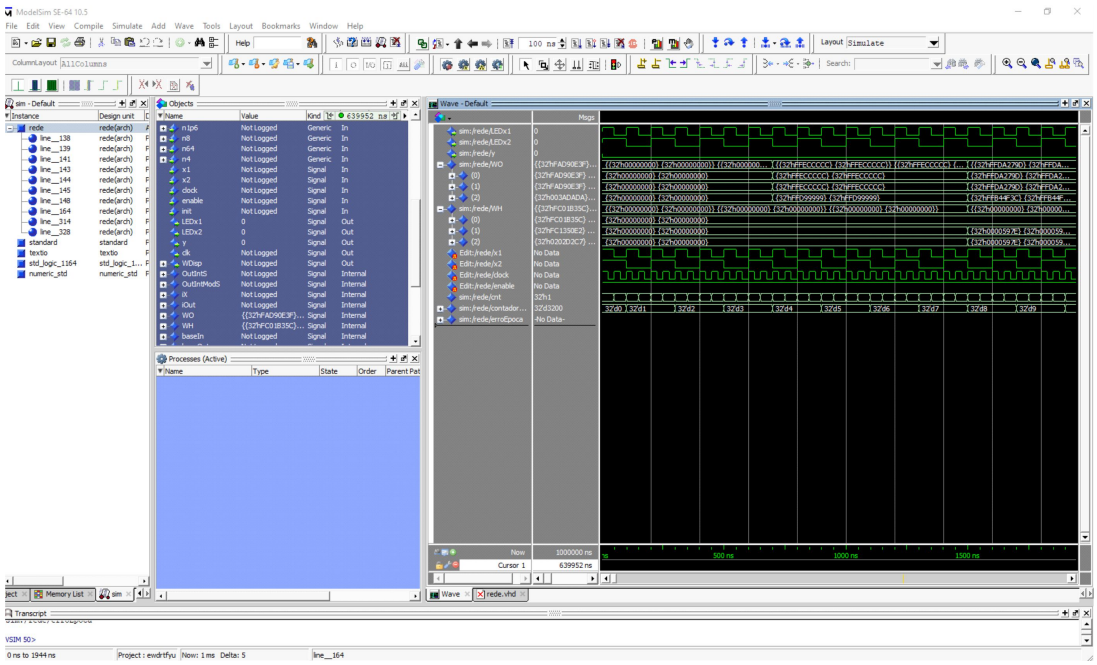
A primeira alternativa a se analisar, tratava-se da função de ativação proposta por Elliott (1993), e possui um custo para implementação bem menor que a função logística conforme descrito na seção 2.2.3 deste trabalho. Porém, ao revisar trabalhos mais recentes, foi possível encontrar soluções com um custo mais baixo, como demonstrado no trabalho de Hariprasath e Prabakar (2012) explanado na seção 2.3.2 deste trabalho. Hariprasath e Prabakar (2012)

4.2.3 Treinamento

Após implementada a função de ativação e o perceptron simples, o mesmo foi sintetizado no dispositivo FPGA Cyclone II 2C35 e testado com entradas no próprio código e pesos de uma rede já treinada anteriormente.

Após os devidos testes e validação dos resultados, foi implementado o treinamento da rede neural que após a sua conclusão, foi simulado no software ModelSim e também sintetizado no dispositivo FPGA citado. A figura 14 mostra a captura de tela dos testes realizados utilizando o *software* ModelSim.

Figura 14 – Captura de tela do software ModelSim durante os testes.



Fonte: O Autor

4.3 Multilayer Perceptron em VHDL

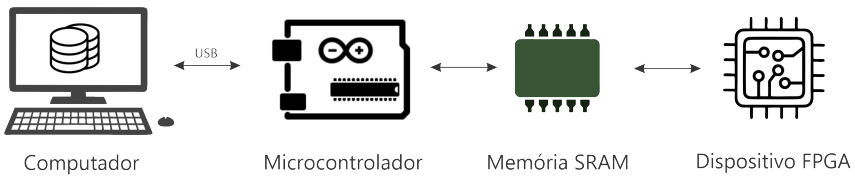
Após ser implementado o *perceptron* simples, iniciou-se o desenvolvimento do *multilayer perceptron*. Além de adicionar mais camadas de neurônios, é necessário também implementar o algoritmo de retropropagação. Este algoritmo cria uma interdependência dos resultados da camada oculta com a camada de saída, o que torna necessário algumas etapas a mais no algoritmo para o treinamento da base.

Após implementado, o algoritmo foi testado no software ModelSim e sintetizado no dispositivo FPGA disponível na placa da Altera DE2. Os testes se deram utilizando como base de dados as entradas e saídas de uma porta lógica E.

4.4 Implementação na Placa Altera DE2

Este projeto foi dividido em várias etapas a fim de que cada uma separadamente cumpra uma função diferente, porém com o objetivo de treinar uma base de dados e obter seus pesos finais. A figura 15 ilustra como diferentes partes do projeto irão comunicar entre si para desempenhar um papel maior.

Figura 15 – Esquema de comunicações entre diferentes partes do projeto.



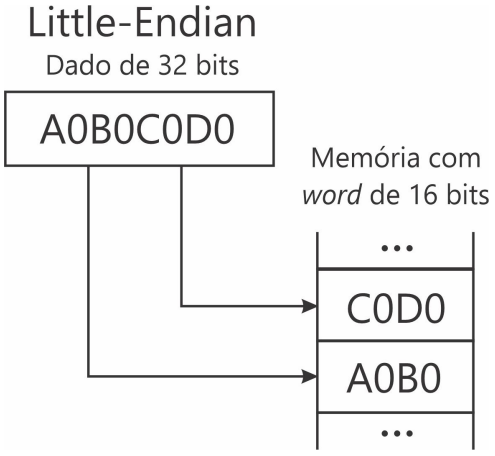
Fonte: O Autor

Em um computador estará armazenado a base de dados a ser treinada, que através de uma conexão USB, um microcontrolador distribuirá os dados em uma memória SRAM. Esta por sua vez, também estará conectada ao dispositivo FPGA que terá acesso aos mesmos dados podendo assim realizar o treinamento. Terminado o treinamento, o dispositivo FPGA salvará os pesos obtidos na memória, que será acessada pelo microcontrolador que enviará estes dados de volta ao computador.

4.5 Memória SRAM

O *kit* de desenvolvimento Altera DE2, dispões de alguns dispositivos de memória, dentre eles, uma memória SRAM de 512KB. Trata-se do *chip* IS61LV25616, uma RAM estática de alta velocidade organizada em 262.144 *words* de 16 bits. Como as *words* projetadas possuem 32 bits, elas foram divididas na memória em pares. A representação escolhida foi a *Little-Endian*, ou seja, foram posicionados primeiramente os bytes menos significativos e após, os bytes mais significativos. A figura 16 mostra o esquema utilizado.

Figura 16 – Esquema da representação *Little-Endian*.



Fonte: O Autor

Para o *dataset*, foi reservado do endereço 0 ao endereço 261.887 e os demais endereços foram reservados para guardarem os pesos obtidos após o treinamento. Como o *dataset* utilizado possui 699 linhas com 10 colunas, isso requererá cerca de 13.980 *words* de 16 *bits*, tendo assim espaço suficiente para realizar o treinamento proposto.

4.6 Tratamento dos Dados

Para ser utilizada uma base de dados, ela deve ser tratada e adequada para o uso no dispositivo FPGA. Foi utilizada para o treinamento o *dataset Wisconsin breast cancer* retirado do UCI *Repository* (WOLBERG, 1992). Os dados a princípio, estão em forma de texto e precisam ser enviados para espaços da memória SRAM reservados para tal.

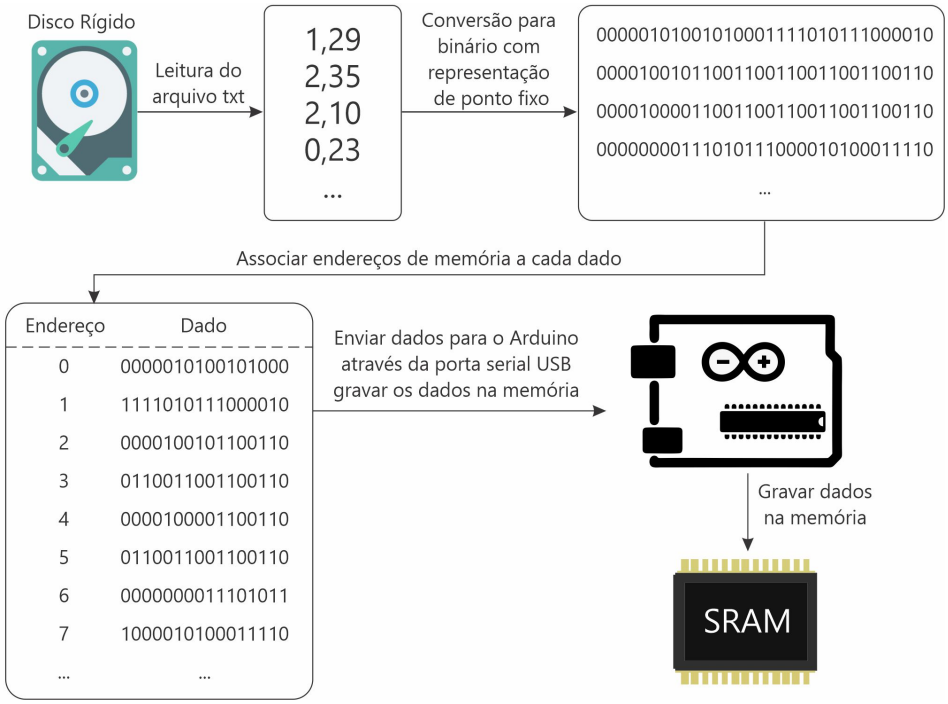
4.6.1 Software para converter dataset

Para se treinar uma rede neural, é necessário um *dataset* para ser o gabarito da rede, para que dessa forma ocorra um aprendizado supervisionado. Grande parte das bases encontradas online, sobretudo no repositório da UCI estão no formato de texto. Para ser possível utilizá-las em nossa rede, precisamos convertê-las no formato em que a rede consiga ser treinada no dispositivo FPGA.

Visto que os dados do *dataset* utilizado estavam no formato de texto com representação numérica de base decimal, fez-se necessário a criação de um *software* para fazer a conversão destes dados. O *software* elaborado, fez a leitura do arquivo e os convertia para binário com representação de ponto fixo conforme especificado para este projeto.

Além de converter os dados, o *software* também faz a organização dos dados nos endereços corretos para se colocar na memória SRAM, pois dessa forma, comunicando com o Arduino conectado à porta USB, através de um protocolo pré-estabelecido, o *software* pode enviar o endereço de memória e o dado a ser armazenado. A figura 17 ilustra o funcionamento do *software*.

Figura 17 – Fluxo do software desenvolvido.



Fonte: O Autor

Semelhantemente ao modo como o *software* converteu os dados decimais para binário, este também é capaz de fazer o caminho inverso, convertendo de binário com a representação de ponto fixo, para decimal. Isto se faz necessário pois ao final do treinamento, objetiva-se obter os pesos que estarão gravados na memória SRAM e estarão, portanto, em binário. Ao receber estes dados, o programa os exibe na base decimal no console do Java.

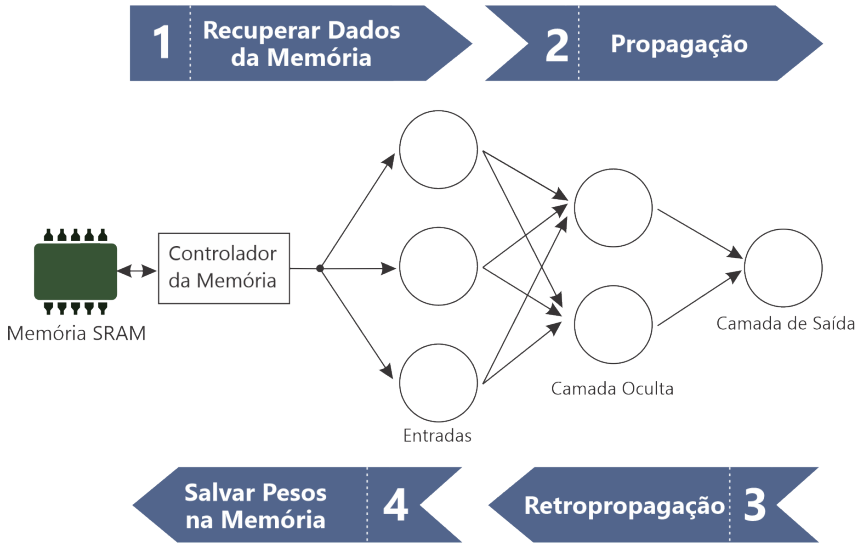
4.6.2 Comunicação com Memória SRAM

A memória SRAM utilizada IS61LV25616, a partir de um endereço e um dado, consegue gravar ou ler os dados instantaneamente, isto é, sem a necessidade ciclos de *clock* (com um *delay* de até 15 ns). Para comunicar diretamente com ela, foi utilizado um Arduino MEGA conectado à memória e conectado também à entrada USB do computador de forma a comunicar com o software em Java através da porta serial. Por meio de um protocolo pré-estabelecido de comunicação com o computador, o arduino recebe o endereço pronto e os dados apenas os grava na memória SRAM. O Apêndice A descreve como foi implementada a comunicação com o Arduino e o software em Java.

4.7 Adequações em VHDL para sintetização

Para realizar o treinamento, o código em VHDL foi dividido em quatro etapas conforme pode-se observar na figura 18: A primeira etapa, consiste em buscar os dados para executar o treinamento de uma época, após alguns ciclos de *clock*, os dados foram colocados em vetores para então serem utilizados nas etapas dois e três. Durante essas etapas é feito de fato o treinamento da base de dados, onde são feitos todos os cálculos e ajustados os pesos. Terminados os cálculos, os pesos atualizados são colocados na memória SRAM.

Figura 18 – Etapas executadas no dispositivo FPGA para o treinamento.



Fonte: O Autor

4.7.1 Problema dos laços de repetição

O algoritmo do *Multilayer Perceptron* possui diversos laços de repetição que precisam iterar de acordo com a quantidade entradas, neurônios da camada intermediária e saídas. Além disso, é preciso iterar sob cada amostra do *dataset* escolhido.

A princípio, utilizou-se processos da linguagem VHDL para fazer essas iterações, porém cada iteração de um laço de repetição em VHDL gera a necessidade de criação de mais blocos lógicos, pois as operações de todos os processos são feitas em paralelo. Então, utilizar o comando FOR nos processos para todos os *loops* existentes no MLP tornava inviável a sintetização no dispositivo FPGA disponível na placa da Altera DE2.

Para contornar o problema, as instruções for foram substituídas por *loops* utilizando ciclos de *clock*. Dessa forma, em um processo que executaria todas a iterações de uma vez, as iterações são executadas a cada ciclo de *clock*. O código abaixo, exemplifica a forma como foi feita esta substituição.

```
1    SIGNAL clock : STD_LOGIC;
2    TYPE vetor_dados IS ARRAY (0 TO 10) OF SIGNED (7 DOWNTO 0);
3    SIGNAL dado : vetor_dados;
4
5    --PROCESSO COM LOOP EM UM CICLO DE CLOCK
6    PROCESS(clock)
7    BEGIN
8        IF(rising_edge(clock)) THEN
9        FOR i IN 0 TO 10 LOOP
10         dado(i) <= "00000000";
11       END LOOP;
12     END IF;
13   END PROCESS;
14
15   --PROCESSO COM LOOP EM 10 CICLOS DE CLOCK
16   SIGNAL j : INTEGER := 0;
17   PROCESS(clock)
18   BEGIN
19     IF(rising_edge(clock)) THEN
20       dado(j) <= "00000000";
21       IF(j < 10) THEN
22         j <= j + 1;
23       ELSE
24         j <= 0;
25       END IF;
26     END IF;
27   END PROCESS;
28
```

No primeiro processo exemplificado no código acima é possível observar a instrução FOR setando dados no vetor dado. O vetor será inteiramente alterado a cada ciclo de *clock*. no segundo processo, podemos ver que o sinal j, responsável por definir a posição do vetor a ser alterada, é incrementado a cada ciclo de *clock* fazendo com que o vetor seja alterado completamente em 10 ciclos de *clock*. Dessa forma, é possível reduzir grandemente a quan-

tidade de blocos lógicos utilizado, porém nos dando uma perda de desempenho. Este tipo de escolha faz parte da decisão de projeto, se será utilizado um *hardware* mais avançado ou se irá diminuir o desempenho para reduzir custos.

4.7.2 Hierarquia de Memória

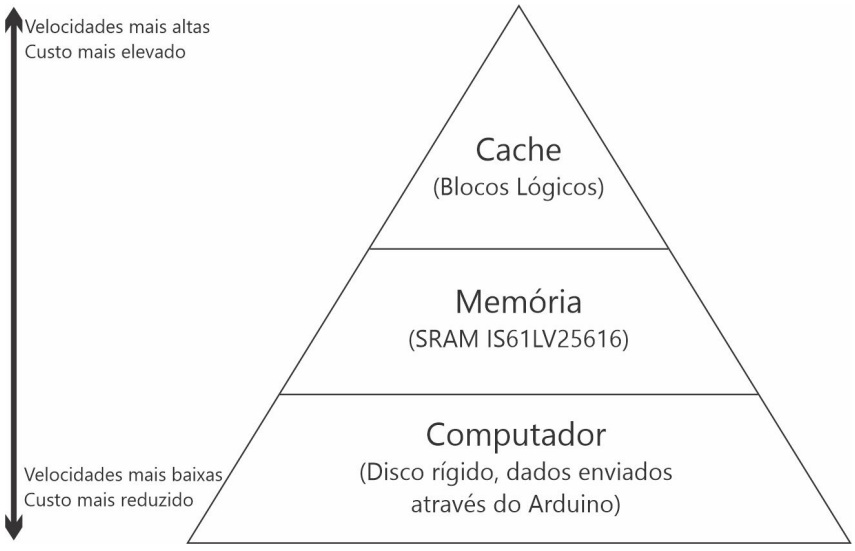
Para realizar o treinamento de uma rede com aprendizado supervisionado, é necessário um *dataset* com uma grande quantidade de amostras. O *dataset* utilizado neste trabalho por exemplo, possui 699 amostras.

Durante a implementação da rede em VHDL, a princípio utilizou-se uma matriz com todas as amostras do *dataset* com seus dados de entrada e saída, porém a utilização desta matriz gerava um grande consumo de blocos lógicos que apesar de permitir a simulação, torna inviável a sintetização no dispositivo FPGA Cyclone II 2C35.

Para solucionar este problema, ao invés de criar uma matriz com todas as amostras do *dataset*, criou-se uma matriz com espaço para apenas duas amostras, então a cada duas amostras treinadas, é pausado o treinamento, busca-se na memória os dados das próximas duas amostras a serem treinadas e então é continuado o treinamento do *dataset*.

Desta forma, foi gerado um modelo de hierarquia de memória para este projeto. Conforme é possível observar na figura 19, no topo da memória, com um custo mais elevado, porém com velocidades de acesso mais altas, estão os blocos lógicos que estão cumprindo a função de cache, ao meio a memória SRAM e em baixo o computador em que está armazenado os dados de onde se obtém o *dataset* através do Arduino utilizado.

Figura 19 – Hierarquia de memória implementada no projeto.



Fonte: O Autor

5 Resultados e Discussão

“É importante extrair sabedoria de diferentes lugares.
Se você levá-la de apenas um lugar, se tornará rígido e velho.”
Iroh

Após implementadas todas as ferramentas descritas neste trabalho, foi possível executar o algoritmo de treinamento de uma base de dados utilizando uma rede neural perceptron de múltiplas camadas em um dispositivo FPGA.

5.1 Treinamento da Porta Lógica E

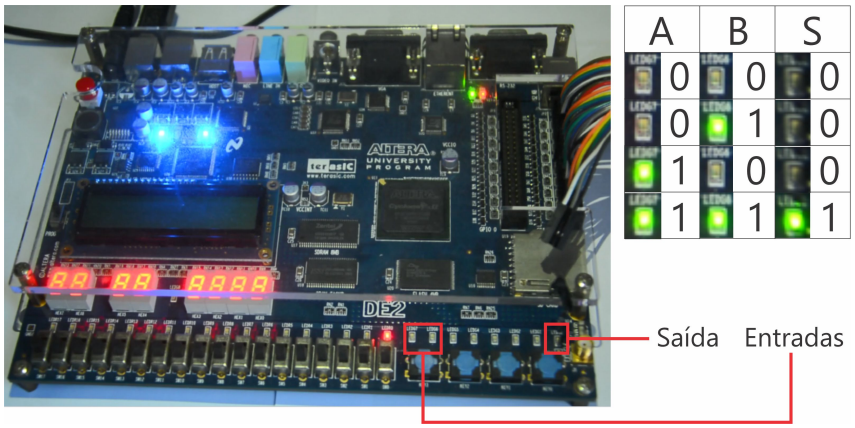
Primeiramente foi implementado o MLP utilizando como *dataset* a porta lógica E. Dessa forma foram feitos testes utilizando ModelSim e conforme é possível ver na tabela 1, após cerca de 200 épocas de treinamento, os pesos extraídos foram bem próximos. Na tabela, WO representam os pesos dos neurônios da camada de saída e WH da camada intermediária.

Tabela 1 – Tabela dos pesos extraídos da rede treinada.

Java			**VHDL**		
WO	**0**	**1**	**WO**	**0**	**1**
0	-1,28802305110430	-1,28802305110430	**0**	-1,28803159296512	-1,28803159296512
1	-1,28802305110430	-1,28802305110430	**1**	-1,28803159296512	-1,28803159296512
2	0,05747077091914	0,05747077091914	**2**	0,05747547745705	0,05747547745705
WH	**0**	**1**	**WH**	**0**	**1**
0	0,50275776154913	0,50275776154913	**0**	0,50275717675686	0,50275717675686
1	-0,99833792660068	-0,99833792660068	**1**	-0,99833923578262	-0,99833923578262
2	-0,98113553381517	-0,98113553381517	**2**	-0,98113676905632	-0,98113676905632

Após feitos os testes no ModelSim, o *hardware* descrito foi sintetizado no dispositivo FPGA disponível na placa da Altera DE2. Então foram atribuídas chaves e LEDs para representar as entradas da porta lógica E e um LED para representar a saída, e dessa forma foi possível verificar que a rede estava classificando as entradas de maneira correta conforme é possível observar na figura 20. Os LEDs ligados representam estado alto, e desligados representam estado baixo.

Figura 20 – Placa Altera DE2 utilizada com a tabela dos resultados obtidos do treinamento da porta lógica E.



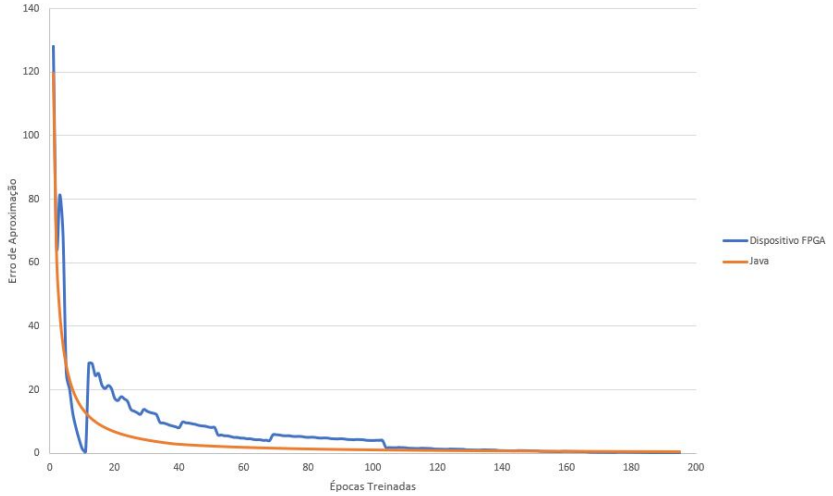
Fonte: O Autor

5.2 Treinamento da Base de Câncer de Mama

5.2.1 Erro de Aproximação no treinamento utilizando o Ciclone II

Posteriormente a implementação do MLP e da sintetização do *hardware*, para validação dos testes, foi calculado o erro de aproximação no treinamento da rede neural por época e comparado com os resultados em Java. Observando o gráfico da figura 21, é possível notar que houve uma disparidade no treinamento no Java e no dispositivo FPGA. Essa discrepância pode ter ocorrido devido à diferente resolução utilizada, no dispositivo FPGA foram utilizados 32 *bits* com representação de ponto fixo, enquanto no Java são 64 *bits* com ponto flutuante. Apesar da diferença, o erro da rede após algumas épocas chegou a valores bem próximos de zero, e a diferença com Java tornou-se mínima conforme incrementam-se as épocas.

Figura 21 – Gráfico comparando o erro de treinamento por época entre Java e o dispositivo FPGA.

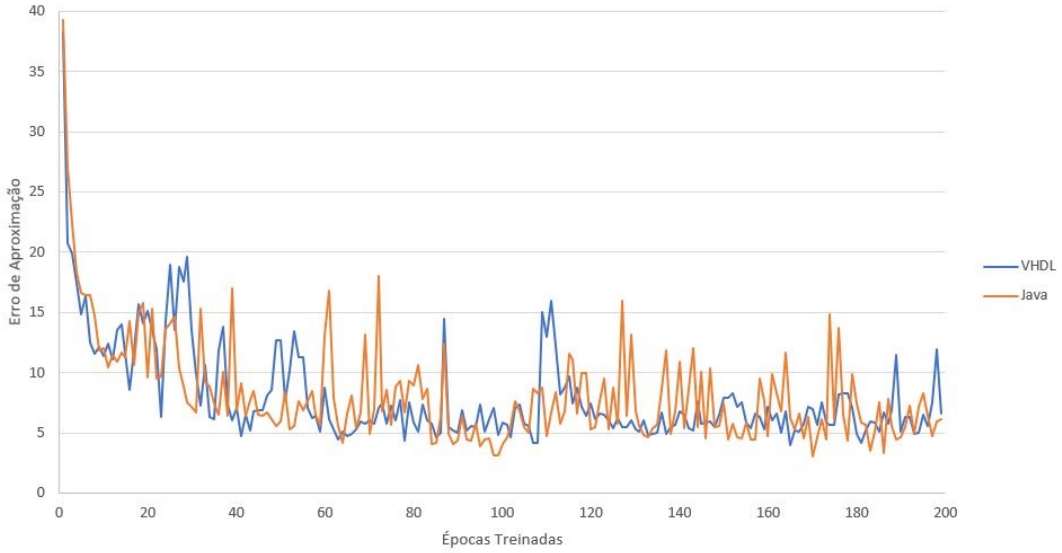


Fonte: O Autor

5.2.2 Erro utilizando base de teste

Além do erro durante o treinamento, para validar o algoritmo, o *dataset* utilizado foi dividido em base de treinamento, com cerca de 75% dos dados, e base de teste, com cerca de 25% dos dados. Desta forma, foi possível observar o comportamento da rede neural conforme variam as épocas. O gráfico da figura 22 mostra a comparação entre o erro de aproximação obtido para cada época na simulação utilizando o software ModelSim e a rede executado na linguagem Java.

Figura 22 – Erro de Aproximação utilizando Base de Teste.



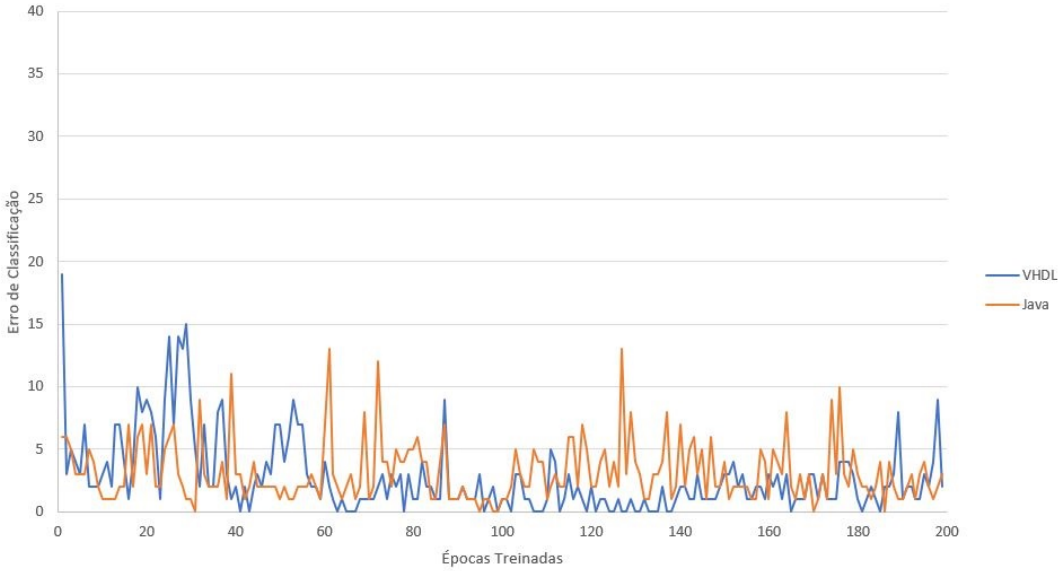
Fonte: O Autor

Outra forma de avaliar o treinamento da rede, é utilizando o erro de classificação. Enquanto o erro de aproximação é calculado a partir do módulo da diferença entre da saída obtida pela rede e a saída desejada (utilizando o gabarito), o erro de classificação é calculado de forma similar, porém é aplicado um *threshold* na saída obtida antes de se fazer a subtração.

O *threshold* é aplicado aproximando a saída obtida para o valor mais próximo da saída desejada. No exemplo do dataset utilizado, todos os valores de saída encontram-se entre 0 e 1, logo, se o valor da saída obtida for maior que $0,5$, a saída é aproximada a 1, caso contrário, é aproximada a 0. A técnica de aproximação numérica é válida pois ajuda a reduzir o número de erros acumulados por aproximação.

No gráfico da figura 23, é possível observar o erro de classificação tanto da rede descrita em VHDL e simulada com o ModelSim quanto o erro da rede no software desenvolvido em Java.

Figura 23 – Erro de Classificação utilizando Base de Teste.



Fonte: O Autor

5.3 Velocidade de Treinamento

Outro aspecto analisado neste capítulo, é o tempo que a rede neural levou para treinar a base de dados em VHDL e em Java. Entretanto, é fundamental ressaltar que as comparações feitas nesta seção são feitas com tecnologias com grandes diferenças de arquitetura e de gerações diferentes, portanto, para se fazer uma análise mais profunda destes resultados, deve ser levado em consideração outras variáveis não abordadas neste trabalho. Contudo, é possível fazer uma avaliação crítica dos dados coletados.

No treinamento da rede neural em Java, conforme mostrado na seção 3.1 deste trabalho, utilizou-se um computador com processador Intel Core i5-7200 com velocidade de até 2.71GHz e com memória RAM de 8GB DDR4 com velocidade de até 2133 MHz. Para medir o tempo de execução, utilizou-se o método *currentTimeMillis* da classe *System* do Java, treinou-se a rede por cerca de 100000 épocas e dividiu-se o tempo para quantidade de épocas a fim de comparar. Esta medição foi repetida por 10 vezes e o resultado apresentado é uma média simples dos dados coletados.

Para calcular o tempo de treinamento do dispositivo FPGA, foi utilizado a simulação utilizando o *software* ModelSim para se obter a quantidade de ciclos de *clock* necessários para treinar uma época. Visto que a memória SRAM utilizada possui um *delay* de 15 ns para leitura, podemos trabalhar com um *clock* de até cerca de 66 MHz, todavia os testes foram feitos utilizando 65 MHz.

Nos dados de tempo utilizando VHDL foram levadas três códigos em consideração. O primeiro, utilizando o sistema de *cache* implementado além das retiradas dos comandos FOR substituídos por mais ciclos de *clock* para que a quantidade de blocos lógicos sejam compatí-

veis com o dispositivo FPGA Cyclone II 2C35 com cerca de 33 mil blocos lógicos. O segundo, trazendo o treinamento utilizando os comandos FOR, que por consequência levaram a utilização de mais blocos lógicos, aproximadamente 128 mil. Por último, o *hardware* utilizando uma matriz de blocos lógicos que suporta todo o dataset sem a necessidade de constantemente buscar dados na memória, porém faz-se necessário a utilização de cerca 444 mil blocos lógicos.

É relevante destacar também que no *software* em Java foi desconsiderado o tempo de leitura do arquivo de texto contendo o *dataset* e no VHDL o tempo para se gravar os dados na memória SRAM.

Linguagem	**Tipo de Implementação**	**Família do dispositivo FPGA**	**Blocos Lógicos**	**Tempo de Treinamento por Época (ns)**
Java	-	-	-	359,33
VHDL	Compilado no Quartus II, Sintetizado no FPGA	Cyclone II	28K	440,24
VHDL	Compilado no Quartus II, Simulado no ModelSim	Cyclone III	128K	243,70
VHDL	Compilador no Quartus II, Simulado no ModelSim	Arria V GX	444K	55,03

Tabela 2 – Comparação entre o tempo necessário para treinar uma época em diferentes cenários.

Conforme é possível observar, o maior gargalo no dispositivo FPGA, é a busca de dados na memória. A solução com adição de mais blocos lógicos na sintetização oferece bem mais performance durante o treinamento. Apesar de ser utilizado neste trabalho o dispositivo FPGA Cyclone II EP2C35F672C6N que possui 33216 elementos lógicos, dispositivos FPGA da linha Cyclone III LS tem versões que possuem 198464 elementos lógicos, como é o caso do modelo EP3CLS200F780I7. Além destes, dispositivos da serie Arria V GX da Intel, como por exemplo o modelo 5AGXFB3H4F35C4N, possui 450000 elementos lógicos, entretanto possuem um custo bem mais alto.

Uma outra solução possível para diminuir o tempo de treinamento é trocar o *chip* de memória SRAM utilizado por um que permite operar em frequências mais altas, como por exemplo o *chip* CY7C1314KV18-300BZXC da Cypress que pode operar em até 300 MHz. No caso do primeiro cenário, com o Cyclone II, é possível aumentar a frequência do *clock* para até 260 MHz, que é a faixa máxima de operação do dispositivo, isto nos daria um tempo de treinamento de 110,06 ns para uma época. No segundo cenário, utilizando Cyclone III LS , é possível operá-lo com uma frequência máxima de 274 MHz possibilitando treinar uma época com 57,81 ns. No caso do terceiro cenário, a velocidade do *clock* é limitada apenas pela faixa de operação do dispositivo FPGA utilizado, que no caso de dispositivos da Arria V GX, trabalham numa frequência máxima de 710 MHz, o que nos permite treinar uma época em 5,03 ns.

Todos os cenários são possíveis implementações válidas, e as diferentes possibilidades de uso de memória, dispositivos FPGA, dentre outros, são decisões de projeto, quanto

maior a performance desejada, maior será o custo do projeto final.

6 Conclusão

“Guarde consigo a sensatez e o equilíbrio,
nunca os perca de vista.”
Provérbio de Salomão

O *hardware* projetado mostrou-se eficaz naquilo em que se propõe a solucionar. Este trabalho compementa os trbalhos relacionados trazendo detalhes de implementação e sintetização da rede neural em um dispositivo FPGA. A rede neural realizou o treinamento da base de dados com sucesso e através da integração criada é possível enviar diversas bases de dados à memória e da mesma forma obter os pesos do treinamento. Apesar da dissemelhança no erro de treinamento da rede neural entre Java e VHDL, ambas se aproximam de zero conforme são treinadas as bases, o que é esperado de uma rede neural. Foi possível observar também que o o erro de aproximação e classificação da base de teste foram coerentes ao comparar VHDL e Java.

Devido à limitação de blocos lógicos, algumas estratégias tiveram de ser implementadas para contornar a restrição física do Cyclone II 2C35, o que compromete em partes o desempenho do algoritmo ao ser executado. Porém, utilizando dispositivos mais novos e de gerações mais recentes, é possível implementar esta rede neural utilizando mais blocos lógicos, trazendo assim por consequência uma maior performance.

Embora tenha-se utilizado um *dataset* específico neste trabalho, com poucas alterações é possível treinar outros *datasets* e obter os resultados. O *software* criado em Java permite converter qualquer *dataset* numérico e colocá-lo na memória SRAM da placa da Altera DE2 desde que haja espaço para tal. Além disso, o *software* também obtém os pesos ao final do treinamento.

6.1 Trabalhos Futuros

Os resultados obtivos mostram que outras experiências com trabalhos futuros podem ser desenvolvidos de maneira a buscar objetivos diferentes no trabalho. Podemos destarcar que:

- Pode-se implementar algoritmos para fazer o cálculo de divisão de números reais utilizando outros métodos numéricos e avaliar a efetividade.
- É possível realizar testes sintetizando o código descrito em outros dispositivos FPGA a fim de comparar seus resultados.
- Desenvolver um controlador para comunicar diretamente com o computador sem a necessidade de um microcontrolador intermediando a comunicação.

- Pesquisar e utilizar Microcontroladores associados com dispositivos FPGAs em uma única plataforma eletrônica de maneira a suprir os gargalos entre comunicações

Referências

ALTERA. Altera de2 board manual. *Altera Corporation*, v. 72, 2006. Citado na página 26.

Arduino. *Getting Started with Arduino MEGA2560*. 2018. Acessado em: 20 nov. 2020. Disponível em: <https://www.arduino.cc/en/Guide/ArduinoMega2560>. Citado na página 25.

D'AMORE, R. *VHDL: Descrição E Síntese de Circuitos Digitais* . [S.l.]: Grupo Gen-LTC, 2000. Citado nas páginas 12 e 15.

ELLIOTT, D. L. *A better activation function for artificial neural networks*. [S.l.], 1993. Citado nas páginas 19 e 30.

GERHARDT, T. E.; SILVEIRA, D. T. *Métodos de pesquisa*. [S.l.]: Plageder, 2009. Citado na página 24.

HARIPRASATH, S.; PRABAKAR, T. Fpga implementation of multilayer feed forward neural network architecture using vhdl. In: IEEE. *2012 International Conference on Computing, Communication and Applications*. [S.l.], 2012. p. 1–6. Citado nas páginas 11, 21, 22 e 30.

Hariprasath, S.; Prabakar, T. N. Fpga implementation of multilayer feed forward neural network architecture using vhdl. In: *2012 International Conference on Computing, Communication and Applications*. [S.l.: s.n.], 2012. p. 1–6. Citado nas páginas 11 e 12.

HAUCK, S.; DEHON, A. *Reconfigurable computing: the theory and practice of FPGA-based computation*. [S.l.]: Elsevier, 2010. Citado nas páginas 14 e 15.

HAYKIN, S. *Redes neurais: princípios e prática*. [S.l.]: Bookman Editora, 2007. Citado nas páginas 11, 12, 16, 17, 18 e 20.

HWANG, E. O. *Digital Logic and Microprocessor Design with Interfacing*. second. [S.l.]: Cengage Learning US, 2018. Citado na página 15.

KAFIG, W. *VHDL 101: Everything you need to know to get started*. [S.l.]: Elsevier, 2011. Citado na página 15.

KAPTANOGLU, S. et al. A new high density and very low cost reprogrammable fpga architecture. In: *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*. [S.l.: s.n.], 1999. p. 3–12. Citado na página 11.

KRÖSE, B. et al. An introduction to neural networks. Citeseer, 1993. Citado na página 18.

LANNA, A. B. Os impactos socio-econômicos da inteligência artificial. *ConTextura*, v. 10, n. 12, 2018. Citado na página 11.

LIVSHIN, I. *Artificial Neural Networks with Java*. [S.l.]: Springer, 2019. Citado na página 16.

OMONDI, A. R.; RAJAPAKSE, J. C. *FPGA implementations of neural networks*. [S.l.]: Springer, 2006. v. 365. Citado na página 20.

RAJEWSKI, J. *Learning FPGAs: Digital Design for Beginners with Mojo and Lucid HDL*. [S.l.]: "O'Reilly Media, Inc.", 2017. Citado na página 14.

RASHID, M. H. *Microelectronic Circuits: Analysis and Design*. third. [S.l.]: Cengage Learning US, 2017. Citado nas páginas 13 e 14.

RESTREPO, H. F. et al. A networked fpga-based hardware implementation of a neural network application. In: IEEE. *Proceedings 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No. PR00871)*. [S.l.], 2000. p. 337–338. Citado nas páginas 11 e 21.

SOUZA, H. A. d. et al. Treinamento de redes neurais com arquitetura multilayer perceptron em fpga. Florianópolis, SC, 2019. Citado nas páginas 22 e 23.

TEIXEIRA, J. *O que é inteligência artificial*. [S.l.]: E-Galáxia, 2019. Citado na página 11.

TOCCI, R. J.; WIDMER, N. S.; MOSS, G. L. *Sistemas digitais*. [S.l.]: Pearson Educación, 2010. Citado na página 13.

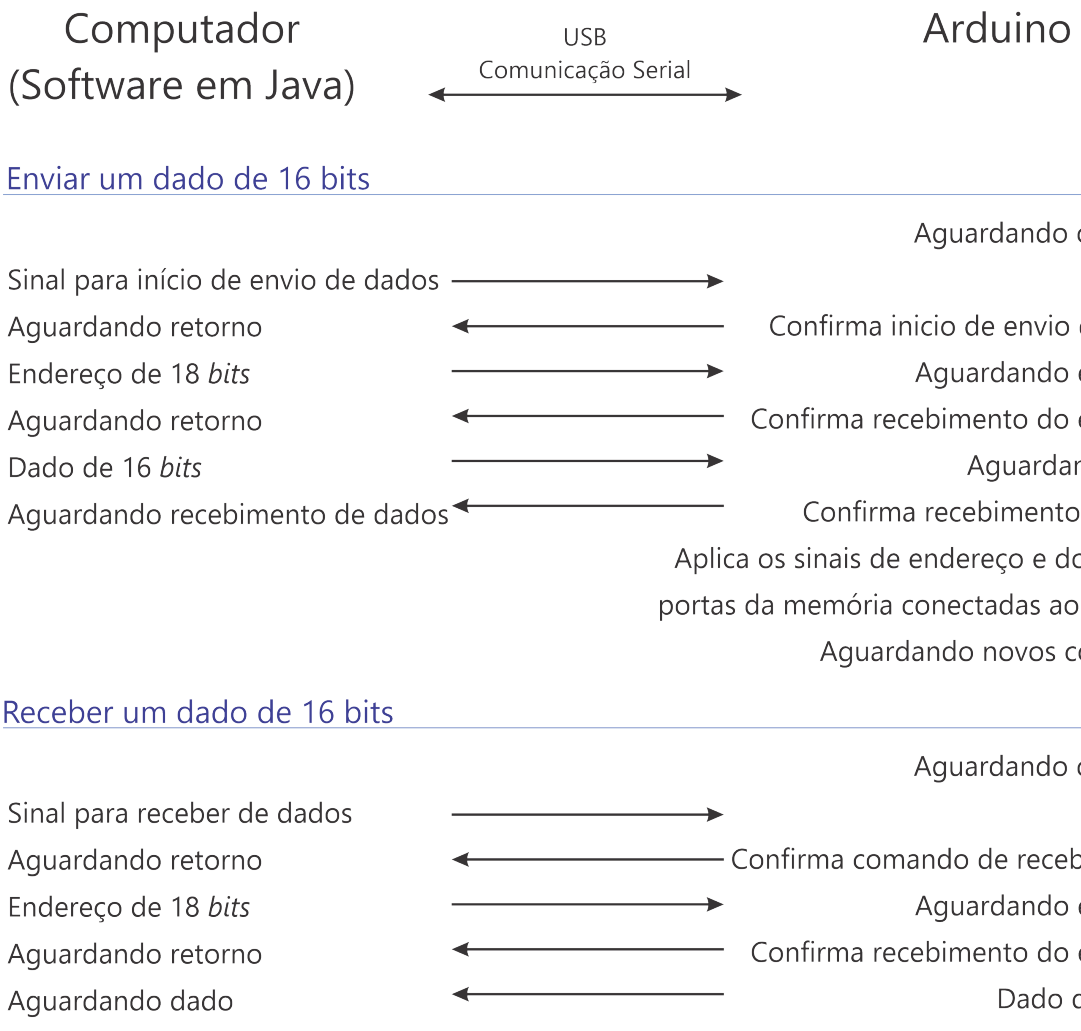
UCI Machine Learning Repository. *Breast Cancer Data Set*. 1998. Acessado em: 20 nov. 2020. Disponível em: <https://archive.ics.uci.edu/ml/datasets/breast+cancer>. Citado na página 12.

WOLBERG, W. H. *UCI Machine Learning Repository*. 1992. Acessado em: 20 nov. 2020. Disponível em: <https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original)>. Citado na página 33.

APÊNDICE A – Fluxo de comunicação entre Arduino e o Software em Java

Neste apêndice será descrito o fluxo de comunicação entre o Arduino e o software em Java para que sejam gravados dados na memória SRAM utilizada ou para obter dados da mesma. A figura 24 mostra como foi feito o fluxo de comunicação para recuperar ou gravar dados na memória utilizando Arduino.

Figura 24 – Esquema de comunicação entre Arduino e o Software em Java.



Fonte: O Autor